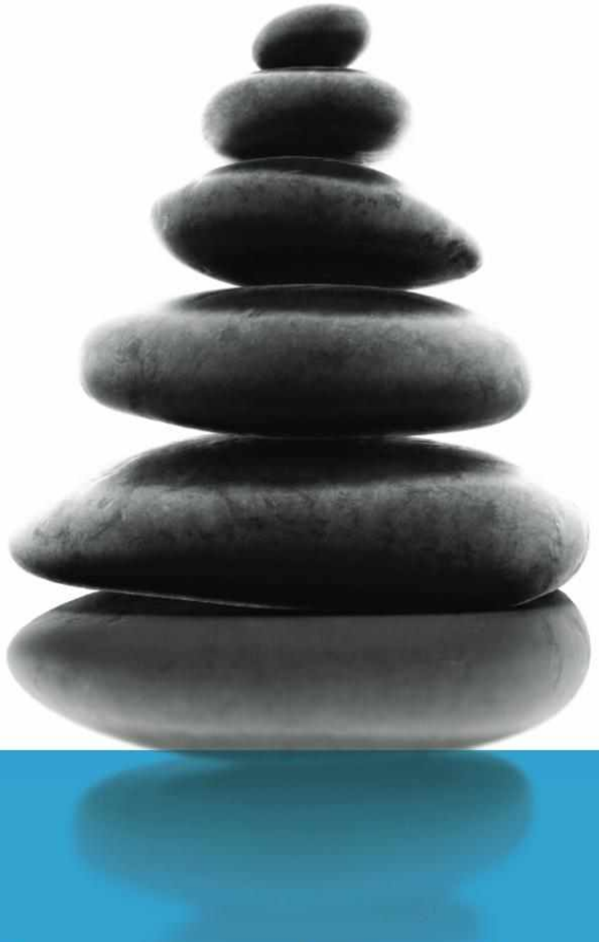

Six-Step Relational Database Design™

A step by step approach to relational database design and development

S e c o n d   E d i t i o n

Fidel A. Captain

Six-Step Relational Database Design™

A step by step approach to relational database design and development

Second Edition
Revised

FIDEL A. CAPTAIN

Six-Step Relational Database Design™

First published: April 2012
Second edition: May 2013
Revised second edition: May 2015

Editor: Richard W. Georges

Cover Design: Wilbert Chambers

ISBN: 1481942727
ISBN-13: 978-1481942720

For my daughter, Ariel Christina. You mean the world to me.

CONTENTS

About the Author

Fidel A. Captain has over fifteen years of experience designing, implementing, and maintaining databases, and over ten years of experience as a lecturer.

He has worked as a systems analyst and systems engineer with the Ministry of Finance in Guyana, as a computer studies lecturer at the H. Lavity Stoutt Community College in the British Virgin Islands, and as a freelance consultant.

Captain has developed several database-based applications that include an online bill- payment system, a logistics and cargo-tracking system, a student-tracking and evaluation system, and a freight and duty calculation system.

He currently lectures at the H. Lavity Stoutt Community College in the British Virgin Islands in areas such as database design and development, website design and development, and Java programming.

Captain is a Queen’s College Guyana Scholar, who received his Bachelor’s of Engineering degree from the University of Manchester Institute of Science and Technology (UMIST) in Computer Systems Engineering and his Master’s in Information Technology from Capella University. He is also a certified MySQL developer and administrator.

Captain is happily married and has a daughter. You can find out more about him at www.fidelcaptain.com.

Acknowledgments

The educational and technological team of CRC SOGEMA initially taught me the technique that is used in this book in the late 1990s, and I am grateful to them for this.

Sauda Smith was very helpful in helping me to choose the direction I took in getting the book published, and I am very grateful for her wise words of advice.

A special thank you to Richard Doherty of the University of Alabama and to his students for providing feedback that went into changes in this revised edition.

Recommendations from other students, lecturers, and readers were also instrumental in making the changes that appear in this revised edition.

Thank you all.

Preface

Changes in this edition

Since the publication of the first edition, there has been a lot of positive feedback and recommendations from students, lecturers, and other readers. This has resulted in the addition of questions at the end of each of the six steps, which can be used to test the reader's understanding of each of these steps; the replacing of one of the previous case studies, making the selection of case studies more diverse; and the inclusion of a chapter on implementation, which more clearly relates the design to implementation and database theory. There are also some other minor changes that make understanding the six-step relational database design process easier.

Six-Step Relational Database Design

Database design is the most important task performed by application developers because the resulting database and all applications that access this database are based on this design. Therefore, if the data for the application is modeled incorrectly and the resulting database poorly designed, poor and difficult to use applications will result. This book is dedicated to structuring and simplifying the database design process by outlining a simple but reliable six-step process for accurately modeling user data, which leads to a sturdy and reliable relational database.

This book bridges the gaps between database theory, database modeling, and database implementation. It gives a simple, step-by-step, easy to follow, and precise technique for designing a relational database that results in a sturdy, reliable and accurate data model that can be implemented on any relational database platform.

It starts with a statement of the problem by the client and goes through the six steps necessary to create a reliable and accurate data model of the client's business requirements. At each stage the reader is told what the inputs are for that stage, what the process is at that stage, and what the outputs are for that stage. The end result is a Crow's Foot Relational Model of the client's business, which can then be implemented on any relational database.

The six-step relational database design technique is one that I have used faithfully for over ten years, for many clients, to design and implement small and medium-sized databases consisting of fewer than fifty tables. I have also taught the six-step technique to many of my students over the same period.

Audience

This book should be used as a handbook for students and professionals in the software-development field. Students can use it as a technique for quickly developing relational databases for their applications, and professionals can use it as a technique for developing sturdy, reliable, and accurate relational database models for their software applications.

Structure of this book

The book uses three case studies to guide the user through the six steps, illustrating the six-step relational database design technique. The cases are introduced in Chapter 2, and at each stage the technique is explained, in detail, using the case studies as an example of how to implement the process for that stage of the technique.

Chapter 8 goes through the six steps for each of the three case studies used throughout the book without the details or explanations. This chapter can be used as a starting point for those familiar with relational database design to get a synopsis of the six-step process. It can also be used as a summary chapter for the six-step process, where each step is summarized and demonstrated.

Finally, Chapter 9 goes through the steps necessary to implement each of the case studies on a relational database management system (RDBMS), thus more clearly relating the design to implementation. This chapter can be used in combination with Chapter 8 to demonstrate the seamless flow from database design to database implementation. It can also be used independently to show how Crow’s Foot Relational Model designs can be easily implemented on RDBMSs.

Companion website

http://fidelcaptain.com/sixstep.html

Here you can find additional self-study tools and resources for material covered in the book:

- **Case studies** – All of the case studies used in the book are online, and include the E-R diagrams, R-M diagrams, and SQL code.

- **Video lectures** – Downloadable video lectures that cover all of the material in the book are available online, and can be used as teaching and learning aids.

- **Online courses** – A combination of the video lectures and material from the book are combined to provide a comprehensive eLearning experience provided by the Udemy, *“the world’s online learning marketplace*”.

Introduction

So you are a programmer, or an analyst, or a software developer, or a student taking a database or software development class, and you have a problem – you must design and develop a database for an application as soon as possible. Like yesterday. Where do you start? What do you do? How do you go about quickly designing and developing a database – one that correctly models the data your client wants to capture and track?

The six-step technique detailed in this book will allow you to quickly design a relational model for your database. A model that is sturdy, reliable, accurate and that can be implemented on any relational database management system (RDBMS). This technique will describe everything you need to do, from start to finish, allowing you to create an accurate data model that you can use for your application, one that you can use to implement your database on any RDBMS.

A database is the backbone of any application, and therefore it must be sturdy, reliable, and accurate. Having a well-designed database is the key to having your database meet these criteria, and the six-step technique ensures that the data model upon which your database is based is sturdy, reliable, and accurate. A design that is sturdy, reliable, and accurate is the key to developing a database that is sturdy, reliable, and accurate.

The six steps outlined in this book are simple, easy to follow, and precise. Extensive knowledge of database theory is not required for you to be able to understand and follow the steps because everything is explained in detail, and the theory is kept to a minimum. The focus of this book is on developing a sound relational model for your data. Just follow the steps from the beginning, and every time you need a database for an application, or just a database, you should be home and dry in six easy steps.

While a database is a part of almost any application, not many developers give it the time or attention it deserves. This lack of attention usually comes back to haunt them in the end because the database is an integral part of the application, and the application is usually built around accessing data held in the database. So even if the application is well designed, poor data design will inevitably result in a poor application. The six-step process described in this book allows developers to correctly model the data that their application will use and shows them how to do it in the shortest possible time.

In preparation for the six steps you must have a basic understanding of relational databases and relational database design. You need to be familiar with the basics of relational databases and relational database design.

There are six steps to creating a sturdy, reliable, and accurate relational model for your data:

Step 1: Discover entities and assign attributes

Before you can attempt to solve any problem or model data for any client, you need to understand the problem. This involves a careful analysis of the processes and documentation used by the client in the problem domain. Having performed your analysis, you then need to carefully and meticulously document your findings – the entities and attributes.

Step 2: Derive unary and binary relationships

Using an Entity-Entity Matrix (E-E Matrix) is one of the fastest methods for deriving all the relationships among all the entities. It is possible that no relationship will exist between entities, that more than one relationship will exist between entities, and that an entity will be related to itself.

Step 3: Develop a simplified Entity-Relationship diagram

Use the information from the E-E Matrix to construct a simple initial Entity-Relationship (E-R) diagram. This diagram will contain the entities and their basic attributes and all the relationships among the entities as indicated in the E-E Matrix.

Step 4: List assertions for all relationships

Derive optionalities and cardinalities for *each* relationship in the initial E-R diagram and use these to record the assertions for *each* relationship.

Step 5: Create a detailed E-R diagram using the assertions

Construct a detailed E-R diagram that includes all the derived information from the assertions. This diagram will contain the optionalities, cardinalities, attributes, identifiers, and relationships.

Step 6: Transform the detailed E-R diagram into an implementable relational database design

Once the detailed E-R diagram is complete, it will conform to certain rules and will be in a specific format. Once in this format, it is easy to translate it into a relational database model diagram – specifically, a Crow’s Foot Relational Model diagram.

Before you can start the six steps you must know what you are about, hence, “Chapter 1: Relational Databases and Relational Database Design.” This is just to whet your appetite and get you ready for the first step.

Chapter 1

Relational databases and relational database design

Relational databases are ubiquitous. Data is collected on everything and stored on a server somewhere in a relational database. Although there are other forms of databases out there, relational databases are the most trusted and widely used type of database. They are the core of most content-management systems for new websites developed and form part of the persistent storage system of new software.

This chapter gives some background on relational databases and relational database design. It gives a brief history of relational databases, along with a formal definition of the term. This is followed by terms commonly used in the industry and terms that will be used in this book. Next, this chapter takes a look at relational database design and the various types of models that can be used to represent a database's design. The chapter ends with an example of how the relational database design process as explained in this book works, relating it to the four main *stages* of database development and the two types of database models.

Definition and History

There are many definitions for the term *relational database*, but the definition used in this book will slant towards the theoretical end, as this definition is more accurate and relevant to the context of the content of this book. The term *database* has its roots in US military information systems in the mid-1960s, and the term *relational* has its roots in the 1970 paper by E. F. Codd, “A Relational Model of Data for Large Shared Data Banks.” Peter Chen’s 1976 paper, “The Entity Relationship Model—Toward a Unified View of Data”, also contributed heavily to what we know today as the *relational database*.

Definition

A relational database can be seen as a collection of organized and inter-related data on a related subject or topic. This data is organized into sets (relations or tables), and each *set* of data may be related to another *set* of data, and there are rules regarding how each *set* of data is made up and how one *set* of data can be related to another *set* of data.

History

Edgar F. Codd and Peter Pin-Shan Chen can be considered godfathers of what we refer to today as the *relational database*. While the term *database* was in use for a few years before Codd's paper, his paper was the first to propose a *relational* view of data manipulated by computer systems. Chen's 1976 paper formalized Codd's *relational* term in what Chen called the *Entity-Relationship* model, which is still in use today.

Although Oracle claims that its Oracle Version 2.0, released in July 1979, is the first commercial relational database management system (RDBMS), that title actually belongs to the Multics Relational Data Store (MRDS), which was released in June 1976 and was designed for Honeywell Information Systems, Incorporated. Oracle, through mergers and acquisitions, is one of the largest RDBMS vendors today.

In 1983 IBM joined the RDBMS race with its release of DB2, which was a direct result of the work done by Edgar F. Codd while at IBM. DB2 also brought with it IBM's release of its version of SQL, a language for querying databases that is still in use today. SQL was first adopted as an ANSI standard in 1986 and an ISO standard in 1987.

Originally called Vulcan, dBase was developed in 1979 by Wayne Ratliff at NASA's Jet Propulsion Laboratory and became the most popular database software in the 1980s. It was different because it was a database management system as well as a programming language, and it was *open*. The fact that dBASE was open and programmable led to many flavors, which included FoxPro and Clipper/Harbour.

Ashton-Tate, the new owners of dBASE, Microsoft, and Sybase, worked together to create Microsoft's SQL server. The first version, version 1.0, called Ashton-Tate/Microsoft SQL Server shipped in 1989. However, it was not until 1992 with Microsoft SQL Server version 4.2 and the success of Microsoft Windows that the product took hold. It would go on to dominate the client-server market in the 1990s and the first decade of the twenty-first century.

Founded in 1995 by Michael Widenius (Monty), David Axmark and Allan Larsson, MySQL AB went *open source* in 2000 and took off a few years later as free and open source software became the mantra of the twenty-first century. It rapidly became the relational database of choice for small to medium-sized websites and web applications that involved a content management system.

Oracle acquired MySQL through its purchase of Sun Microsystems in 2009. Oracle, IBM, and Microsoft are poised to be the big three [relational] database development corporations in the twenty-first century.

Terms used in relational database design and development

Even though this book takes a non-theoretical approach to the design and development of a relational database, it is important to know the terms and phrases used in the industry. Most of these terms are rooted in database theory and are sometimes misused and misinterpreted.

Here is a list of the terms that will be used throughout this book and what they mean within the context of this book:

Entity – An entity is a tangible object of interest that exists in the user's domain. It is something of interest to the user that the user keeps track of somewhere or somehow. Collective nouns or nouns are usually used to name entities.

Relation – A Relation is a two-dimensional table that contains rows and columns. A Relation is also referred to as a table.

Tuple/Row – A tuple or row is an entity occurrence (entity instance) within a Relation (table), or a row within a table (Relation).

Attribute – An attribute is *one* of the various properties that describe the entity's characteristics, and it describes *one and only one* characteristic or property of an entity.

Column – A column is an attribute of an entity within a Relation (table), or a column within a table (Relation).

Identifier – An identifier is an attribute that can be used to identify entity occurrences (rows). They can be unique, non-unique, or composite. Unique identifiers identify one and only one entity occurrence (row), whereas non-unique identifiers can identify more than one entity occurrence (row). Composite identifiers consist of more than one attribute.

Index/Key – An index or key is a unique or non-unique identifier that can be used to speed up searches.

Primary Key – The primary key is a unique identifier that is used to distinguish or identify rows in a Relation (table).

Relationship – This is what exists between entities or between Relations (tables). Verbs are used to describe relationships, which can be one-to-one, one-to-many, or many-to-many.

Foreign Key – A foreign key is a primary key in one Relation (table) that appears as a column in another Relation (table) and is used to join the two Relations (tables) together in a relationship.

Relational Database Design

The word *design* can be used as either a noun or a verb. As a noun it represents something tangible and as a verb something intangible, a process. As a noun it represents the end product of the process, and as a verb it refers to the process itself. *The design* (noun) is the end product of the design process and is either *the plan* or the actual manifestation of the product or artifact you are creating or trying to create. *To design* (verb) is the process of creating the plan that creates the product or artifact.

This book deals with both the process and the end product of the process. It looks at the relational database design process as a six-step process and at the relational database design that is the end result of that six-step process, the relational model. So within the context of this book, relational database design refers to the process, six-steps in this case, of creating a relational database design, which is represented in the end as a Crow's Foot Relational Model diagram.

The design process sometimes involves creating a model of the artifact you are trying to create, and it is no different in relational database design. The six-step process described in this book takes you through two different models, the *entity-relationship model* and the *relational model*. The *entity-relationship model* is a **conceptual model**, and the *relational model* is an **implementation model**.

Conceptual models are concerned with the logical nature of the data and **what** is being represented, and **implementation models** are concerned with the physical nature of the data and with **how** the data will be represented in the database. Generally, conceptual models are concerned with the *user's view* of the data, and implementation models are concerned with the *developer's view* of the data.

It is important that both views of the data, the user's view and the developer's view, are congruent and accurate. It is also important that one follows from the other – that the developer's view follows from the user's view – and not the other way around. The user's view should be developed first and the developer's view derived from the user's view.

The approach taken by this book is to first develop the entity-relationship diagram, which represents the user's view of the data, then to derive the relational model diagram, which represents the developer's view of the data. The first five steps of the six-step process are dedicated to creating the relational model, which represents the user's view. The sixth, and final, step is used to derive the relational model, which represents the developer's view of the data. This model can then be used to implement the relational database on any relational database management system.

Five of the six steps are dedicated to modeling the user's view of his or her data because of the importance of the user's view of the data and the importance of getting it right. The developer's view of the data is then derived from the user's view of the data, minimizing the risk of getting the developer's view wrong. In fact, the technique described in this book all but eliminates the chance of developer-introduced error.

The subsequent example shows the relational database design process in action as outlined in this book. While this example does not follow the six-steps step by step, it highlights the four main stages of database development:

Problem Specification » Conceptual Model » Implementation Model » Relational Database

The relational database design process in action

Here is a simple example of how the relational database design process works, as explained in this book. It does not contain six-steps, but rather, the four main *stages* of database development.

Problem Specification – Customer's Requirements

Consider the following scenario:

Jim is a lecturer at a community college, and he wants an application to help him to keep track of the students he is advising. He wants to know which students register for which courses in which semester and what grade they got in those courses. He also wants the application to be password protected, with each user having a different login and password, and it must keep track of user logins/outs.

Your job as a database developer is to design a database that will serve as the *back-end* of either a web-based or client-server based application. This database must capture and store all the relevant persistent data that will be used by the application.

Conceptual Model – E-R Diagram

The diagram overleaf shows the result of the first five steps in the six-step relational database design process, the detailed entity relationship model diagram (E-R Diagram).

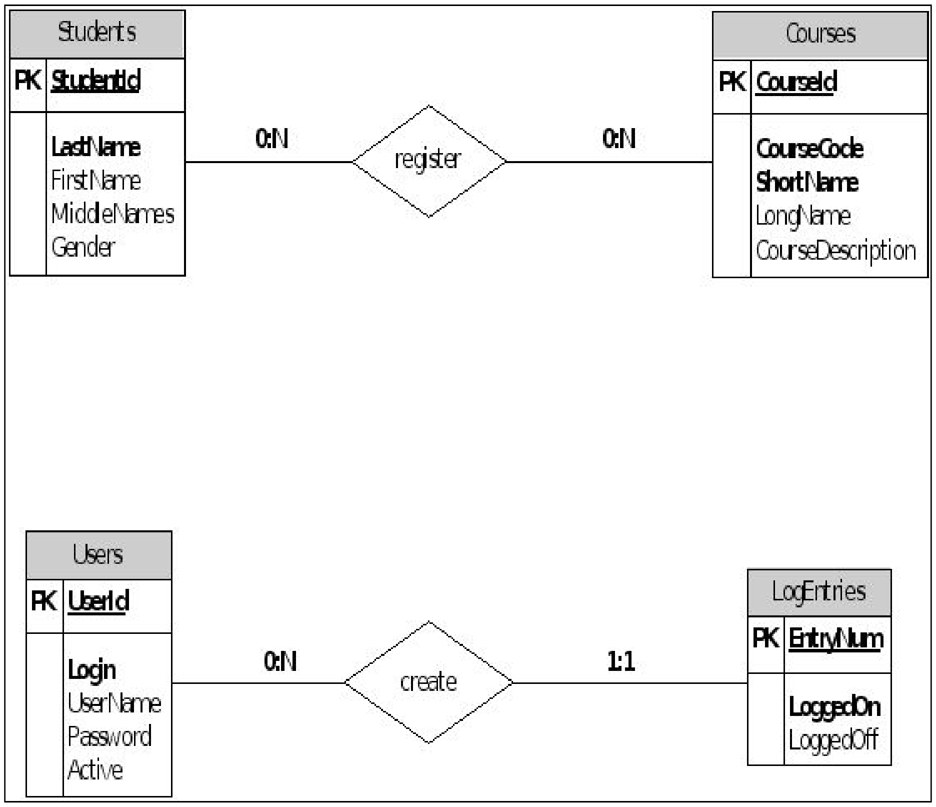

Figure 1.1 – Detailed E-R Diagram

The diagram is clear, concise, and complete, and represents the user's view of their data.

The notation used is an adaptation of the conventional ERM notation, but it nonetheless captures and documents all that needs to be captured and documented. This notation will be explained in subsequent chapters as the six steps of the relational database design process unfolds.

Implementation Model – R-M Diagram

The diagram below shows the result of the sixth and final step in the six-step relational database design process, the relational model diagram (R-M Diagram).

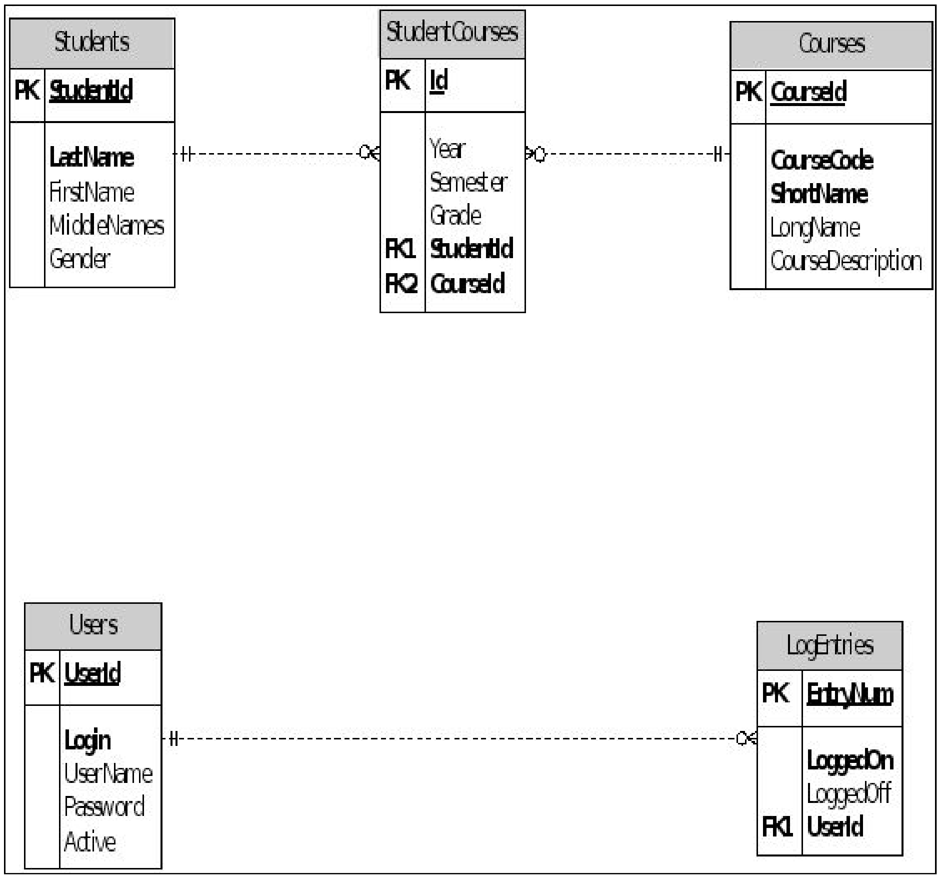

Figure 1.2 – R-M Diagram

The diagram is a Crow's Foot Relational Model diagram and was derived from the E-R diagram in figure 1.1. It shows all the tables necessary to implement a relational database that meets the requirements outlined in the problem specification on page 7.

Implementation – Relational Database

The R-M diagram in figure 1.2 can be implemented on any RDBMS. The SQL commands on the subsequent pages can be used to create the tables in a MySQL database. Similar SQL commands can be used to create the tables on any RDBMS.

```
CREATE TABLE Courses (
CourseId int(11) NOT NULL AUTO_INCREMENT,
CourseCode varchar(6) NOT NULL,
ShortName varchar(75) NOT NULL,
LongName varchar(150) DEFAULT NULL,
CourseDescription text,
PRIMARY KEY (CourseId),
INDEX CourseCode (CourseCode)
);
```

```
CREATE TABLE Students (
StudentId int(11) NOT NULL AUTO_INCREMENT,
LastName varchar(50) NOT NULL,
FirstName varchar(50) DEFAULT NULL,
MiddleNames varchar(50) DEFAULT NULL,
Gender varchar(6) NOT NULL DEFAULT 'MALE',
PRIMARY KEY (StudentId),
INDEX FullName (LastName,FirstName)
);

CREATE TABLE Users (
UserId int(11) NOT NULL AUTO_INCREMENT,
Login varchar(25) NOT NULL,
UserName varchar(50) DEFAULT NULL,
Password varchar(50) NOT NULL,
Active tinyint(4) NOT NULL DEFAULT '1',
PRIMARY KEY (UserId),
UNIQUE INDEX Login (Login)
);

CREATE TABLE LogEntries (
EntryNum bigint(20) NOT NULL AUTO_INCREMENT,
LoggedOn datetime NOT NULL,
LoggedOff datetime DEFAULT NULL,
UserId int(11) NOT NULL,
PRIMARY KEY (EntryNum),
FOREIGN KEY UserId (UserId)
    REFERENCES Users (UserId)
    ON UPDATE CASCADE ON DELETE RESTRICT
);

CREATE TABLE StudentsCourses (
Id int(11) NOT NULL AUTO_INCREMENT,
StudentId int(11) NOT NULL,
CourseId int(11) NOT NULL,
YearTaken year(4) NOT NULL,
SemesterTaken tinyint(3) unsigned NOT NULL DEFAULT '1',
Grade char(2) DEFAULT 'F',
PRIMARY KEY (Id),
FOREIGN KEY StudentId (StudentId)
    REFERENCES Students (StudentId)
    ON UPDATE CASCADE ON DELETE RESTRICT,
FOREIGN KEY CourseId (CourseId)
    REFERENCES Courses (CourseId)
    ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Chapters 2 to 7 detail the six steps that can be taken to reliably derive implementation models, such as the one shown on page 9, which can then be used to create sturdy, reliable, and efficient relational databases. Chapter 8 summarizes these steps.

Chapter 9 looks at how the derived designs (implementation models), such as the one shown on page 9, can be implemented on relational database management systems (RDMBSs). It looks at the steps necessary to implement each of the case studies on a relational database management systems (RDBMS).

Review Questions

1. Who were the two primary contributors (individuals) in the 1970s to what is today known as “relational databases”?

2. What is a relational database?

3. List five terms used in relational database design and development.

4. Using appropriate examples, describe the different types of database models created as part of the database design process.

5. Using an appropriate example, describe the four main stages of database development.

Exercises

1. Research the following on the internet:

   a. Structured Query Language

   b. MariaDB

   c. Object-oriented databases

2. Use the internet to list at least three open source relational databases that have been developed in the last five years, if any.

Chapter 2

The Six Steps – Step 1

Discover entities and assign attributes

Design is part art and part science, and the six steps outlined in this book help to make the relational database design process more of a science (technique) and less of an art.

This chapter details the first step of the six-step design process. It deals with understanding the problem by analyzing the processes and documentation used by the client in the problem domain. This chapter illustrates, using case studies, how to meticulously document your findings using entities and attributes.

At the end of this chapter, readers should be able to discover the entities in the problem domain, assign attributes to each entity discovered, and select identifiers and keys for each entity from the attributes of that entity.

Step 1: Discover entities and assign attributes

Great emphasis is placed on this stage of the process because it is the most important and because mistakes made during this stage will be propagated through all of the other stages and will be reflected in the actual database.

Three case studies, each increasingly complex, will illustrate the process of discovering the entities and assigning attributes to these entities.

Step 1-1: Discover the entities

As stated in Chapter 1, an entity is *“a tangible object of interest that exists in the user’s domain. It is something of interest to the user that the user keeps track of somewhere or somehow.”* Nouns or collective nouns in the problem domain denote objects of interest (entities). In order to discover the entities easily, a clear and concise statement of the problem is required.

This book will assume that you have obtained or created a clear and concise statement of the problem. This task may not be a simple one, and it may require some tedious work with the client. However, it is a very important part of the database-development process, and care should be taken to ensure that the statement of the problem is as clear and as concise as possible without omitting any important detail.

You will notice that in this book the first paragraph of all the statements of the problem for the case studies contains the phrase *“…wants to keep track of…”* or a version of this. This is because one of the prime purposes of a database is to *“keep track of”* data – to record and organize data.

Here is what to do to discover the entities:

First:

Identify all the collective nouns and nouns in the statement of the problem that represent *objects of interest* from the problem domain. These should **not** be descriptions or characteristics of the objects of interest, but rather, the nouns or collective nouns representing the objects of interest.

Second:

List the discovered objects of interest. The convention used in this book is to use plural nouns for objects of interest (entities). Using plural nouns makes it easier in the later stages of the database design process when relationships are derived (step 2).

Each object of interest becomes an entity that will be used as part of the database design.

Case Study 1

Here is a simple scenario:

A small accounting firm wants a simple HR application that will help it to keep track of its employees, their positions, allowances, salary scales, and which company vehicles their employees drive.

The application must keep track of all the positions at the firm, the employees filling these positions, the allowances for these positions, the salary scales for these positions, and the company vehicles assigned to these positions.

First:

Identify all the collective nouns and nouns in the statement of the problem that represent *objects of interest* from the problem domain:

A small accounting firm wants a simple HR application that will help it to keep track of its <u>employees</u>, their <u>positions</u>, <u>allowances</u>, <u>salary scales</u>, and which company <u>vehicles</u> their <u>employees</u> they drive.

The application must keep track of all the <u>positions</u> at the firm, the <u>employees</u> filling these positions, the <u>allowances</u> for these <u>positions</u>, the <u>salary scales</u> for these <u>positions</u>, and the company <u>vehicles</u> assigned to these <u>positions</u>.

Second:

List the discovered objects of interest *(entities)*:

- Employees
- Positions
- Allowances
- Salary Scales
- Vehicles

It may be argued that *Salary Scales* is a property or characteristic (attribute) of the *Position* and not a separate object of interest (entity). However, in most HR systems *Salary Scales* is a separate object of interest with its own properties or characteristics, which is tracked separately by the client. In this scenario we will assume that this is the case.

Remember: Double check to ensure that the entities listed capture all of the objects of interest in the problem domain.

Case Study 2

Here is a slightly more complicated scenario:

The owners of a small computer repair shop would like to keep track of the repair jobs for computers they repair, the items used for each repair job, the labor costs for each repair job, the repairmen performing each repair job, and the total cost of each repair job.

When customers bring their computers in to be repaired, they make a deposit on the repair job and are given a date to return and uplift their computer. Repairmen then perform repairs on the customers' computers based on the repair job, and detail the labor costs and the items used for each repair job.

When customers return they pay the total cost of the repair job less the deposit, collect a receipt for their payment, and uplift the repaired computer using this payment receipt.

First:

Identify all the collective nouns and nouns in the statement of the problem that represent *objects of interest* from the problem domain:

The owners of a small computer repair shop would like to keep track of the repair jobs for computers they repair, the items used for each repair job, the labor costs for each repair job, the repairmen performing each repair job, and the total cost of each repair job.

When customers bring their computers in to be repaired, they make a deposit on the repair job and are given a date to return and uplift their computer. Repairmen then perform repairs on the customers' computers based on the repair job, and detail the labor costs and the items used for each repair job.

When customers return they pay the total cost of the repair job less the deposit, collect a receipt for their payment, and uplift the repaired computer using this payment receipt.

Second:

List the discovered objects of interest *(entities)*:

- Repair Jobs
- Computers
- Items
- Repairmen
- Customers
- Deposits
- Payment Receipts (Payments)

It may be tempting to put *Labor Costs* as an entity, but it is an attribute and not an entity. It is an attribute of the *Repair Jobs* entity, because *Labor Costs* are for *Repair Jobs*. It may also be tempting to put *Total Cost* as an entity, but it is also an attribute and not an entity. Further, *Total Cost* is a calculated or derived attribute and not a regular attribute. Step

1-2 deals with attributes and how to find them and ensure that they are assigned to the correct entity.

A case can be made for making *Deposits* and *Payments* as part of the *Repair Jobs* entity if one deposit and one payment are made per repair job (no financing). However, *Deposits* and *Payments* represent the client's *financial transactions* and it is best to create separate entities for them. Chapter 9 explains the impact this has on the implementation of the design on RDBMSs

Remember: Double check to ensure that the entities listed capture all of the objects of interest in the problem domain.

Case Study 3

This is an even more complicated scenario:

The registrar at a small college wants an application that will help their department keep track of the scheduled classes, the courses and lecturers appearing in the schedule, and the students registering for courses according to the schedule.

Courses are scheduled every semester and this is documented in the schedule of classes, which also documents the lecturers assigned to each scheduled class. Students register for courses according to the list of scheduled classes.

Users (students, lecturers, and other college staff) must login to the application to gain access, and the application must keep track of user logins/logouts. In addition, users must have different levels of access, which will determine their access to different parts of the application.

First:

Identify all the collective nouns and nouns in the statement of the problem that represent *objects of interest* from the problem domain:

The registrar at a small college wants an application that will help their department keep track of the scheduled classes, the courses and lecturers appearing in the schedule, and the students registering for courses according to the schedule.

Courses are scheduled every semester and this is documented in the schedule of classes, which also documents the lecturers assigned to each scheduled class. Students register for courses according to the list of scheduled classes.

Users (students, lecturers, and other college staff) must login to the application to gain access, and the application must keep track of user logins/logouts. In addition, users must have different levels of access, which will determine their access to different parts of the application.

Second:

List the discovered objects of interest *(entities)*:

- Scheduled Classes (Schedule of classes or Schedule)
- Courses
- Lecturers
- Students
- Semesters
- Users
- User Logins/Logouts (Log Entries)
- Levels of Access (Access Levels)

It is important to note that the entity *Scheduled Classes* refers collectively to the different schedules that are made for *Courses* – each *Course* has a day and time that it is scheduled to run, which is referred to as a *Scheduled Class*. In subsequent steps, when attributes are assigned to this entity and relationships are built, its role will become clearer.

This highlights the importance of getting a clear and concise statement of the problem. It also highlights how important it is that the database designer understands the problem. It is advisable that as a database designer, you spend as much time with the client as possible clarifying details of how data is collected, documented, manipulated, and interpreted on a daily basis.

Remember: Double check to ensure that the entities listed capture all of the objects of interest in the problem domain.

Step 1-2: Assign attributes to each entity discovered

As stated in Chapter 1, an attribute is *“one of the various properties that describe the entity’s characteristics, and it describes one and only one characteristic of an entity.”* You need to find *all* of the attributes for *all* of the entities you discovered in the previous step.

Finding all of the attributes can be very tedious, and it requires you to work closely with the client to ensure that you get all of the relevant attributes for all of the entities. Gathering documents and forms from the problem domain usually helps because they contain information that the client captures. In addition, a meeting with the client and relevant stakeholders helps to ensure that all relevant information (attributes) is captured in the database design.

Ensure that everything the client records on paper is recorded in your database design and that everything in your database design is what the client wants recorded. Remember, you are designing the database for your client and not for yourself and therefore it must satisfy their needs.

Attributes should represent *one and only one* characteristic or property of an entity. They should *not* represent multiple characteristics or properties. It is a common mistake to include characteristics or properties such as *Full Name*, *Address*, and *Telephone Numbers* as attributes. The subsequent case studies show how these characteristics or properties should be treated.

There is heated debate as to whether calculated or derived attributes should be included in the database design. Database theory says that if a field is calculated or derived, there is no need to include it in in the design because it can be calculated or derived from existing fields. From my experience, it is best to include calculated fields in the database design and the database because server resources that are used to calculate the value of the field can be saved on subsequent queries requesting the calculated field. That is, the field is only calculated once, after which its value is pulled straight from the database.

Here is what to do to assign attributes to each entity discovered:

First:

For each entity, list the possible characteristics and/or properties that are recorded in the problem domain and are relevant to the client or end user. It is best to do this using a table with column headings for each entity discovered, with the cells of that column containing the characteristics and/or properties for that entity. A meeting with the client and relevant stakeholders is usually necessary to ensure that all attributes are captured.

The convention used in this book is to combine attribute names that are more than one word into one word, and capitalize the first letter of each word. For example, *First Name* becomes *FirstName* and *Last Name* becomes *LastName*.

Second:

Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it has been placed and not in any other entity or entities, and that it is not shared between or among entities.

For each entity, go through each attribute one at a time ensuring that it is where it belongs:

LastName is for *Lecturers*

FirstName is for *Lecturers*

…

DegreeProgram is for *Students*

…

LongName is for *Courses*

…

Password is for *Users*

…

LoggedOn is for *LogEntries*

If one of the attributes does not belong in the entity, or if you are not sure it belongs with that entity, or if it is shared between two entities, *remove it.* For example, it would be tempting to put *UserName* or *Login* in *LogEntries*, but they do not belong there because they are for *Users* not *LogEntries*. Eventually the primary key from *Users* would become part of the *LogEntries* table, but as a foreign key. This is explained in Chapter 7.

Let us look at each of the case studies that we have looked at before and assign attributes to each of the entities discovered.

Case Study 1

List of discovered entities:

- Employees
- Positions
- Allowances
- Salary Scales
- Vehicles

First:

For each entity, list the possible properties and/or characteristics recorded in the problem domain and those relevant to the client:

Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
SSNumber	PositionName	AllowanceName	SalaryScaleCode	VIN
LastName	PositionDescription	AllowanceDescription	SalaryScaleName	Year
FirstName	Details	Amount	SalaryScaleDescription	Make
MiddleName			MinimumSalary	Model
Gender			MaximumSalary	Color
DOB				RegistrationNo
Email				
Mobile				
HTel				
AddressLine1				
AddressLine2				
City				
State				
PostCode				

The above case study assumes that only *SalaryScales* have codes. It may be that *Positions* and *Allowances* also have codes. Once again, be sure that the attributes come from the problem domain, that is, the client.

Second:

Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it has been placed in and not in any other entity or entities, and that it is not shared between or among entities.

> **Remember:** Go through every attribute for every entity verifying that it is a valid attribute and that it is where it belongs.

Case Study 2

List of discovered entities:

- Repair Jobs
- Computers
- Items
- Repairmen
- Customers
- Invoices
- Payment Receipts

First:

For each entity, list the possible properties and/or characteristics recorded in the problem domain and those relevant to the client:

RepairJobs	**Computers**	**Items**	**Repairmen**	**Customers**
JobNum	SerialNum	PartNum	LastName	LastName
DateReceived	Make	ShortName	FirstName	FirstName
DateToReturn	Model	ItemDescription	MI	MI
DateReturned	ComputerDescription	Cost	Email	Email
DateStarted		NumInStock	Mobile	Mobile
DateEnded		ReorderLow	HTel	HTel
RepairDetails			Extension	AddressLine1
LaborDetails				AddressLine2
LaborCost				City
TotalCost				State
PaidInFull				PostCode
AdditionalComments				

Deposits	**Payments**		
DepositNum	PaymentNum		
DepositDate	PaymentDate		
Amount	Amount		

Second:

Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it has been placed in and not in any other entity or entities, and that it is not shared between or among entities.

> **Remember:** Go through every attribute for every entity verifying that it is a valid attribute and that it is where it belongs.

Case Study 3

List of discovered entities:

- Students
- Courses
- Scheduled Classes (Schedule of classes or Schedule)
- Semesters
- Lecturers

- Users
- Levels of Access (Access Levels)
- User Logins/Logouts (Log Entries)

First:

For each entity, list the possible properties and/or characteristics recorded in the problem domain and those relevant to the client:

Students	**Courses**	**ScheduledClasses**	**Semesters**
SSNumber	CourseCode	ScheduleCode	SemesterNumber
LastName	ShortName	Section	SemesterName
FirstName	LongName	Day	SemesterYear
MiddleName	CourseDescription	Time	StartDate
Gender		Location	EndDate
DOB			
Email			
Mobile			
HTel			
WTel			
AddressLine1			
AddressLine2			
City			
State			
PostCode			

Lecturers	**Users**	**AccessLevels**	**LogEntries**
SSNumber	Login	AccessLevelCode	LoggedOn
LastName	UserName	ShortName	LoggedOff
FirstName	Password	LongName	
MiddleName	Active	AccessLevelDescription	
Gender			
Email			
Mobile			
HTel			
WTel			
About			

Second:

Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it has been placed in and not in any other entity or entities, and that it is not shared between or among entities.

Remember: Go through every attribute for every entity verifying that it is a valid attribute and that it is where it belongs.

Step 1-3: Select identifiers, keys and primary keys from attributes of each entity

As stated in Chapter 1, an identifier is *"an attribute that can be used to identify entity occurrences (rows). They can be unique, non-unique, or composite. Unique identifiers identify one and only one entity occurrence (row), whereas non-unique identifiers can identify more than one entity occurrence (row). Composite identifiers consist of more than one attribute*." An index or key is *"a unique or non-unique identifier that can be used to speed up searches,"* and a primary key is *"a unique identifier that is used to distinguish or identify rows in a Relation (table)."*

Identifiers and keys are important for two reasons. First, every table *should* have a primary key; second, indexes or keys help speed up searches in relational database management systems (RDBMSs). Certain RDBMSs will not permit you to create a table without identifying a primary key for that table, so it is best to specify a primary key for every entity (table) in your database design at this stage.

At the end of the six-step process, each entity we have discovered will become a table, and each table should have a primary key, so selecting primary keys at this early stage gets this very important task out of the way.

Candidates for primary keys are attributes whose values will be different in *every* entity occurrence (row). Candidates for indexes or keys are attributes whose values will be different in *most* entity occurrences (rows) and that are likely to be used when searching the database.

After selecting the primary key for each entity, it is necessary to ensure that every other attribute in the entity depends on the primary key and the primary key only. In essence, does every attribute depend on *"the key, the whole key, and nothing but the key"* (Kent, 1983)? The reason for this has to do with normalization, which is dealt with in Chapter 9.

Here is what to do in order to obtain identifiers, keys and primary keys for entities:

First:

Go through each entity and list the possible identifiers and keys from the list of attributes.

Second:

Select the unique identifiers for each entity from the list of possible identifiers and keys.

Third:

From the list of unique identifiers, select one as the primary key. If there are no unique identifiers, then create one and call it *ID* or a derivative of *ID*, such as, *UserID* or *UserId*. In most instances it may be better to create a primary key and call it *ID*, despite the fact that the entity has a unique identifier. Some designers create a field called *ID* for every entity and make it the primary key. There is nothing wrong with this because it creates a unique identifier for every entity, which is what we are striving for. It is a design decision.

Fourth:

Ensure that every other attribute in the entity depends wholly and solely on the primary key – does every attribute depend on the primary key, the whole primary key, and nothing but the primary key? If the answer to this question is *no* for any attribute, then that attribute does not belong in that entity and you must remove it.

Case Study 1

The list overleaf shows the entities with primary keys (PK) and indexes or keys (bold) having completed the first, second, third, and fourth steps outlined above:

Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
(PK) EmployeeId	**(PK) PositionId**	**(PK) AllowanceId**	**(PK) SalaryScaleCode**	**(PK) VehicleId**
SSNumber	**PositionName**	**AllowanceName**	**SalaryScaleName**	**VIN**
LastName	PositionDescription	AllowanceDescription	SalaryScaleDescription	**RegistrationNo**
FirstName	Details	Amount	MinimumSalary	Year
MiddleName			MaximumSalary	Make
Gender				Model
DOB				Color
Email				
Mobile				
HTel				
AddressLine1				
AddressLine2				
City				
State				

PostCode

In the above case study, an *Id* field was created for all of the entities except the *SalaryScale* entity. It is assumed that *SalaryScaleCode* is a positive integer starting at 1, and that it contains no prefixes or suffixes.

As mentioned above, some designers prefer to create an *Id* field for each entity and make it the primary key. It is my preference to use this technique wherever possible, and to use text fields (*codes*) only when they will be short and simple, and not have very many *entity instances* (rows). Chapter 9 explains the ramifications of these types of decisions on the implementation of the design on RDBMSs.

Remember: Verify that every attribute depends on the primary key, the whole primary key, and nothing but the primary key.

Case Study 2

The list on the facing page shows the entities with primary keys (PK) and indexes or keys (bold) having completed the first, second, third, and fourth steps outlined above:

RepairJob	**Computers**	**Items**	**Repairmen**	**Customers**
(PK) JobNum	**(PK) ComputerId**	**(PK) ItemId**	**(PK) RepairmenId**	**(PK) CustomerId**
DateReceived	**SerialNum**	**PartNum**	**LastName**	**LastName**
DateToReturn	Make	**ShortName**	FirstName	FirstName
DateReturned	Model	ItemDescription	MI	MI
DateStarted	ComputerDescription	Cost	Email	Email
DateEnded		NumInStock	Mobile	Mobile
RepairDetails		ReorderLow	HTel	HTel
LaborDetails			Extension	AddressLine1
LaborCost				AddressLine2
TotalCost				City
PaidInFull				State
AdditionalComments				**PostCode**

Deposits	**Payments**			

(PK) DepositNum
DepositDate
Amount

(PK) PaymentNum
PaymentDate
Amount

In this case study the *JobNum*, *InvoiceNum*, and *PayReceiptNum* are sequential numbers, which makes them ideal candidates for primary keys. In some instances this may not be the case, and it is recommended that in transaction based systems the primary key always is a sequential number for entities representing objects such as, invoices, receipts, and transactions.

Remember: Verify that every attribute depends on the primary key, the whole primary key, and nothing but the primary key.

Case Study 3

The list overleaf shows the entities with primary keys (PK) and indexes or keys (bold) having completed the first, second, third, and fourth steps outlined above:

Students	**Courses**	**ScheduledClasses**	**Semesters**
(PK) StudentId	**(PK) CourseCode**	**(PK) ScheduleId**	**(PK) SemesterId**
SSNumber	**ShortName**	**ScheduleCode**	SemesterNumber
LastName	LongName	Section	SemesterName
FirstName	CourseDescription	Day	SemesterYear
MiddleName		Time	StartDate
Gender		Location	EndDate
DOB			
Email			
Mobile			
HTel			
WTel			
AddressLine1			
AddressLine2			
City			
State			
PostCode			

Lecturers	**Users**	**AccessLevels**	**LogEntries**
(PK) LecturerId	**(PK) UserId**	**(PK) AccessLevelId**	**(PK) LogEntryId**
SSNumber	**Login**	**AccessLevelCode**	LoggedOn
LastName	UserName	**ShortName**	LoggedOff
FirstName	Password	LongName	
MiddleName	Active	AccessLevelDescription	
Gender			
Email			
Mobile			
HTel			
WTel			
About			

An *AccessLevelId* and *AccessLevelCode* is included in the *AccessLevels* entity because at this stage we are not sure how long the *AccessLevelCode* will be, and it is always better to have an *Id* as the primary key than a code. *CourseCode* is maintained as the primary key for *Courses* even though it a code and not a number. Chapter 9 explains the ramifications of these types of decisions on the implementation of the design on RDBMSs.

Remember: Verify that every attribute depends on the primary key, the whole primary key, and nothing but the primary key.

Summary

This chapter focused on how to discover the entities and assign attributes to these entities. Here is a list of steps to discovering entities and assigning attributes:

Step 1-1: Discover the entities

First: Identify all the collective nouns and nouns in the statement of the problem that represent *objects of interest* from the problem domain. These should **not** be descriptions or characteristics of objects of interest.

Second: List the discovered objects of interest using plural nouns for objects of interest (entities).

> **Remember:**Double check to ensure that the entities listed capture all of the objects of interest in the problem domain.

Step 1-2: Assign attributes to each entity discovered

First: For each entity list the possible properties and/or characteristics recorded in the problem domain and relevant to the client.

Second: Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it has been placed in and not in any other entity or entities, and that it is not shared between or among entities.

> **Remember:** Go through every attribute for every entity verifying that it is where it belongs.

Step 1-3: Select identifiers and keys from attributes of each entity

First: Go through each attribute in each entity and list the possible identifiers and keys.

Second: Select the unique identifiers for each entity from the list of possible identifiers and keys.

Third: Out of the list of unique identifiers, select one as the primary key. If there are no unique identifiers, then create one and call it *ID* or a derivative of *ID* such as, *UserID* or *UserId*.

Fourth: Ensure that every other attribute in the entity depends wholly and solely on the primary key.

Remember: Verify that every attribute depends on the primary key, the whole primary key, and nothing but the primary key.

Review Questions

1. What is an entity?
2. What is an attribute?
3. What is an identifier?
4. What is a primary key?
5. Why are identifiers important?
6. Why are primary keys important?
7. What is the difference between unique and non-unique identifiers
8. What is the difference between a key and a primary key?
9. Briefly describe the steps involved in discovering the entities from a problem statement.
10. Using an appropriate example, describe how you would select the keys and primary keys for entities discovered.
11. With regards to discovering entities and assigning attributes to these entities, explain why it is important to work closely with the client.
12. Using an appropriate example, explain what can happen if the database designer neglects to consult with the client during this stage of database design.

Exercises

Use the problem specification below to answer the questions that follow.

> The librarian of a small town's library wants a database for the library that will help it to keep track of its collections (e.g. fiction, non-fiction, and journals), the books or items in those collections, the physical location in the library of these collections, the members of the library and the books they borrow from the various collections.
>
> The database must keep track of which book or item belongs to which collection, which collection is located where in the library, and which members borrowed books or items from which collections.

1. List the entities for the above scenario.

2. Assign attributes to each of the entities listed in part 1 above. State any assumptions you make regarding the client's needs.

3. For each of the entities listed in part 1 above, identify the primary key and other keys. State any assumptions you make regarding the client's needs.

Chapter 3

The Six Steps – Step 2

Derive unary and binary relationships

The second of the six steps deals with relationships and deriving all of the unary and binary relationships that exist between all of the entities. This chapter describes and uses the Entity-Entity Matrix to derive all of the unary and binary relationships that exist between all of the entities. The three case studies from the previous chapter are used to explain how to use the Entity-Entity Matrix to derive the relationships between the entities discovered.

At the end of this chapter, readers should be able to find all of the relationships that exist between any pair of entities in the database.

Step 2: Derive unary and binary relationships

In Chapter 1, a relationship is defined as something that *exists between entities*. An Entity-Entity Matrix (E-E Matrix) can be used to easily discover all of the relationships that exist between any two entities discovered in Step 1 of the six-step process. These relationships are either unary or binary relationships.

A unary relationship occurs when there are two entities involved in a relationship and they are the same entity. For example, *Courses* can be related to itself via the relationship ‘*prerequisite*’: *Courses* are *prerequisites* for *Courses*. A binary relationship occurs when there are two entities involved in a relationship and the entities are different. For example, *Lecturers lecture Courses*.

It must be noted that database theory allows for higher order relationships—between three or more entities. However, this book ignores these types of relationships because unary and binary relationships are usually sufficient to answer any queries that will be made on the data in the database.

The same three case studies used in Chapter 2 will be used in this chapter to illustrate how to use the E-E Matrix to derive all of the unary and binary relationships that exist between entities discovered during Step 1 of the six-step process.

Step 2-1: Build the Matrix

The E-E Matrix is built using entities discovered in Step 1 of the six-step process. It is a table consisting of an equal number of rows and columns, with each entity discovered heading a row and a column. The intersection of the rows and columns represents the relationships that may exist between the entities.

While not essential, it is strongly recommended that the order of the entities on the row and column headings is the same, which makes filling in the relationships easier in Step 2-2 described later in this chapter.

Case Study 1

List of discovered entities:

- Employees
- Positions
- Allowances
- SalaryScales
- Vehicles

Construct the Matrix using the entities:

	Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
Employees					
Positions					
Allowances					
SalaryScales					
Vehicles					

Remember: Verify that every entity discovered is listed on the heading row and heading column and that the order of the entities is the same.

Case Study 2

List of discovered entities:

- Repair Jobs
- Computers
- Items
- Repairmen

- Customers
- Deposits
- Payment Receipts (Payments)

Construct the Matrix using the entities:

	RepairJobs	**Computers**	**Items**	**Repairmen**	**Customers**	**Deposits**	**Payments**
RepairJobs							
Computers							
Items							
Repairmen							
Customers							
Deposits							
Payments							

Remember: Verify that every entity discovered is listed on the heading row and heading column and that the order of the entities is the same.

Case Study 3

List of discovered entities:

- Students
- Courses
- Scheduled Classes
- Semesters
- Lecturers
- Users
- AccessLevels
- LogEntries

Construct the Matrix using the entities:

	Students	Courses	Scheduled Classes	Semesters	Lecturers	Users	Access Levels	Log Entries
Students								
Courses								
Scheduled Classes								
Semesters								
Lecturers								
Users								
Access Levels								
LogEntries								

Remember: Verify that every entity discovered is listed on the heading row and heading column and that the order of the entities is the same.

Step 2-2: Fill in the Matrix

Each cell in the E-E Matrix represents a potential relationship that can exist between the entity on the row heading and the entity on the column heading. There is the possibility that no relationship exists between the two entities; that there is one relationship between the two entities; and that there is more than one relationship between the entities.

There is also the possibility that relationships can exist when the entity on the heading row is the same as the entity that is on the heading column. Relationships in cells where this is the case are called *unary* relationships.

In order to determine what relationships exist, it is necessary to revisit the problem domain and thoroughly question the client and end users about relationships and potential relationships. You may need to go through the Matrix cell by cell, questioning the client and end users about each potential relationship.

Here is what to do to fill in the Matrix:

First:

Go through each cell in the Matrix, asking the question *'is [Entity in Row Heading] related to [Entity in Column Heading]?'* For example, *'is Lecturers related to Courses?'* If a relationship [or relationships] exists, place a verb in the cell for each relationship. Each verb that is placed in the cell represents a relationship that exists between the entity on the row heading and the entity on the column heading.

It is possible that the same verb can be used to represent different relationships but in different contexts depending on the entities that are in the relationship, thus creating different relationships with the same verb.

You will notice that the relationships are duplicated for half of the Matrix, the difference being that the relationships are in reverse. This reversal does not change the fact that a relationship exists, nor does it change the relationship – it is the same relationship, only in reverse.

Second:

Ignore the top half of the Matrix drawn down the diagonal from the top left of the Matrix to the bottom right of the Matrix, since it is a mirror image of the bottom half.

You will need to examine each relationship closely to ensure that it is a relationship that you want to capture in your database, or to ensure that it is a relationship that is not captured through another pair or other pairs of relationships.

Remember to verify that each relationship captured in the E-E Matrix is a valid relationship to the client. You may need to go through the Matrix cell by cell with the client, verifying that each relationship is valid and relevant.

Case Study 1

First:

Go through each cell in the Matrix asking the question, *'is [Entity in Row Heading] related to [Entity in Column Heading]?'*

The following E-E Matrix is the result for case study 1:

	Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
Employees		fill			
Positions	fill		allocated	attached	allotted
Allowances		allocated			
SalaryScales		attached			
Vehicles		allotted			

Second:

Ignore the top half of the Matrix drawn down the diagonal from the top left of the Matrix to the bottom right of the Matrix since it is a mirror image of the bottom half.

This will result in a Matrix that looks like the following:

	Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
Employees		▨	▨	▨	▨
Positions	fill		▨	▨	▨
Allowances		allocated		▨	▨
SalaryScales		attached			▨
Vehicles		allotted			

There are no relationships between *Employee* and *SalaryScale* and *Employee* and *Allowances.* None is necessary because *SalaryScale* and *Allowances* are related to the *Position not* the *Employee*. By virtue of the relationship between *Employee* and *Position*, there will be an indirect relationship between *Employee* and *SalaryScale* and *Employee* and *Allowances*.

These are nuances that can only be verified by examining the Matrix closely and comparing it with the requirements of the database. In addition, you need to work closely with the client to ensure that all of the relevant relationships are captured and that the captured relationships are correct.

Remember: Verify with the client that each relationship represented in the E-E Matrix is valid. Do not create relationships that are neither required nor captured in the problem domain.

Case Study 2

First:

Go through each cell in the Matrix asking the question, *'is [Entity in Row Heading] related to [Entity in Column Heading]?'*

The following E-E Matrix is the result for case study 2:

	RepairJobs	**Computers**	**Items**	**Repairmen**	**Customers**	**Deposits**	**Payments**
RepairJobs		conducted	use	perform	request	make	make
Computers	conducted				own		
Items	use			order			
Repairmen	perform		order				
Customers	request	own					
Deposits	make						
Payments	make						

Second:

Ignore the top half of the Matrix drawn down the diagonal from the top left of the Matrix to the bottom right of the Matrix since it is a mirror image of the bottom half.

This will result in a Matrix that looks like the following:

	RepairJobs	**Computers**	**Items**	**Repairmen**	**Customers**	**Deposits**	**Payments**
RepairJobs							
Computers	conducted						
Items	use						
Repairmen	perform		order				

Customers	request	own					
Deposits	make						
Payments	make						

Notice that *Deposits* and *Payments* are only related to *RepairJobs*, and this is because only this relationship is necessary. There will be an indirect relationship between *Deposits* and *Customers* via *RepairJobs*.

The importance of eliminating the redundant relationships cannot be over-emphasised. In many instances you may have to re-do the matrix several times, collaborating with the client each time, confirming and eliminating relationships. Remember that you are designing the database for your client and not for yourself, so you want to capture the relationships as the client sees them and not as you see them.

Remember: Verify with the client that each relationship represented in the E-E Matrix is valid. Do not create relationships that are neither required nor captured in the problem domain.

Case Study 3

First:

Go through each cell in the Matrix asking the question, *'is [Entity in Row Heading] related to [Entity in Column Heading]?'*

The following E-E Matrix is the result for case study 3:

	Students	**Courses**	**Scheduled Classes**	**Semesters**	**Lecturers**	**Users**	**Access Levels**	**Log Entries**
Students			register			assigned		
Courses		pre-requisite	scheduled		lecture			
Scheduled Classes	register	scheduled		created for	lecture			
Semesters			created for					
Lecturers		lecture	lecture			assigned		
Users	assigned				assigned	create	designated	create
Access Levels						designated		
Log Entries						create		

Second:

Ignore the top half of the Matrix drawn down the diagonal from the top left of the Matrix to the bottom right of the Matrix since it is a mirror image of the bottom half.

This will result in a Matrix that looks like the following:

	Students	**Courses**	**Scheduled Classes**	**Semesters**	**Lecturers**	**Users**	**Access Levels**	**Log Entries**
Students								
Courses		pre-requisite						
Scheduled Classes	register	scheduled						
Semesters			created for					
Lecturers		lecture	lecture					
Users	assigned				assigned	create		
Access Levels						designated		
Log Entries						create		

The Matrix above may look odd; however, a close look at the problem specification would answer some questions. First, you may be wondering why *Students* do not *register* for *Courses*. This is because *Courses* are *scheduled* every semester in the *Schedule*, and *Students* register according to *ScheduledClasses*.

Second, you may be wondering why *Lecturers* lecture *Courses* **and** *Lecturers* lecture *ScheduledClasses*. This is because *Lectures* do lecture *Courses*, but they also lecture as stated in *ScheduledClasses*.

Once again it is apparent how important it is to collaborate with the client and the end user to correctly identify all of the relationships. Remember that you are designing the database for your client and not for yourself, so it is the client’s view of the data that should be captured.

Remember: Verify with the client that each relationship represented in the E-E Matrix is valid. Do not create relationships that are neither required nor captured in the problem domain.

Summary

This chapter focused on how to discover the entities and assign attributes to these entities. Here is a list of steps to discovering entities and assigning attributes:

Step 2-1: Build the Matrix

The E-E Matrix is built using entities discovered in Step 1 of the six-step process. It is a table consisting of an equal number of rows and columns, with each entity discovered heading a row and a column. The intersection of the rows and columns represents relationships that may exist between the entities.

> **Remember:** Verify that every entity discovered is listed on the heading row and heading column and that the order of the entities is the same.

Step 2-2: Fill in the Matrix

First: Go through each cell in the Matrix asking the question, *'is [Entity in Row Heading] related to [Entity in Column Heading]?'* If a relationship exists, place a verb in the cell for each relationship.

Second: Ignore the top half of the Matrix drawn down the diagonal from the top left of the Matrix to the bottom right of the Matrix since it is a mirror image of the other half.

> **Remember:** Verify with the client that each relationship represented in the E-E Matrix is valid. Do not create relationships that are neither required nor captured in the problem domain.

Review Questions

1. What is an Entity-Entity Matrix?

2. What is a relationship?

3. List two types of relationships that can exist between entities.

4. Using appropriate examples, explain the difference between the two types of relationships that can exist between entities.

5. Using an appropriate example, explain how the E-E Matrix can be used to derive all of the relationships that may exist between entities.

6. With regards to finding relationships that can exist between entities, explain why it is important to work closely with the client.

7. Using an appropriate example, explain what can happen if the database designer neglects to consult with the client during this stage of database design.

Exercises

Use the problem specification below to answer the questions that follow. It is similar to the scenario from the previous chapter, but with a little twist.

> The librarian of a small town's library wants a database for the library that will help it to keep track of its collections (e.g. fiction, non-fiction, and journals), the books or items in those collections, the physical location in the library of these collections, the members of the library and the books they borrow from the various collections.
>
> The database must keep track of which book or item belongs to which collection, which collection is located where in the library, and which members borrowed books or items from which collections.
>
> Library members and library staff (users) must be able to login to the database to use it to locate books, and the database must keep track of user logins/logouts. In addition, users must have different levels of access, which will determine their access to different parts of the database.

1. List the entities for the above scenario.
2. Assign attributes to each of the entities listed in part 1 above. State any assumptions you make regarding the client's needs.
3. For each of the entities listed in part 1 above, identify the primary key and other keys. State any assumptions you make regarding the client's needs.
4. Use the entities discovered in part 1 above to derive all of the unary and binary relationships between all of the pairs of entities. State any assumptions you make regarding the client's needs.

Chapter 4

The Six Steps – Step 3

Create simplified Entity-Relationship diagram

This chapter uses the information obtained from Step 1 and Step 2 of the six-step process to create a simplified Entity-Relationship (E-R) diagram. This simplified diagram only contains information about the entities and the relationships that exist between the entities. It does not contain information about the Optionality and Cardinality of the relationships.

At the end of this chapter, readers should be able to use the information gathered in the previous two steps to construct a simplified E-R diagram using the notation presented in this book.

Step 3: Create simplified Entity-Relationship diagram

In this book, a simplified Entity-Relationship (E-R) diagram refers to an E-R diagram that does not contain information about Optionality and Cardinality. Deriving the Optionalities and Cardinalities for each relationship is covered in the next step, Step 4. This is then used to develop a detailed E-R diagram in Step 5 of the six-step process.

The notation used in this book follows most conventions, in which a rectangle is used to represent an entity, and a diamond is used to represent a relationship. The name of the entity is printed inside the rectangle at the top, with the primary key (PK), indexes and other important attributes printed below it. The diamond contains the name of the relationship. Here are examples:

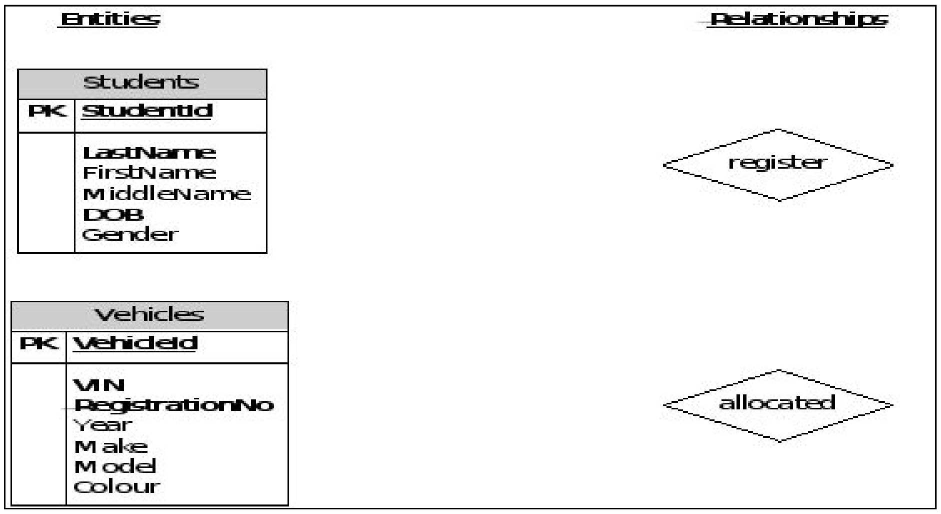

Figure 4.1 – Notation for entities and relationships

You do not need to include all of the attributes for all of the entities on the simplified E-R diagram because doing so can clutter the diagram by making the entities unusually long. Therefore, in the interest of space, only the primary key and the most important and relevant attributes should be included in the entities on the simplified E-R diagram. Candidates for indexes or keys are represented in bold.

Step 3-1: Create simplified E-R diagram

Each of the entities discovered in Step 1 of the six-step process is represented by a rectangle, clearly indicating the primary key and other important attributes. Each of the relationships derived in Step 2 of the six-step process is represented by a diamond with the name of the relationship in the diamond.

The entities (rectangles) are then connected to the relationships (diamonds) according to the E-E Matrix derived in Step 2 of the six-step process. If there are many entities and many relationships, the diagram may span multiple pages.

Here is what to do to create a simplified E-R diagram:

First:

Create the entities (rectangles), clearly indicating the primary key and other important attributes.

Second:

Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix.

You many need to re-organize the diagram so that it is not confusing. Entities that have relationships with themselves can be represented on the diagram more than once, or by simply having the diamond clearly feed to and from the same entity. Case Study 3 has examples of this, with the *Users* and *Courses* entities.

Case Study 1

First:

Create the entities (rectangles), clearly indicating the primary key and other important attributes.

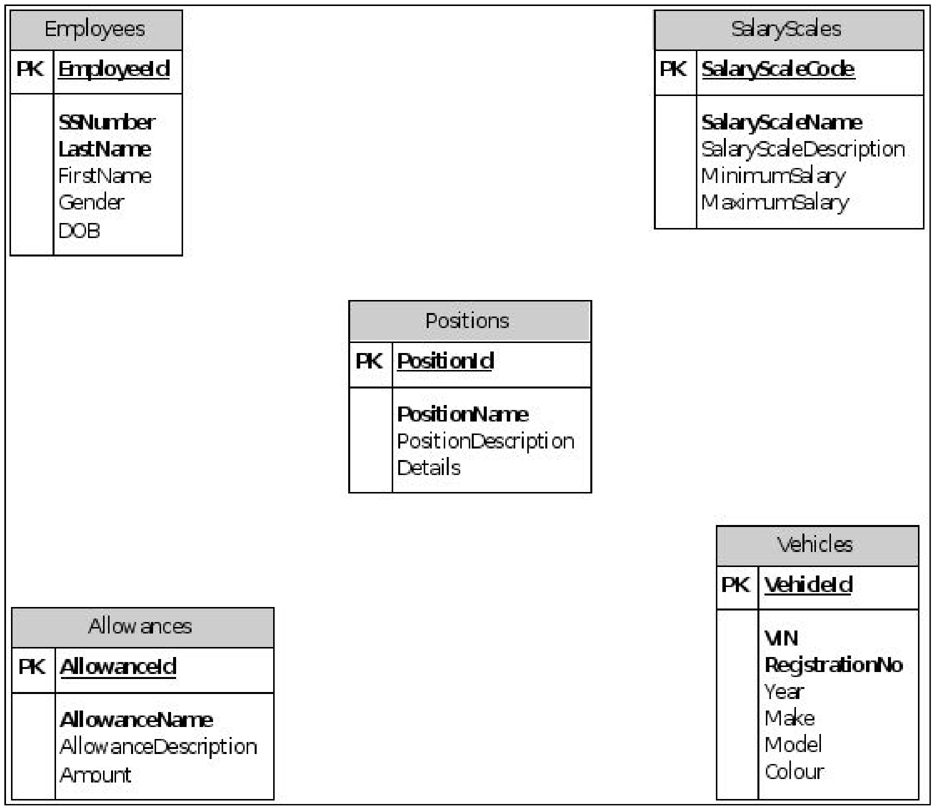

Figure 4.2 – Entities for Case Study 1

Second:

Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix. You many need to re-organize the diagram so that it is not confusing.

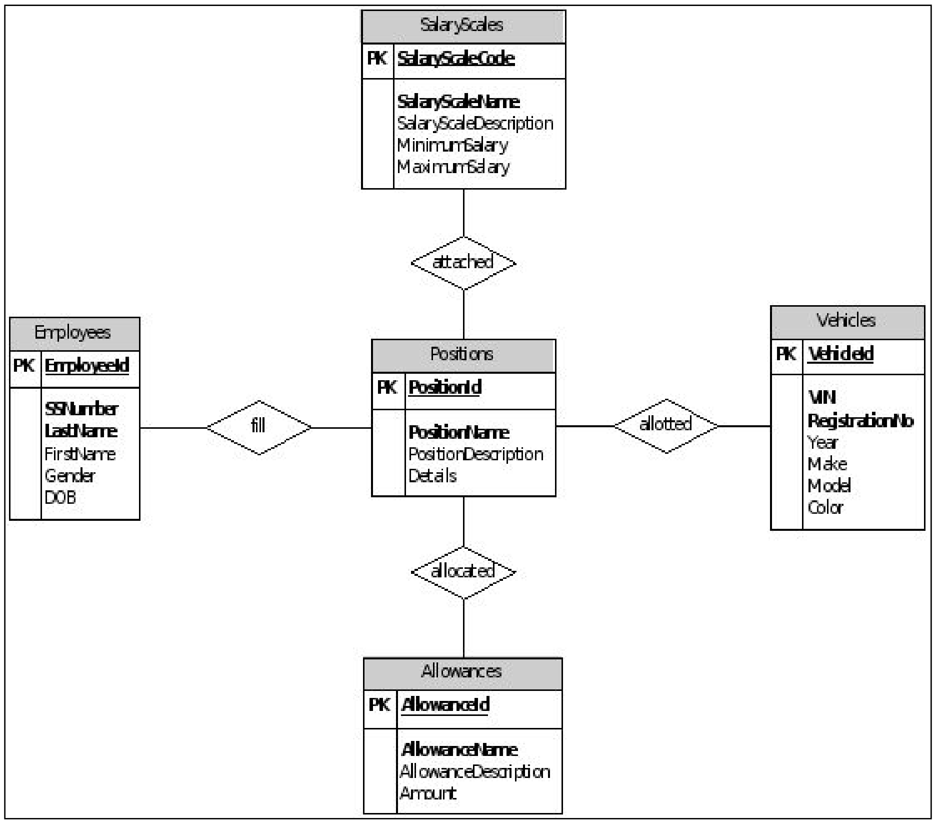

Figure 4.3 – Simplified E-R diagram for Case Study 1

Remember: Ensure that every entity has a primary key and that the important attributes are depicted on that entity. Also, ensure that every relationship has a name, and that the name is correct, and that it is associated with the correct entities.

Case Study 2

First:

Create the entities (rectangles), clearly indicating the primary key and other important attributes.

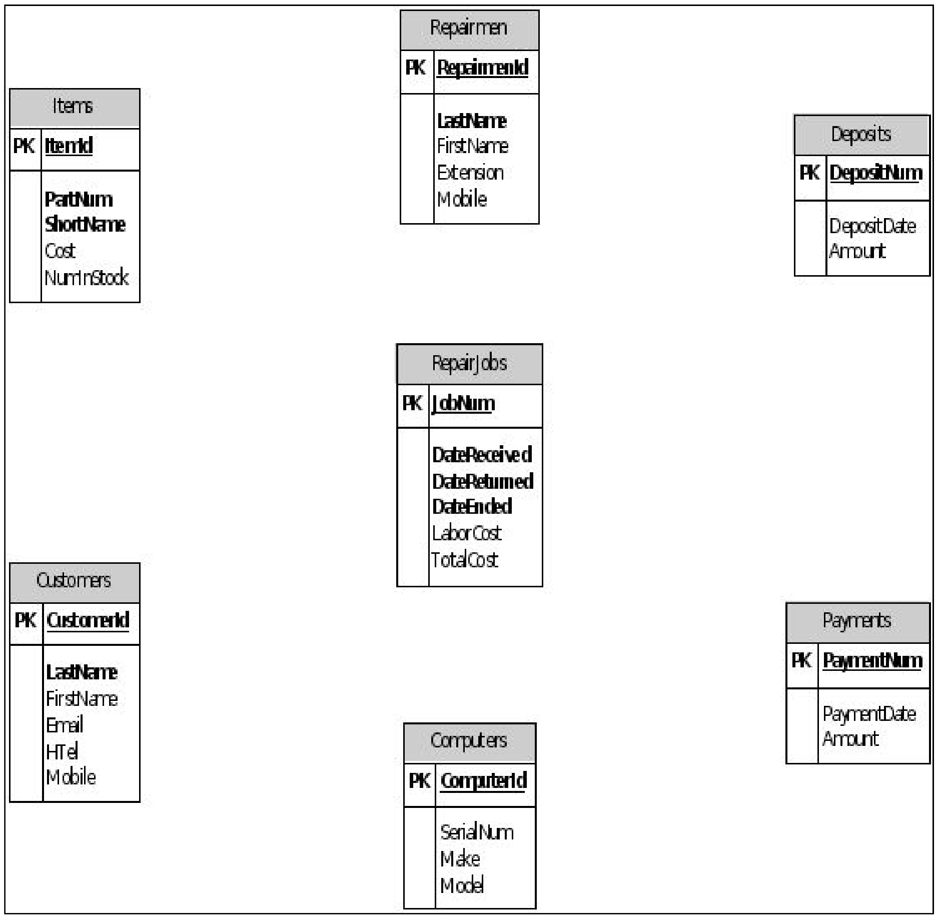

Figure 4.4 – Entities for Case Study 2

Second:

Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix. You many need to re-organize the diagram so that it is not confusing.

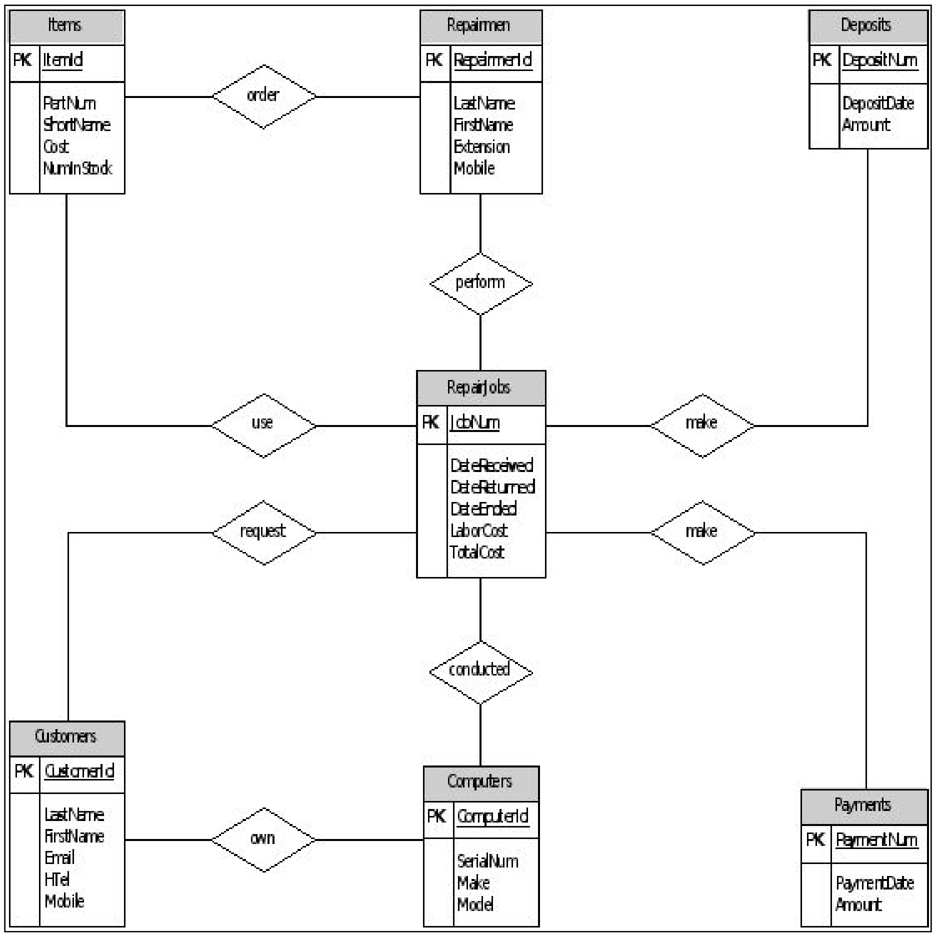

Figure 4.5 – Simplified E-R diagram for Case Study 2

Remember: Ensure that every entity has a primary key and that the important attributes are depicted on that entity. Also, ensure that every relationship has a name, and that the name is correct, and that it is associated with the correct entities.

Case Study 3

First:

Create the entities (rectangles), clearly indicating the primary key and other important attributes.

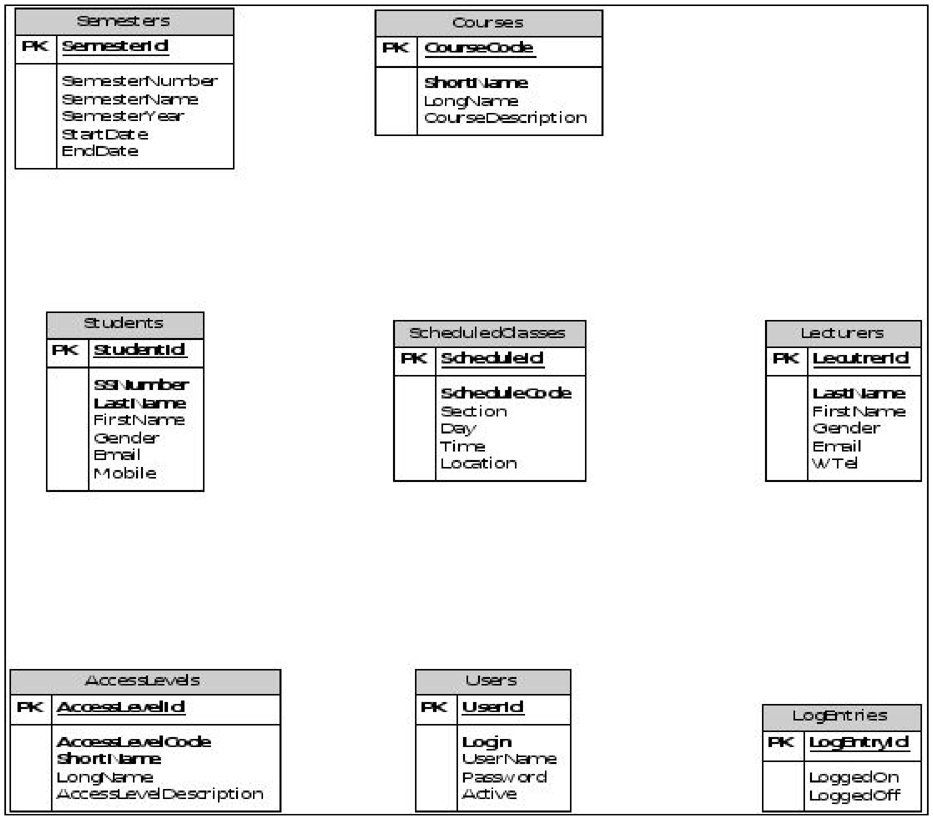

Figure 4.6 – Entities for Case Study 3

Second:

Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix. You many need to re-organize the diagram so that it is not confusing.

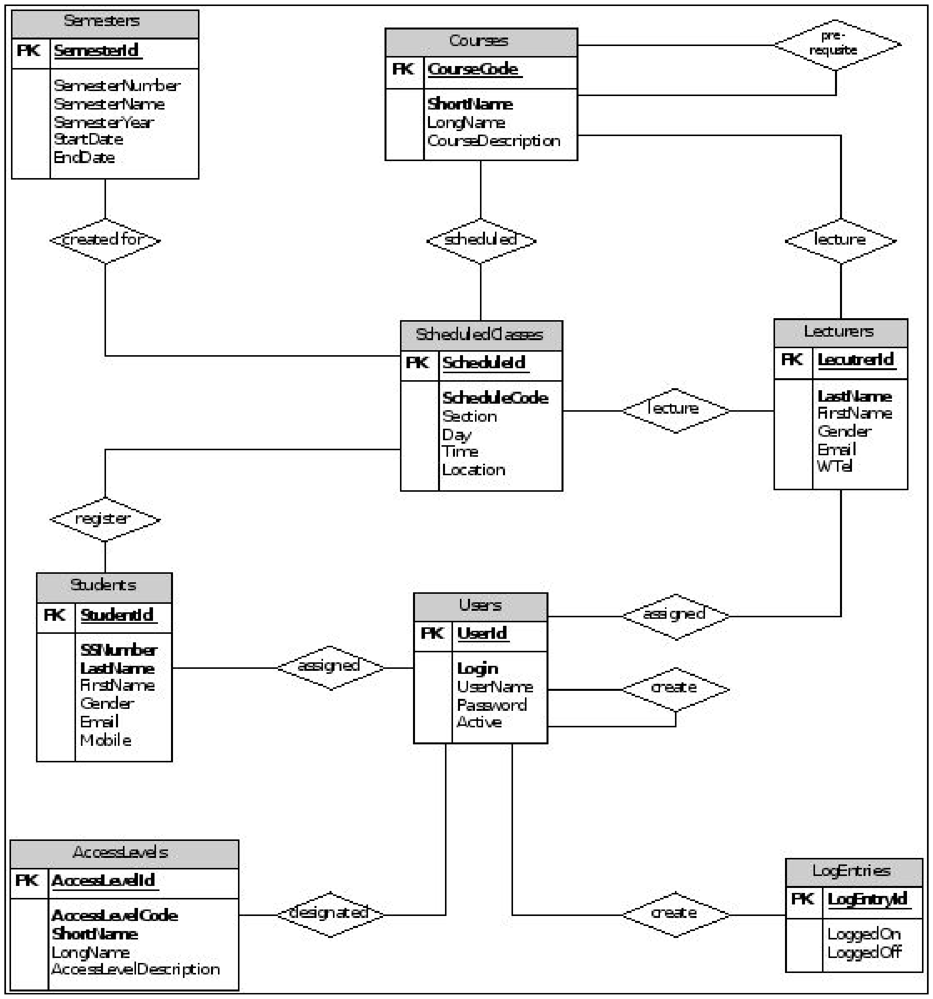

Figure 4.7 – Simplified E-R diagram for Case Study 3

Remember: Ensure that every entity has a primary key and that the important attributes are depicted on that entity. Also, ensure that every relationship has a name, and that the name is correct, and that it is associated with the correct entities.

Summary

This chapter focused on how to create a preliminary E-R diagram from the information derived in Step 1 and Step 2 of the six-step process. Here is a list of steps to create the simplified E-R diagram:

Step 3-1: Create simplified E-R diagram

Each of the entities derived in Step 1 the six-step process is represented by a rectangle, clearly indicating the primary key and important attributes. Each of the relationships derived in Step 2 of the six-step process is represented by a diamond with the name of the relationship in the diamond.

First: Create the entities (rectangles), clearly indicating the primary key and other important attributes.

Second: Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix.

Remember: Ensure that every entity has a primary key and that the important attributes are depicted on that entity. Also, ensure that every relationship has a name, and that the name is correct, and that it is associated with the correct entities.

Review Questions

1. What is a simplified Entity-Relationship (E-R) diagram?

2. What notation is used in this book to depict E-R diagrams?

3. Using an appropriate example, describe the process of creating a simplified E-R diagram from the E-E Matrix and list of entities and their attributes.

Exercises

1. Use the E-E Matrix below to construct a simplified E-R diagram as demonstrated in this book. You can make up the attributes for each entity.

	Cars	**Mechanics**	**WorkOrders**	**CarParts**	**Customers**
Cars					
Mechanics					
WorkOrders	create	complete			
CarParts		requisition	utilize		
Customers	own		fill		

Chapter 5

The Six Steps – Step 4

List assertions for all relationships

This chapter uses the simplified E-R diagram created in the previous step, Step 3, to create a list of assertions for the database. These assertions give definitive information about the entities in the database and their relationship with each other.

At the end of this chapter, readers should be able to use the simplified E-R diagram created in Step 3 of the six-step process to create a list of assertions for the database that is being designed.

Step 4: List assertions for all relationships

The *assertion* is based on the *predicate* in mathematics, which can be seen as a true/false statement, and when used in this book, it is a predicate that is always true. Therefore, within the context of this book, an *assertion* refers to a true and factual statement about two [and only two] entities in the database that we are trying to model and the relationship between these two entities.

Assertions are necessary to iron out the details about the relationships between the entities in the database. The relationships between the entities must satisfy the assertions made about them. Further, the client must verify that every assertion made is true and correct within the context of the database that is being modeled because the database is going to be modeled assuming that the assertions made at this stage are true and correct.

In order to obtain all of the assertions for the database that is being modeled, each relationship must be looked at individually. In addition, each relationship must be looked at from two directions: from *Entity A* to *Entity B* and then from *Entity B* to *Entity A*. At the end of this process, each relationship yields two assertions.

Optionality, Cardinality, Entity Class, and Entity Occurrences

Optionality, cardinality, entity class, and entity occurrences help to record the assertions that are made about the entities and their relationship to each other. They tell us what **can** and **must** happen in a relationship and the number of occurrences an entity may have in a relationship.

The *entity class* refers to the collective occurrences of all instances of an entity, or the entity itself, for example, *Employees*. An *entity occurrence* is one actual instance of the particular entity and entity occurrences are several instances of a specific entity. For example, the entity *Employees* can have *John Jones* or *Jack Smith* as entity occurrences.

Optionality says what **can** and **must** happen in a relationship. It can take one of two values, 0 or 1:

Optionality ‘0’ – Can (optional)

Optionality ‘1’ – Must (obligatory)

A hidden nuance about having an optionality of ‘0’ (can) is that it infers the possibility that something **cannot** happen in a relationship. Be mindful of this and always verify with the client what can and what cannot happen in a relationship.

Cardinality indicates the **number** of entity occurrences in a relationship, which can be any number from zero to infinity. However, in this book, cardinality will have one of two values, 1 and N:

Cardinality ‘1’ – Only one

Cardinality ‘N’ – Many or At least one

Step 4-1: List assertions using the simplified E-R diagram

Each relationship on the simplified E-R diagram will yield two assertions because the relationship is looked at from two directions: from *Entity A* to *Entity B*, then from *Entity B* to *Entity A*. Therefore, there will be twice as many assertions as there are relationships.

Here are the steps for listing the assertions using the simplified E-R diagram:

First:

Look at each relationship from *Entity A* to *Entity B* and write out the relationship in words, using the entities involved in the relationship, the optionalities, and cardinalities. Here is an example:

A Customer <u>can</u> make <u>many</u> Payments

The word **a** (**A**) is used to specify that what is being referred to is the entity occurrence and not the entity class.

The assertion always starts with an entity occurrence and always ends with an entity occurrence or an entity class, and the relationship is the verb in the middle of the two. The optionality comes after the first entity occurrence, the cardinality comes before the second entity occurrence or entity class, and the relationship separates the two entities. This is how assertions should be written:

*Entity Occurrence **optionality** relationship **cardinality** Entity Occurrence or Entity Class*

Second:

Look at each relationship in reverse, from *Entity B* to *Entity A*, and write out the relationship in words, using the entities involved in the relationship, the optionalities, and the cardinalities. Here is the same example in reverse:

Each Payment <u>must</u> be made by <u>only one</u> Customer

The word *each* is used to specify that what is being referred to is the entity occurrence and not the entity class.

The client must verify that every assertion made is true and correct within the context of the database that is being modeled.

Case Study 1

First:

- A Position can be allocated many Allowances
- A Position can be allotted only one Vehicle
- A Position must be attached to only one SalaryScale
- A Position can be filled by many Employees

Second:

- Each Allowance can be allocated to many Positions
- Each Vehicle must be allotted to only one Position
- Each SalaryScale can be attached to many Positions
- Each Employee can fill many Positions

For this scenario it is assumed that the client wishes to track the employee if he or she moves from one position to another within the company over time, hence the assertions, *each employee can fill many positions* (over time) and *a position can be filled by many employees* (over time).

Here is the simplified E-R diagram for the first case study:

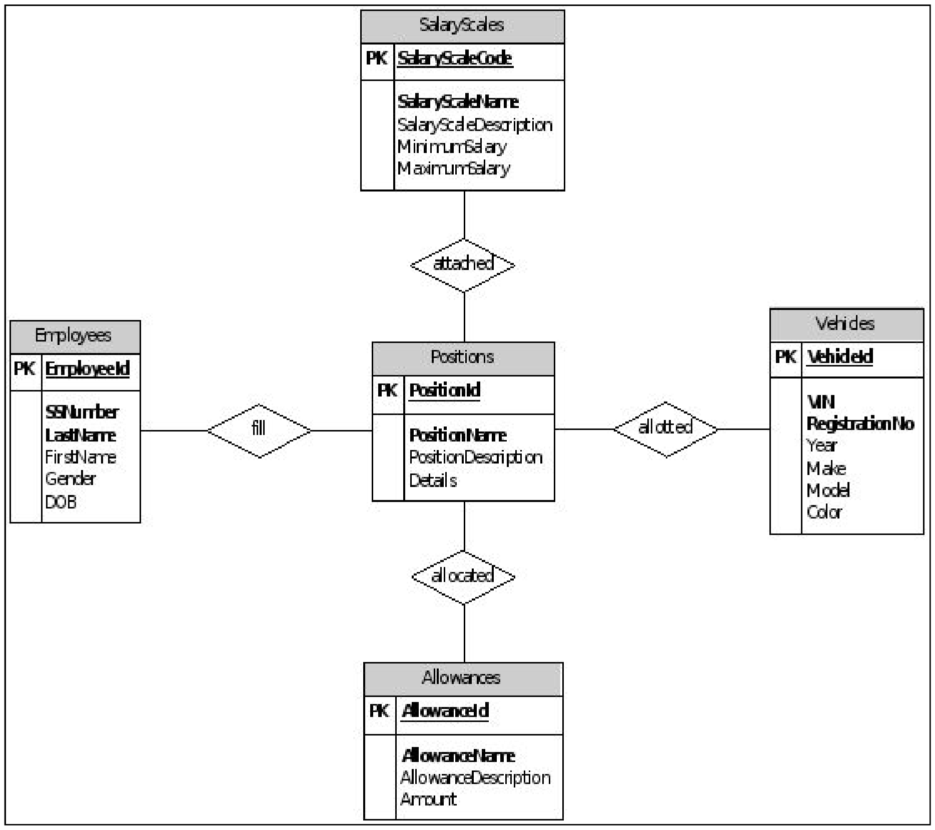

Figure 5.1 – Simplified E-R diagram for Case Study 1

Remember: Verify with the client that each assertion is true and correct within the context of the database that is being modeled.

Case Study 2

First:

- A RepairJob must be conducted on only one Computer
- A RepairJob must be requested by only one Customer
- A RepairJob can use many Items
- A RepairJob must be performed by at least one Repairman
- A RepairJob must have made at least one Deposit
- A RepairJob can have made many Payments
- An Item can be ordered by many Repairmen
- A Customer must own at least one Computer

Second:

- Each Computer must have conducted at least one RepairJob
- Each Customer can request many RepairJobs
- Each Item can be used in many RepairJobs
- Each Repairman can perform many RepairJobs
- Each Deposit must be made for only one RepairJob
- Each Payment must be made for only one RepairJob
- Each Repairman can order many Items
- Each Computer must be owned by only one Customer

Note that from the assertions above, each repairman can perform many repair jobs, which means there is the possibility that a repairman may never perform any repair jobs. This is slightly different to saying that each repairman must perform at least one repair job. Nuances like these need to be verified with the client.

This scenario assumes that in some cases the deposit(s) made can cover the cost of the repair job and no payment will be necessary. Hence, a repair job **can** have many payments made, implying that a repair job can have **no** payments made [towards it]. This scenario also assumes that a computer must be owned by only one customer, which means that if a computer changes ownership it will be entered in the database as two separate entity occurrences. This is fine because we are not using serial numbers to identify computers, and it also means that we can use serial numbers to track owners – a neat trick.

Here is the simplified E-R diagram for the second case study:

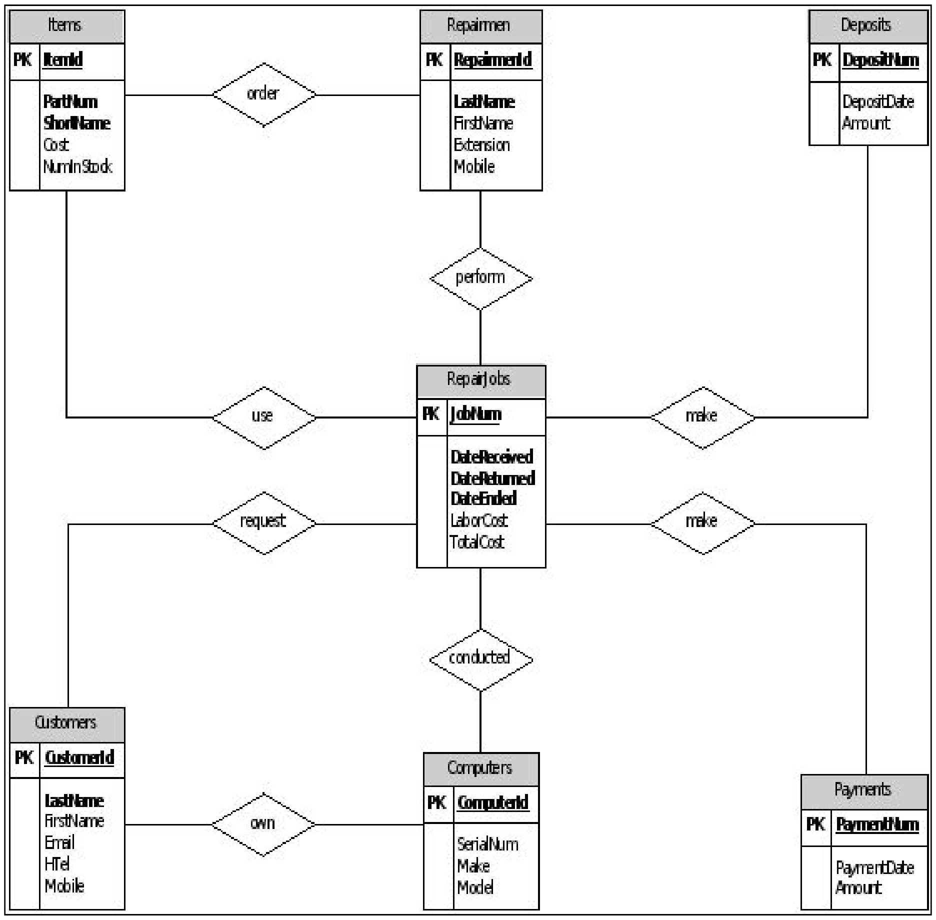

Figure 5.2 – Simplified E-R diagram for Case Study 2

Remember: Verify with the client that each assertion is true and correct within the context of the database that is being modeled.

Case Study 3

First:

- A User can create many Users
- A User can create many LogEntries
- An AccessLevel can be designated to many Users
- A Student must be assigned to only one User [login]
- A Lecturer must be assigned to only one User [login]
- A Course can be the prerequisite for many [other] Courses
- A Lecturer can lecture many Courses
- A Lecturer can lecture many ScheduledClasses
- A Course can be scheduled many times in the Schedule [of classes]
- A Semester can be created for many ScheduledClasses
- A Student can register for many ScheduledClasses

Second:

- Each User must be created by only one [other] User
- Each LogEntry must be created by only one User
- Each User must be designated only one AccessLevel
- Each User [login] can be assigned to only one Student
- Each User [login] can be assigned to only one Lecturer
- Each Course can have as prerequisites many [other] Courses
- Each Course can be lectured by many Lecturers
- Each ScheduledClass must be lectured by only one Lecturer
- Each ScheduledClass must be scheduled for only one Course
- Each ScheduledClass must be created for only one Semester
- Each ScheduledClass can be registered for by many Students

It is important to note that in the assertions above the entity *ScheduledClasses* or *Schedule [of classes]* refers collectively to the different schedules that are made for classes, and *ScheduledClass* is the entity occurrence of this entity class.

Here is the simplified E-R diagram for the third case study:

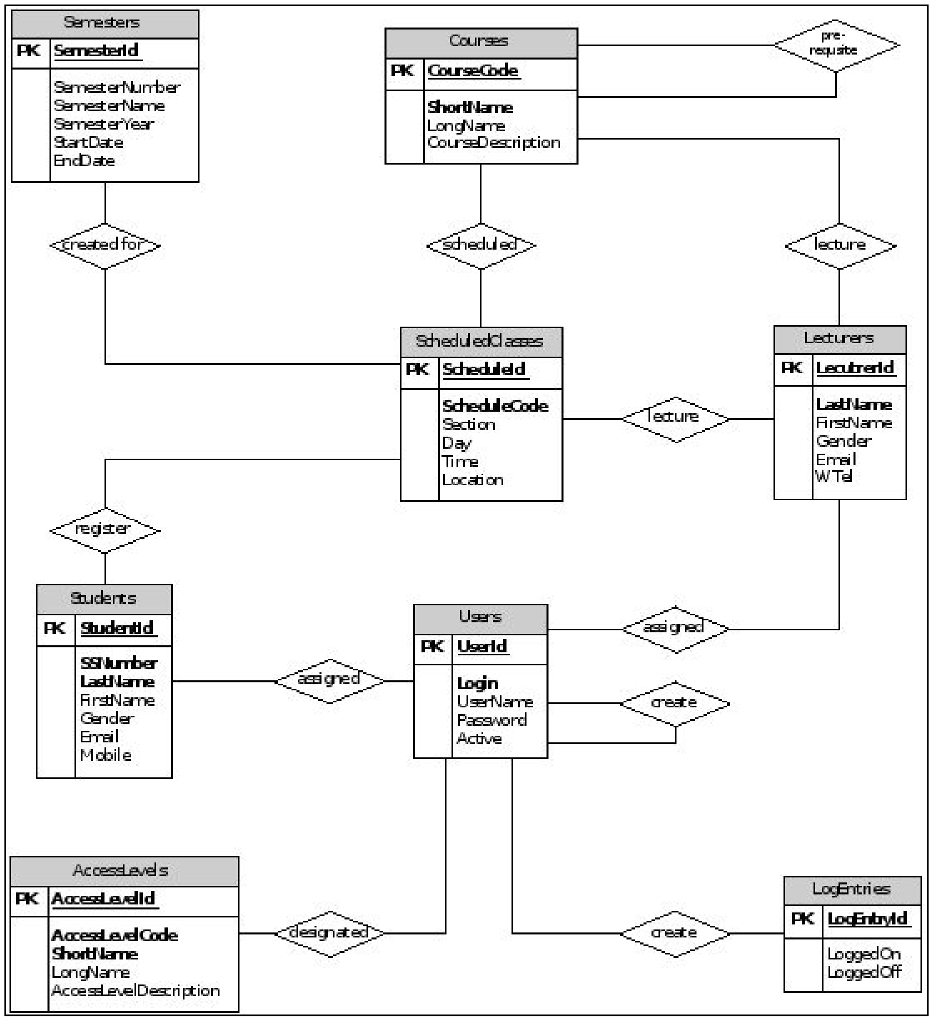

Figure 5.3 – Simplified E-R diagram for Case Study 3

Remember: Verify with the client that each assertion is true and correct within the context of the database that is being modeled.

Summary

This chapter focused on how to create a list of assertions from the simplified E-R diagram derived in Step 3 of the six-step process. Here is a list of steps to create the list of assertions:

Step 4-1: List assertions using the simplified E-R diagram

Each relationship on the simplified E-R diagram will yield two assertions because the relationship is looked at from two directions: from *Entity A* to *Entity B*, then in the other direction from *Entity B* to *Entity A*. Therefore, there will be twice as many assertions as there are relationships.

First: Look at each relationship from *Entity A* to *Entity B*, and write out the relationship in words, using the entities involved in the relationship, the optionalities, and cardinalities. This is how assertions should be written:

*Entity Occurrence **optionality** relationship **cardinality** Entity Occurrence or Entity Class*

Second: Look at each relationship in reverse, from *Entity B* to *Entity A*, and write out the relationship in words, using the entities involved in the relationship, the optionalities, and cardinalities.

Remember: Verify with the client that each assertion is true and correct within the context of the database that is being modeled.

Review Questions

1. What is an assertion?
2. What is optionality?
3. What is cardinality?
4. What is an entity occurrence?
5. What is an entity class?
6. Why are assertions important?
7. Using an appropriate example, explain the difference between entity occurrence and entity class.
8. Using an appropriate example, explain the uses of optionality and cardinality.
9. Using an appropriate example, explain why it is important to verify each assertion with the client.
10. Using an appropriate example, describe the process of listing assertions for all relationships depicted on the simplified E-R diagram.

Exercises

Consider the following simplified E-R diagram for an online store that sells designer T-Shirts:

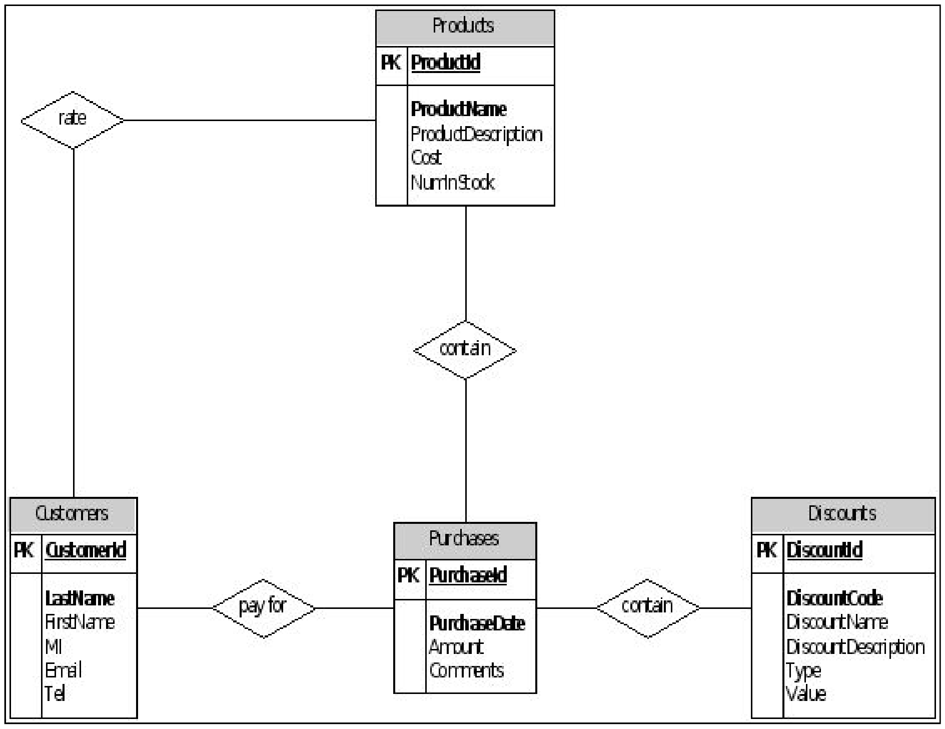

Figure 5.4 – Simplified E-R diagram for online store

1. List the assertions for the simplified E-R diagram in figure 5.4 above, stating any assumptions you make.

Chapter 6

The Six Steps – Step 5

Create detailed E-R diagram using assertions

This chapter combines the assertions generated in the previous step, Step 4, with the simplified E-R diagram created in Step 3, to create a detailed E-R diagram. This detailed E-R diagram gives a complete conceptual model of the data, one that represents the user's view of the data and the logical structure of the database.

At the end of this chapter, readers should be able to combine the assertions with the simplified E-R diagram to create a detailed E-R diagram.

Step 5: Create detailed E-R diagram using assertions

This step is the easiest of the six steps in the six-step process. All that has to be done in this step is to place the assertions on the simplified E-R diagram, thus creating the detailed E-R diagram. The assertions are placed on the detailed E-R diagram between the entity and the relationship using the following convention:

optionality:cardinality

where:

Optionality ‘0’ – Can (optional)

Optionality ‘1’ – Must (obligatory)

Cardinality ‘1’ – Only one

Cardinality ‘N’ – Many or At least one

Here is a list of all the possible combinations of **optionality:cardinality** that can appear on the detailed E-R diagram:

0:1 – [Entity] **can** [*relationship*] **only one** [Entity]

0:N – [Entity] **can** [*relationship*] **many** [Entity]; or [Entity] **can** [*relationship*] **at least one** [Entity]

1:1 – [Entity] **must** [*relationship*] **only one** [Entity]

1:N – [Entity] **must** [*relationship*] **many** [Entity]; or [Entity] **must** [*relationship*] **at least one** [Entity]

The assertions are always positioned between the entity (rectangle) and the relationship (diamond) on the simplified E-R diagram, and they have a different meaning depending on which side of the relationship they appear on. However, the assertion is always written and interpreted as **optionality:cardinality**. The examples below show how to correctly insert assertions on simplified E-R diagrams.

Assertion (optionality:cardinality): *A Customer* ***can*** *make* ***many*** *Payments* ***(0:N)***

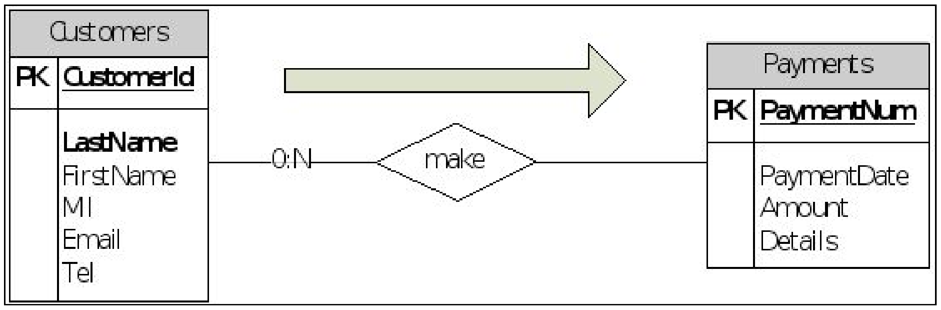

Figure 6.1 – Inserting Assertions

Pay attention to the arrow and the interpretation of the *optionality*:*cardinality* on the diagram.

Assertion (optionality:cardinality): *Each Payment* ***must*** *be made by* ***only one*** *Customer* ***(1:1)***

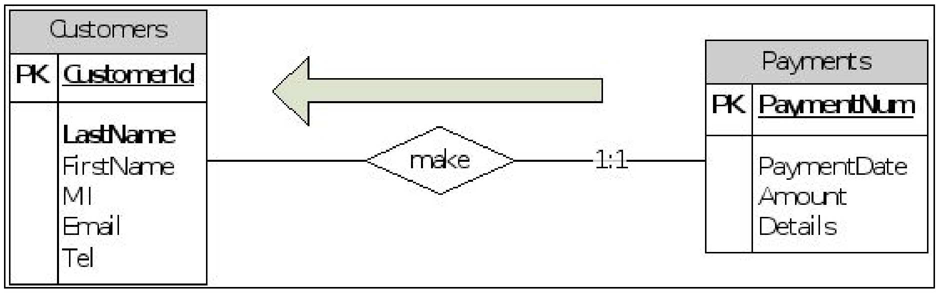

Figure 6.2 – Inserting Assertions (reverse)

Pay attention to the arrow and the interpretation of the *optionality*:*cardinality* on the diagram. Even though the arrow has changed direction, the interpretation of the *optionality*:*cardinality* has not changed. It remains *optionality*:*cardinality*, and the assertion is always placed closest to the first entity occurrence stated in the assertion.

Together, the two previous assertions appear as follows:

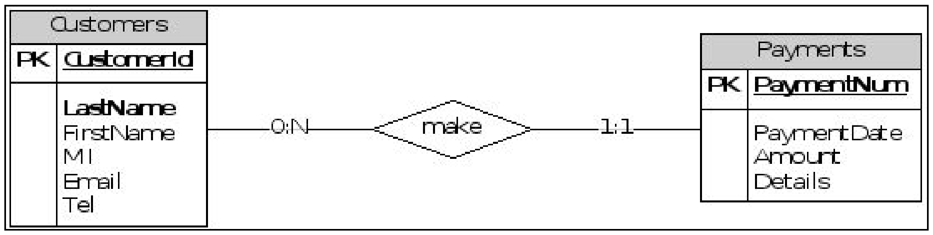

Figure 6.3 – Inserting Assertions (completed)

Step 5-1: Use assertions and the simplified E-R diagram to create the detailed E-R diagram

Each relationship on the simplified E-R diagram yielded two assertions. These two assertions will now appear on either side of the relevant relationship as described above. Go through each relationship and its associated assertions one at a time until each assertion has been inserted on the diagram.

To use the assertions and the simplified E-R diagram to create a detailed E-R diagram take the following actions:

First:

List the generated assertions and include **(optionality:cardinality)** at the end of the assertion.

Second:

Insert the generated assertions as **optionality:cardinality** one at a time on the simplified E-R diagram, in the correct position, creating the detailed E-R diagram. Do this one relationship at a time, one assertion at a time.

Remember, each relationship has two associated assertions and should have one assertion on either side of it in the detailed E-R diagram. Also, each assertion is always placed closest to the first entity occurrence stated in the assertion.

It is advisable to have the simplified E-R diagram handy.

Case Study 1

Here is the simplified E-R diagram along with the list of assertions for the first case study:

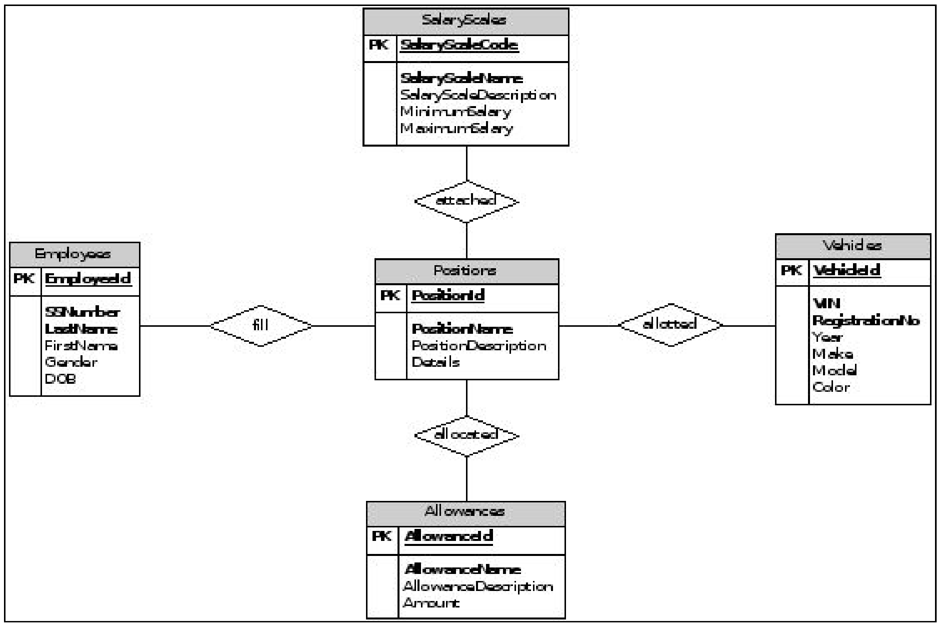

Figure 6.4 – Simplified E-R diagram for Case Study 1

First:

List of assertions with **(optionality:cardinality)**:

- A Position can be allocated many Allowances **(0:N)**
- Each Allowance can be allocated to many Positions **(0:N)**
- A Position can be allotted only one Vehicle **(0:1)**
- Each Vehicle must be allotted to only one Position **(1:1)**
- A Position must be attached to only one SalaryScale **(1:1)**
- Each SalaryScale can be attached to many Positions **(0:N)**
- A Position can be filled by many Employees **(0:N)**
- Each Employee can fill many Positions **(0:N)**

Second:

Insert the generated assertions as **optionality:cardinality** one at a time on the simplified E-R diagram. The resulting detailed E-R diagram for Case Study 1 can be seen below.

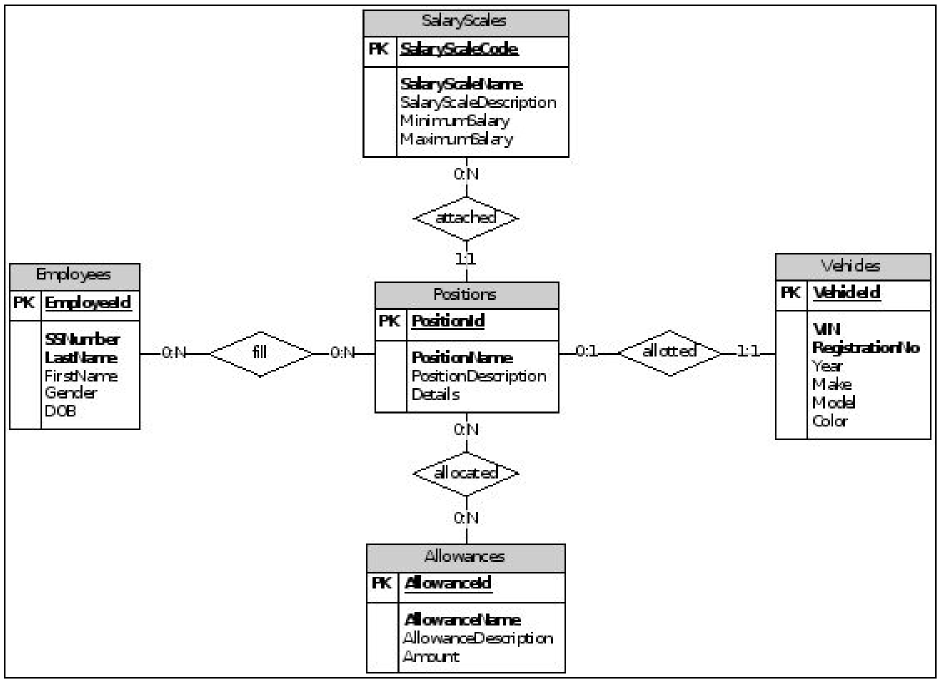

Figure 6.5 – Detailed E-R diagram for Case Study 1

Remember: In the detailed E-R diagram each assertion is always placed closest to the first entity occurrence stated in the assertion; and each relationship has two associated assertions, one on either side of the relationship.

Case Study 2

Here is the simplified E-R diagram along with the list of assertions for the second case study:

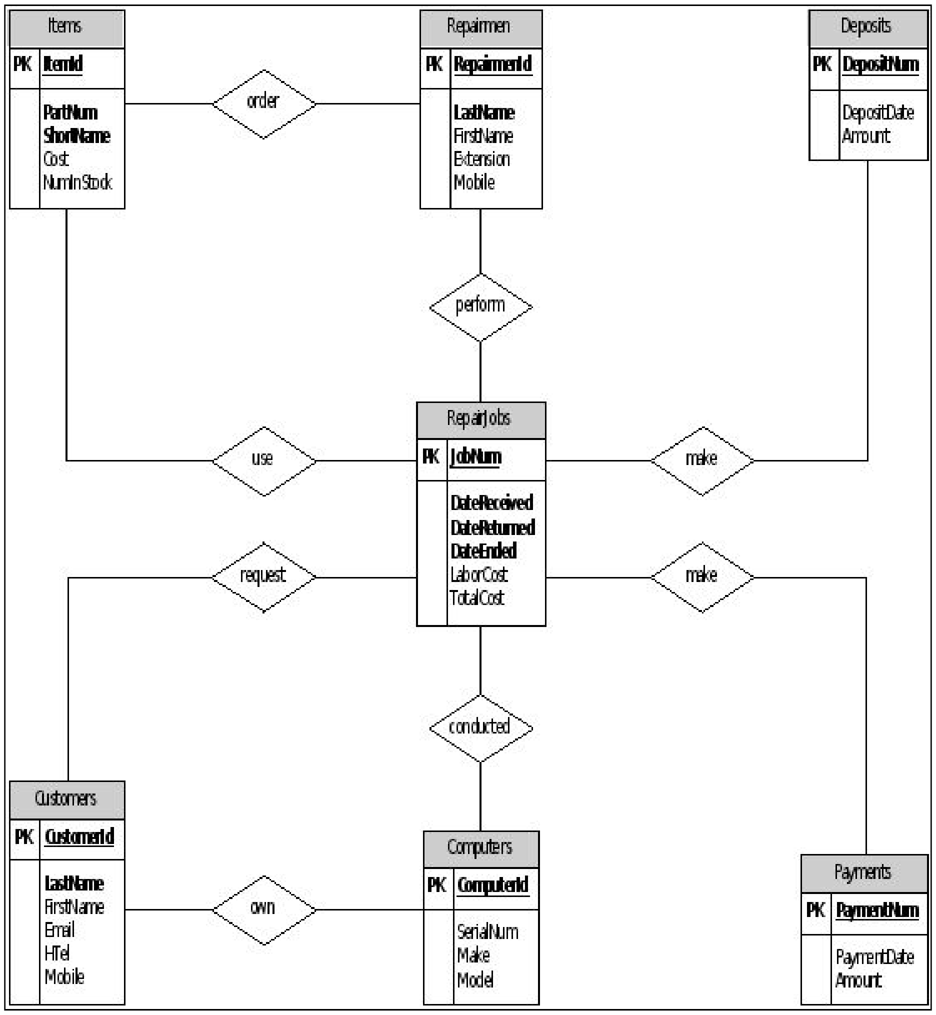

Figure 6.6 – Simplified E-R diagram for Case Study 2

First:

List of assertions with **(optionality:cardinality)**:

- A RepairJob must be conducted on only one Computer **(1:1)**
- Each Computer must have conducted at least one RepairJob **(1:N)**
- A RepairJob must be requested by only one Customer **(1:1)**
- Each Customer can request many RepairJobs **(0:N)**
- A RepairJob can use many Items **(0:N)**
- Each Item can be used in many RepairJobs **(0:N)**
- A RepairJob must be performed by at least one Repairman **(1:N)**
- Each Repairman can perform many RepairJobs **(0:N)**
- A RepairJob must have made at least one Deposit **(1:N)**
- Each Deposit must be made for only one RepairJob **(1:1)**
- A RepairJob can have made many Payments **(0:N)**
- Each Payment must be made for only one RepairJob **(1:1)**
- An Item can be ordered by many Repairmen **(0:N)**

- Each Repairman can order many Items **(0:N)**
- A Customer must own at least one Computer **(1:N)**
- Each Computer must be owned by only one Customer **(1:1)**

Second:

Insert the generated assertions as **optionality:cardinality** one at a time on the simplified E-R diagram. The resulting detailed E-R diagram for Case Study 2 can be seen overleaf.

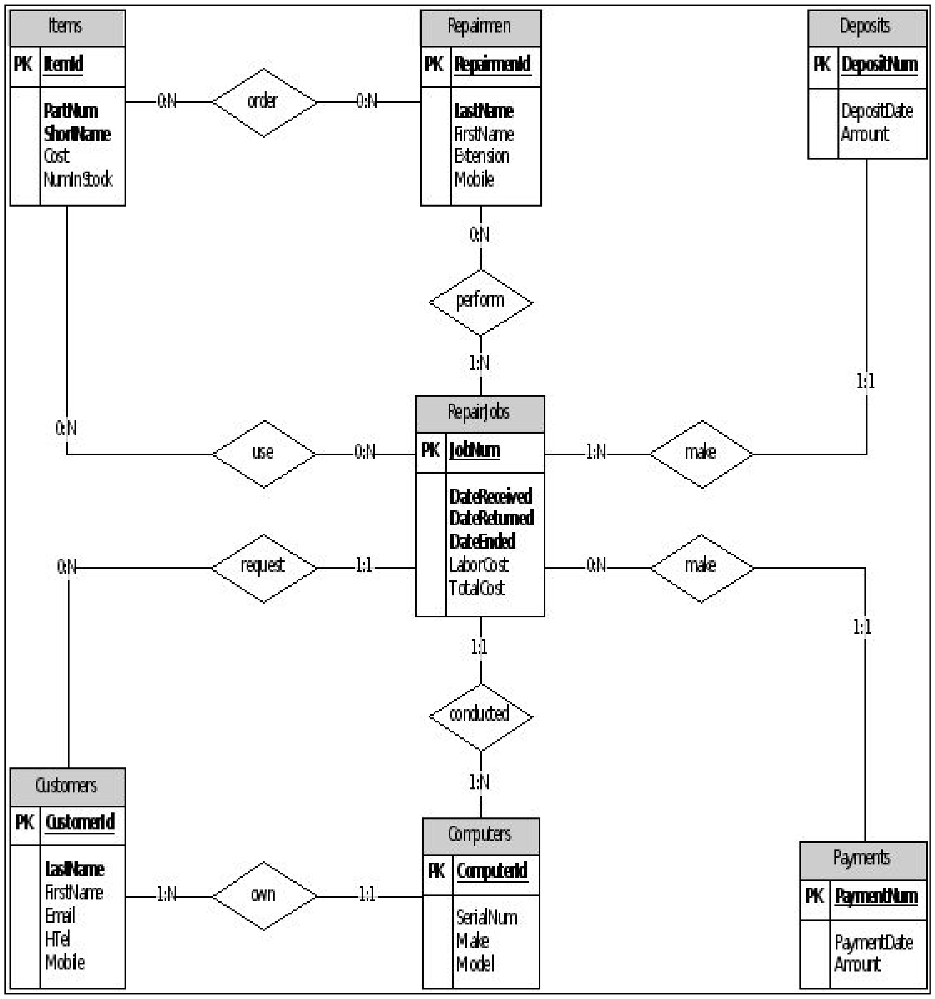

Figure 6.7 – Detailed E-R diagram for Case Study 2

Remember: In the detailed E-R diagram each assertion is always placed closest to the first entity occurrence stated in the assertion; and each relationship has two associated assertions, one on either side of the relationship.

Case Study 3

Here is the simplified E-R diagram along with the list of assertions for the third case study:

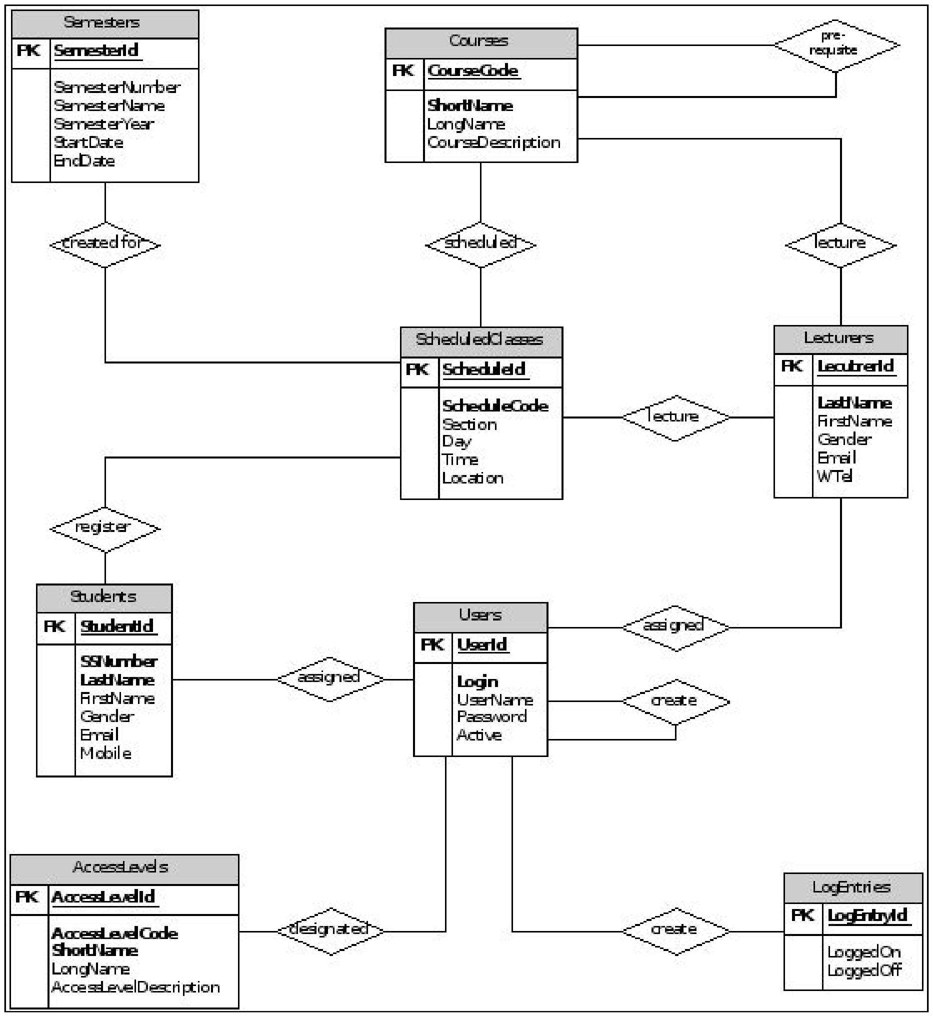

Figure 6.8 – Simplified E-R diagram for Case Study 3

First:

List of assertions with **(optionality:cardinality)**:

- A User can create many Users **(0:N)**
- Each User must be created by only one [other] User **(1:1)**
- A User can create many LogEntries **(0:N)**
- Each LogEntry must be created by only one User **(1:1)**
- An AccessLevel can be designated to many Users **(0:N)**
- Each User must be designated only one AccessLevel **(1:1)**
- A Student must be assigned to only one User [login] **(1:1)**
- Each User [login] can be assigned to only one Student **(0:1)**
- A Lecturer must be assigned to only one User [login] **(1:1)**
- Each User [login] can be assigned to only one Lecturer **(0:1)**
- A Course can be the prerequisite for many [other] Courses **(0:N)**
- Each Course can have as prequisites many [other] Courses **(0:N)**
- A Lecturer can lecture many Courses **(0:N)**

- Each Course can be lectured by many Lecturers **(0:N)**
- A Lecturer can lecture many ScheduleClasses **(0:N)**
- Each ScheduledClass must be lectured by only one Lecturer **(1:1)**
- A Course can be scheduled many times in the Schedule [of classes] **(0:N)**
- Each ScheduledClass must be scheduled for only one Course **(1:1)**
- A Semester can be created for many ScheduledClasses **(0:N)**
- Each ScheduledClass must be created for only one Semester **(1:1)**
- A Student can register for many ScheduledClasses **(0:N)**
- Each ScheduledClass can be registered for by many Students **(0:N)**

Second:

Insert the generated assertions as **optionality:cardinality** one at a time on the simplified E-R diagram. The resulting detailed E-R diagram for Case Study 3 can be seen overleaf:

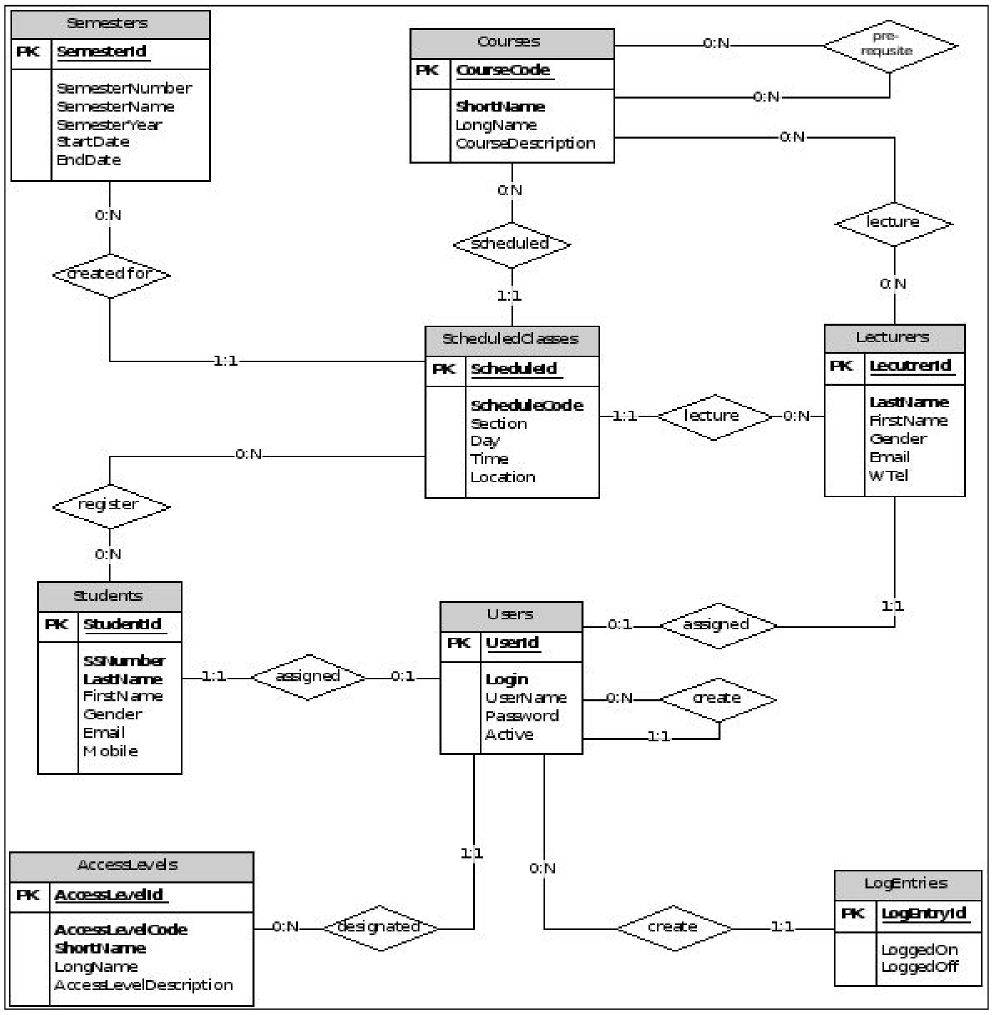

Figure 6.9 – Detailed E-R diagram for Case Study 3

Remember: In the detailed E-R diagram each assertion is always placed closest to the first entity occurrence stated in the assertion; and each relationship has two associated assertions, one on either side of the relationship.

Summary

This chapter combines the assertions generated in the previous step, Step 4, with the simplified E-R diagram created in Step 3 to create a detailed E-R diagram. Here is what to do to create a detailed E-R diagram:

Step 5-1: Use assertions and simplified E-R diagram to create detailed E-R diagram

Each relationship on the simplified E-R diagram yielded two assertions. These two assertions will now appear on either side of the relevant relationship. Go through each relationship and its associated assertions one at a time until they have all been inserted on the diagram.

First: List the assertions and include **(optionality:cardinality)** at the end of each assertion.

Second: Insert the generated assertions as **optionality:cardinality** one at a time on the simplified E-R diagram, in the correct position, creating the detailed E-R diagram. Do this one relationship at a time, one assertion at a time.

Remember: In the detailed E-R diagram each assertion is always placed closest to the first entity occurrence stated in the assertion; and each relationship has two associated assertions, one on either side of the relationship.

Review Questions

1. What purpose does the detailed E-R diagram serve?

2. How is the detailed E-R diagram different from the simplified E-R diagram?

3. Using an appropriate example, explain how optionality and cardinality are used to create a detailed E-R diagram.

4. Using an appropriate example, describe the steps involved in creating the detailed E-R diagram from the simplified E-R diagram.

Exercises

Consider the following simplified E-R diagram for an online store that sells designer T-Shirts:

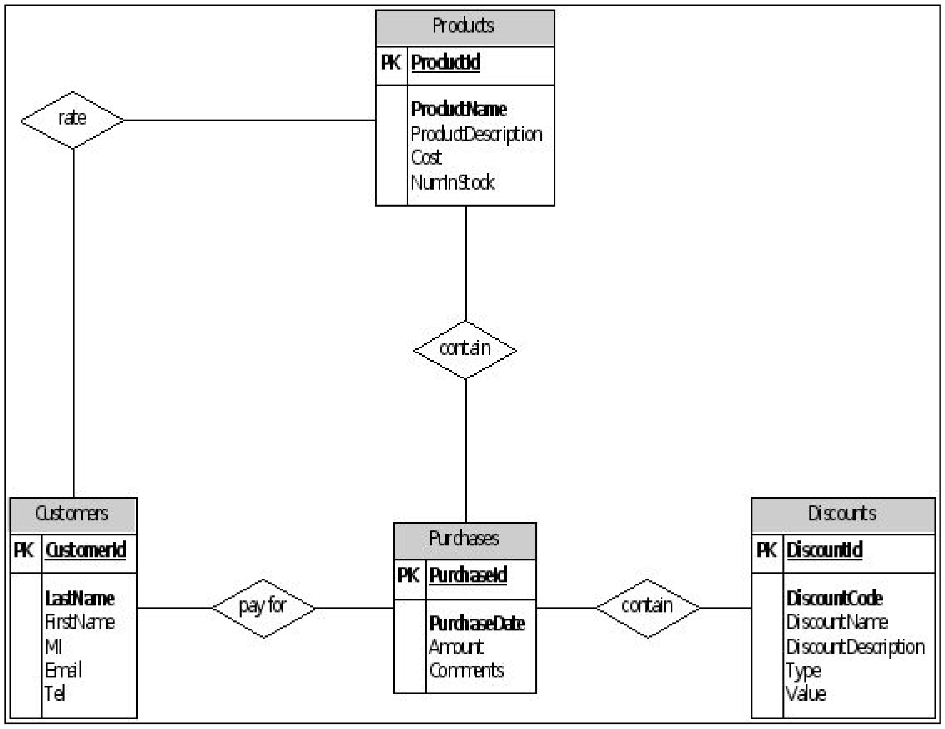

Figure 6.10 – Simplified E-R diagram for online store

1. List the assertions for the simplified E-R diagram in figure 6.10 above, stating any assumptions you make about the client.

2. Use the assertions you listed in part 1 above to create a detailed E-R diagram for the online store database.

Chapter 7

The Six Steps – Step 6

Transform the detailed E-R diagram into an implementable R-M diagram

This is the sixth and final step of the six-step process. However, it is one of the most important steps because any mistake made here will immediately show up on the database design. This chapter describes how to transform the detailed E-R diagram created in the previous step, Step 5, into a Crow's Foot Relational Model diagram that can be easily implemented on any RDBMS.

At the end of this chapter, readers should be able to convert previously created detailed E-R diagrams into implementable Crow's Foot R-M diagrams.

Step 6: Transform the detailed E-R diagram into an implementable R-M diagram

This is the final step of the six-step process. However, it is the most complicated. It involves transforming the detailed E-R diagram created in the previous step into a Crow's Foot Relational Model diagram.

Transforming an E-R diagram into an R-M diagram is not just a change of diagrams, but also a change of *view*. The E-R diagram is a **conceptual model** and represents the *user's view* of the data and the logical structure of the database, whereas the R-M diagram is an **implementation model** and represents the *developer's view* of the data and the physical structure of the database.

As mentioned in Chapter 1, there are two types of models in relational database design, *conceptual models*, and *implementation models*. *Conceptual models* are concerned with the logical nature of the data and what is being represented, and *implementation models* are concerned with the physical nature of the data and with how the data will be represented in the database. The E-R diagram is a *conceptual model* and the R-M diagram is an *implementation model*.

It is important that both views of the data, the *user's view* and the *developer's view*, are congruent and accurate. It is also important that one follows from the other—that the *developer's view* follows from the *user's view*—and not the other way around. The *user's view* should be developed first and the *developer's view* derived from the *user's view*, which is why the E-R diagram is created first and the R-M diagram derived from this E-R diagram.

The R-M diagram uses slightly different terminology and symbols for entities and relationships. The *Entity* on the E-R diagram remains but is now called a *Relation*, defined in Chapter 1 as *"a two dimensional table, which contains rows and columns."* The previous five steps ensure that no modification is necessary to convert an *Entity* into a *Relation*. Each *Relation* must have a primary key and may have foreign keys. Relationships can also exist between Relations, but these are represented differently on the R-M diagram. Here are examples:

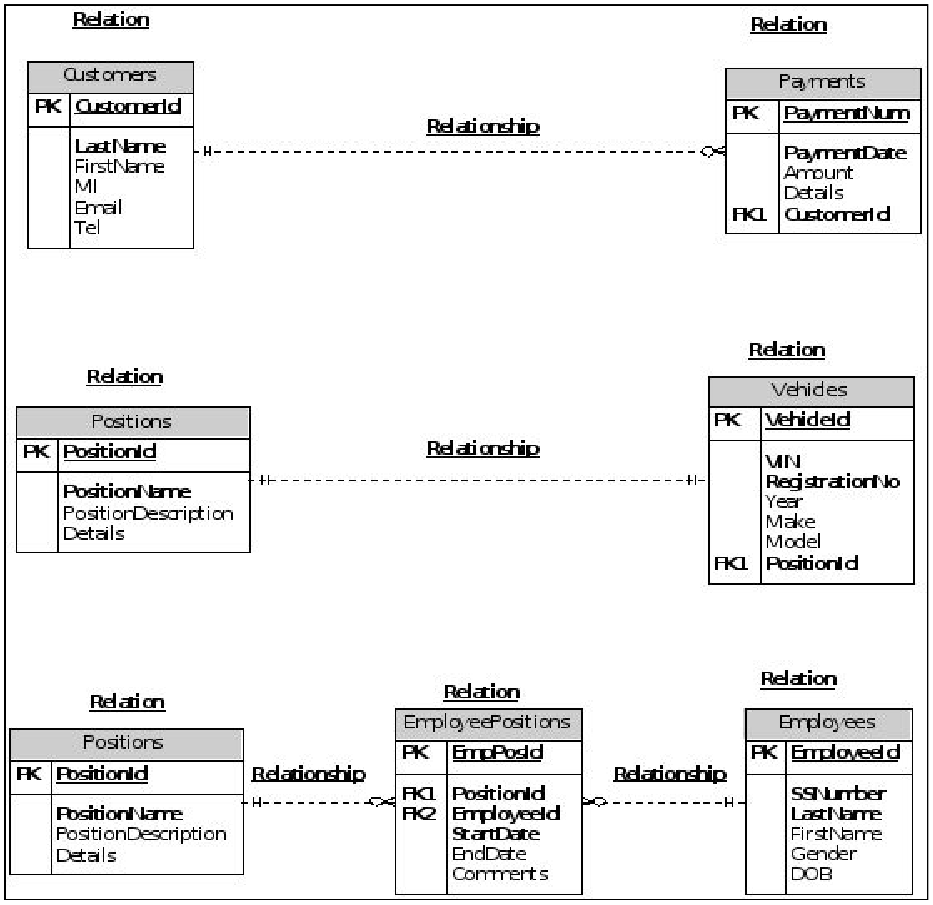

Figure 7.1 – Notation for Crow’s Foot Relational Model diagram

The diagram above is an example of a Crow’s Foot Relational Model diagram and shows the Relations, relationships, and how Relations are connected via relationships. In the Relations, *FK1* and *FK2* refer to foreign keys, defined in Chapter 1 as *“a primary key in one Relation (table) that appears as a column in another Relation (table) and is used to join the two Relations (tables) together in a relationship.”*

The diagram below gives a breakdown of the Crow’s Foot notation for relationships as they will be used in this book.

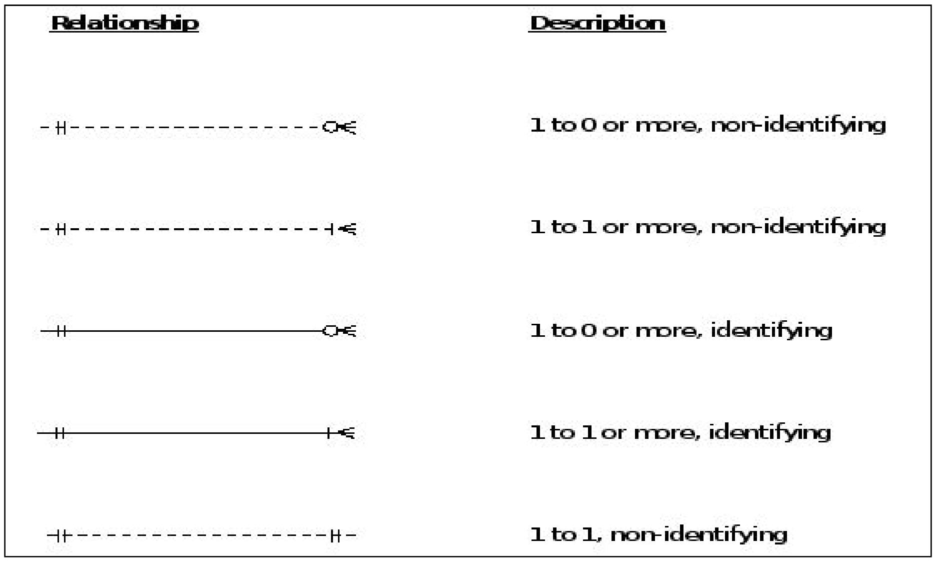
Relationship
Description
1 to 0 or more, non-identifying
1 to 1 or more, non-identifying
1 to 0 or more, identifying
1 to 1 or more, identifying
1 to 1, non-identifying

Figure 7.2 – Notation for relationships on Crow’s Foot Relational Model diagrams

Crow’s feet can only represent one-to-one and one-to-many relationships. Many-to-many relationships are created using a *join-table*. An example of this is shown as *EmployeePositions* in Figure 7.1.

Transforming the detailed E-R diagram into a Crow’s Foot R-M diagram is a three-step process. Each step is performed iteratively on the detailed E-R diagram, and the results are combined to create the Crow’s Foot R-M diagram.

Step 6-1: Many-to-many relationships

This step will transform many-to-many relationships on the detailed E-R diagram into many-to-many relationships on the R-M diagram. Many-to-many relationships in the E-R diagram are identified by a cardinality of *N (many or at least one)* on both sides of the relationship (diamond), as shown in the following example:

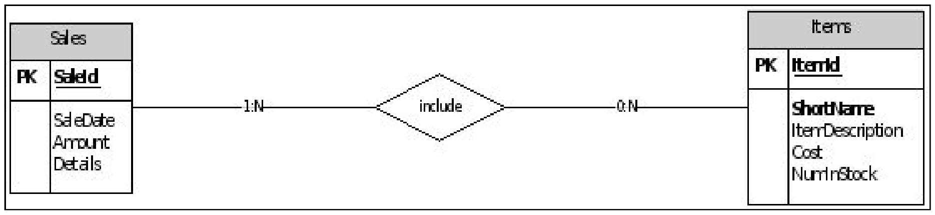

Figure 7.3 – Many-to-many relationship on detailed E-R diagram

Here is what to do to transform these relationships into many-to-many relationships on an R-M diagram:

First:

Identify all many-to-many relationships. These relationships are identified by a cardinality of *N (many or at least one)* on both sides of the relationship, as shown in figure 7.3.

Second:

Remove both the relationship and the connectors to that relationship. Replace the relationship with a new *Relation*. The name of this new Relation should be a combination of the names of the two Relations that are on either side of the removed relationship. Combine these names in a manner that makes sense to form the name of the new Relation.

Third:

Create new *1 to 0 or more*, or *1 to 1 or more* relationships that connect the two existing Relations to the new Relation. Use a *1 to 0 or more* relationship if the optionality of the old relationship was *0 (can)*, and a *1 to 1 or more* if the optionality was *1 (must)*.

The many side (Crow's Foot) of the two new relationships should be on the new Relation. Ensure that the primary key for each of the two existing Relations becomes a foreign key in the new Relation, and create a new and separate primary key for the new Relation, such as a unique identifier (Id).

In some instances the two *foreign keys* can combine to form a *composite primary key*, but that depends on the data – whether or not there can be more than one instance of the composite primary key. If you are not sure, just create a new and separate primary key.

Fourth:

Explore whether attributes exist for the newly created Relation. In most cases they will not, but in some cases they will. These attributes must be valid for both foreign keys and not for any one foreign key individually; otherwise that attribute belongs in one of the relations for the foreign keys and not in this new relation.

Here is the resulting representation of the many-to-many relationship on the Crow’s Foot Relational Model diagram with a new and separate primary key and new attributes:

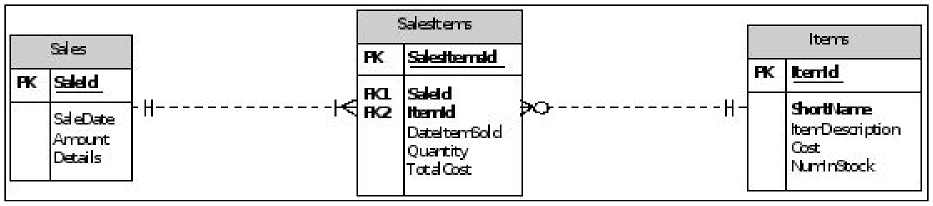

Figure 7.4 – Many-to-many relationship with a new primary key on the Crow’s Foot R-M diagram

Note: In the scenario above, the same item can be included in the same sale more than once – *SaleId* and *ItemId* can have the same composite values more than once.

Here is the resulting representation of the many-to-many relationship on the Crow’s Foot Relational Model diagram with a composite primary key and new attributes:

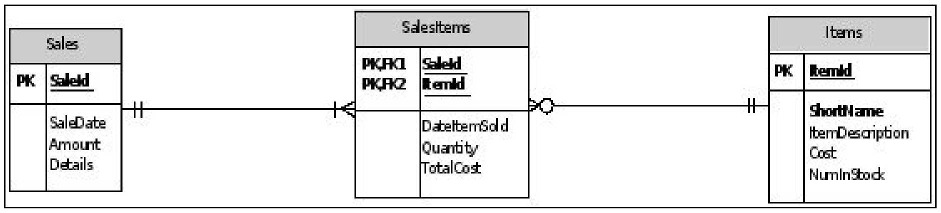

Figure 7.5 – Many-to-many relationship with a composite primary key on the Crow’s Foot R-M diagram

Note: In the scenario depicted in figure 7.5, the same item cannot be included in the same sale more than once – *SaleId* and *ItemId* cannot have the same composite values more than once because they form a composite key.

Case Study 1

First:

The diagram below shows the many-to-many relationships for Case Study 1:

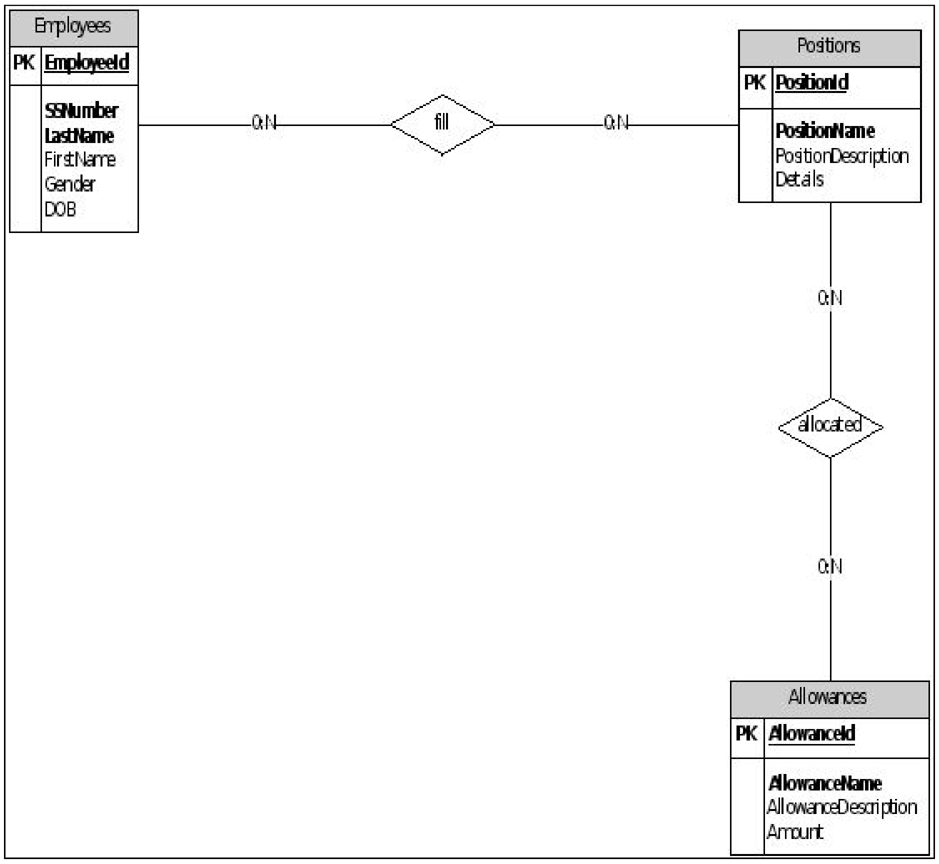

Figure 7.6 – Many-to-many relationships for Case Study 1

Second:

The diagram below shows the new Relations replacing the old relationships and connectors for Case Study 1:

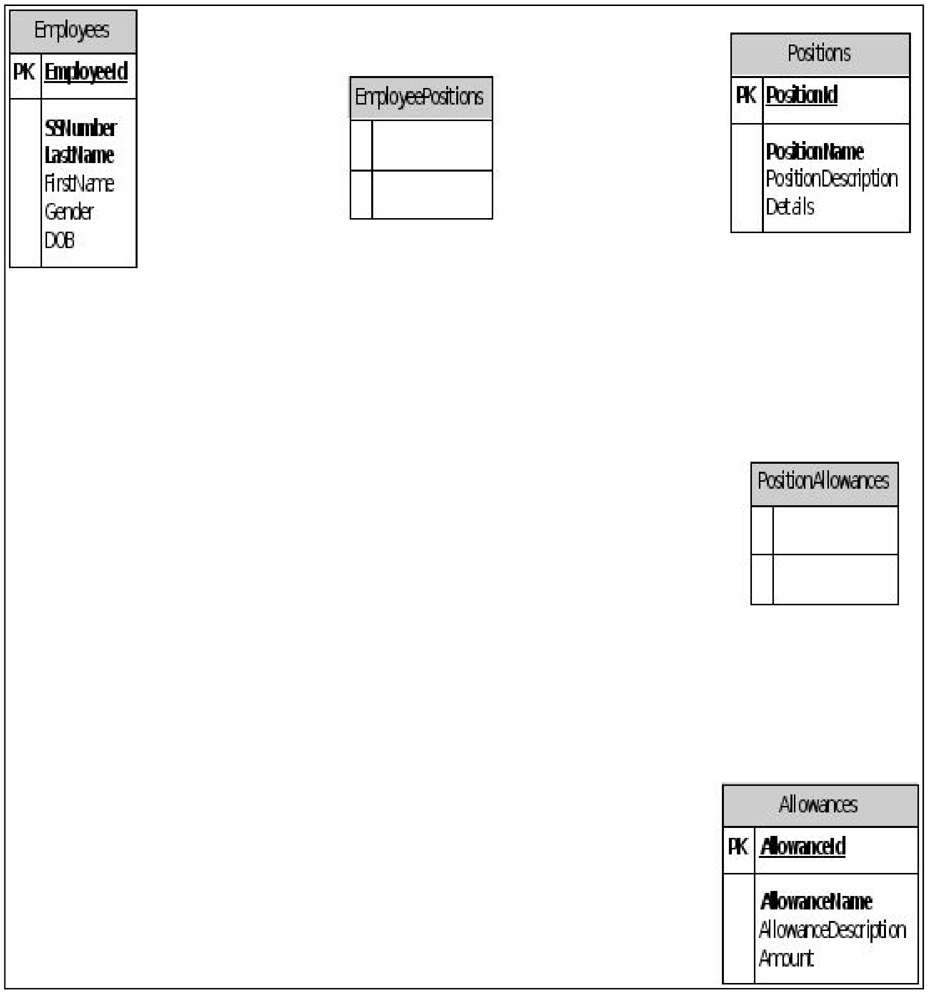

Figure 7.7 – New Relations for Case Study 1

Third:

The diagram below shows the new Relations with new relationships, new primary key, and foreign keys for Case Study 1:

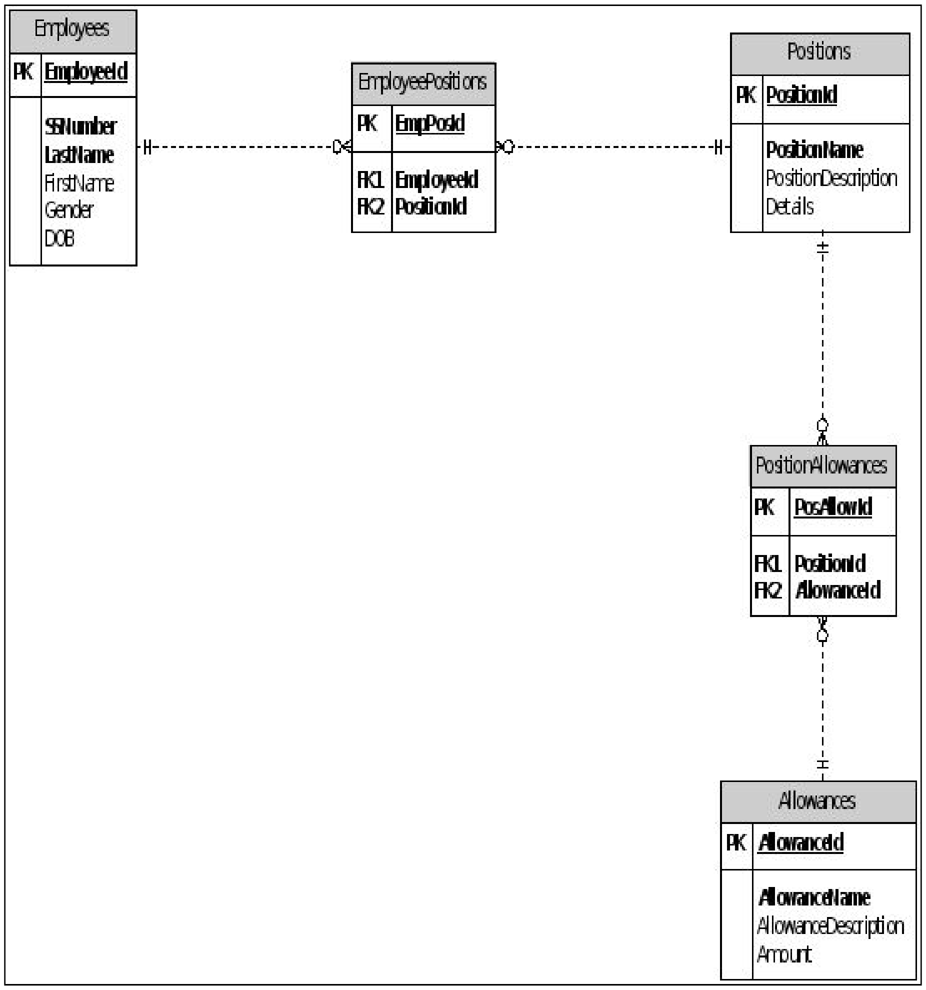

Figure 7.8 – New Relations and relationships for Case Study 1

Fourth:

The diagram below shows the completed many-to-many transformation for Case Study 1, including added attributes:

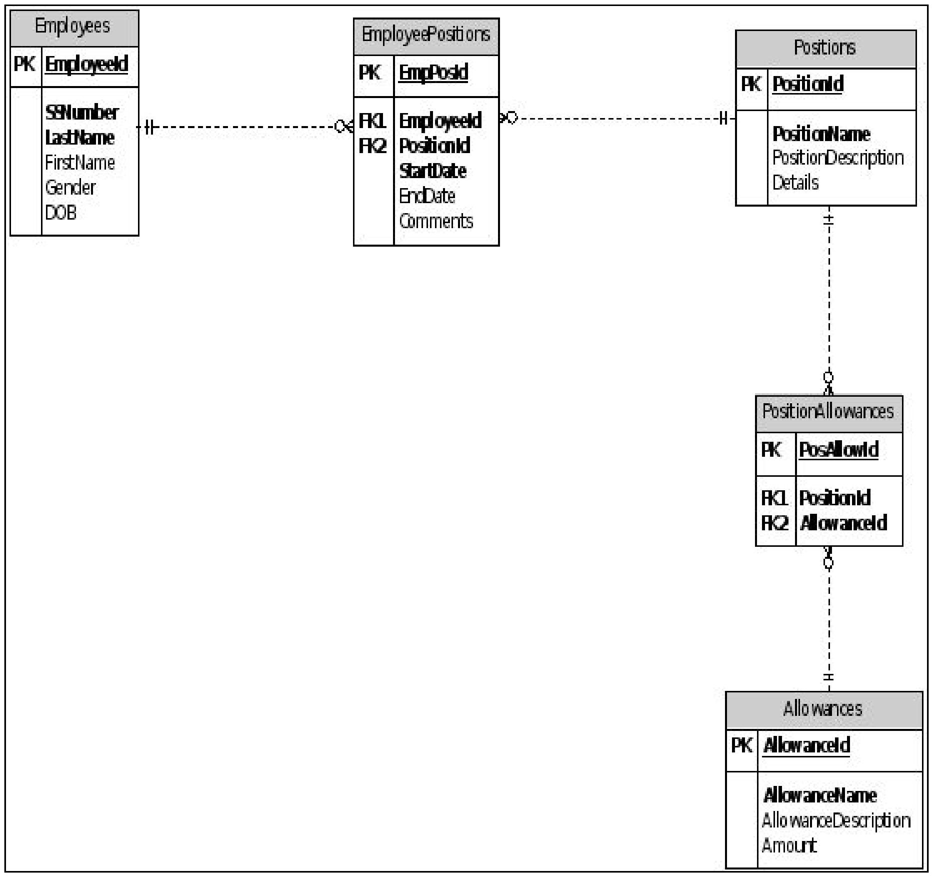

Figure 7.9 – Completed many-to-many transformation for Case Study 1

Remember: Verify that each new Relation contains two foreign keys. Also check to see whether there are possible attributes for the new Relations, and if there are, ensure that these new attributes depend on both foreign keys of the new Relation and not on any one foreign key individually.

Case Study 2

First:

The diagram below shows the many-to-many relationships for Case Study 2:

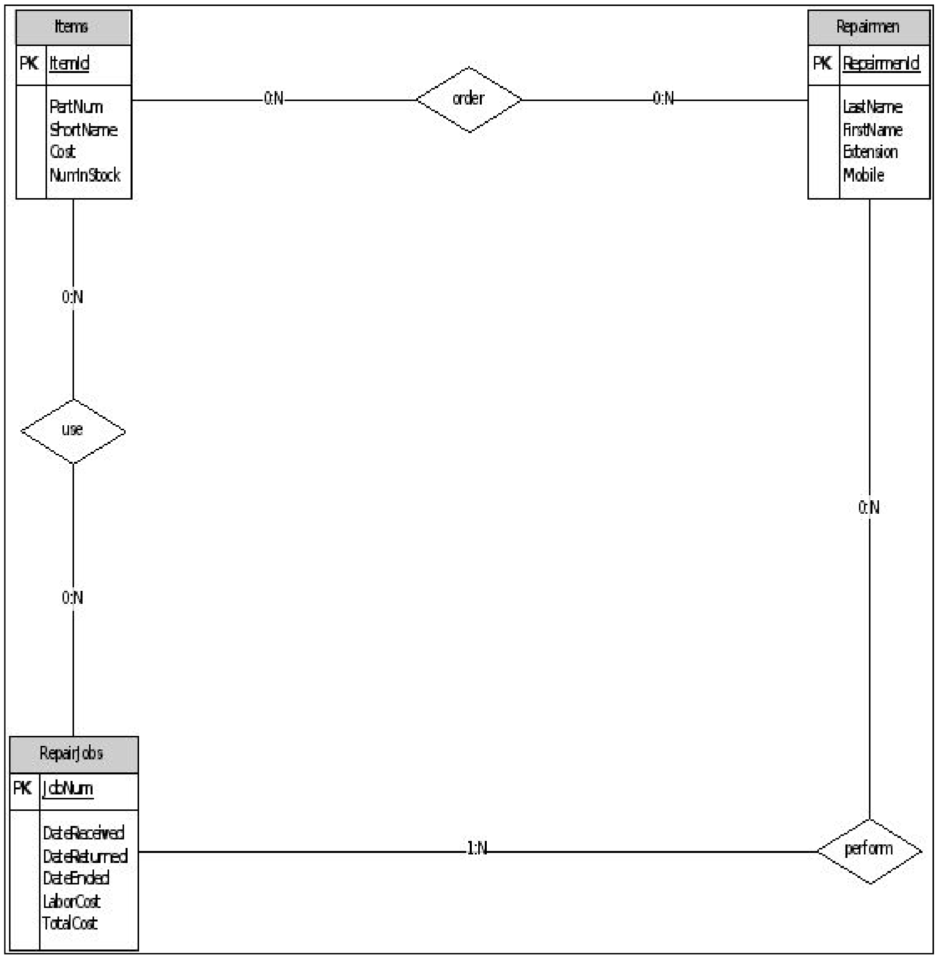

Figure 7.10 – Many-to-many relationships for Case Study 2

Second:

The diagram below shows the new Relations replacing the old relationships and connectors for Case Study 2:

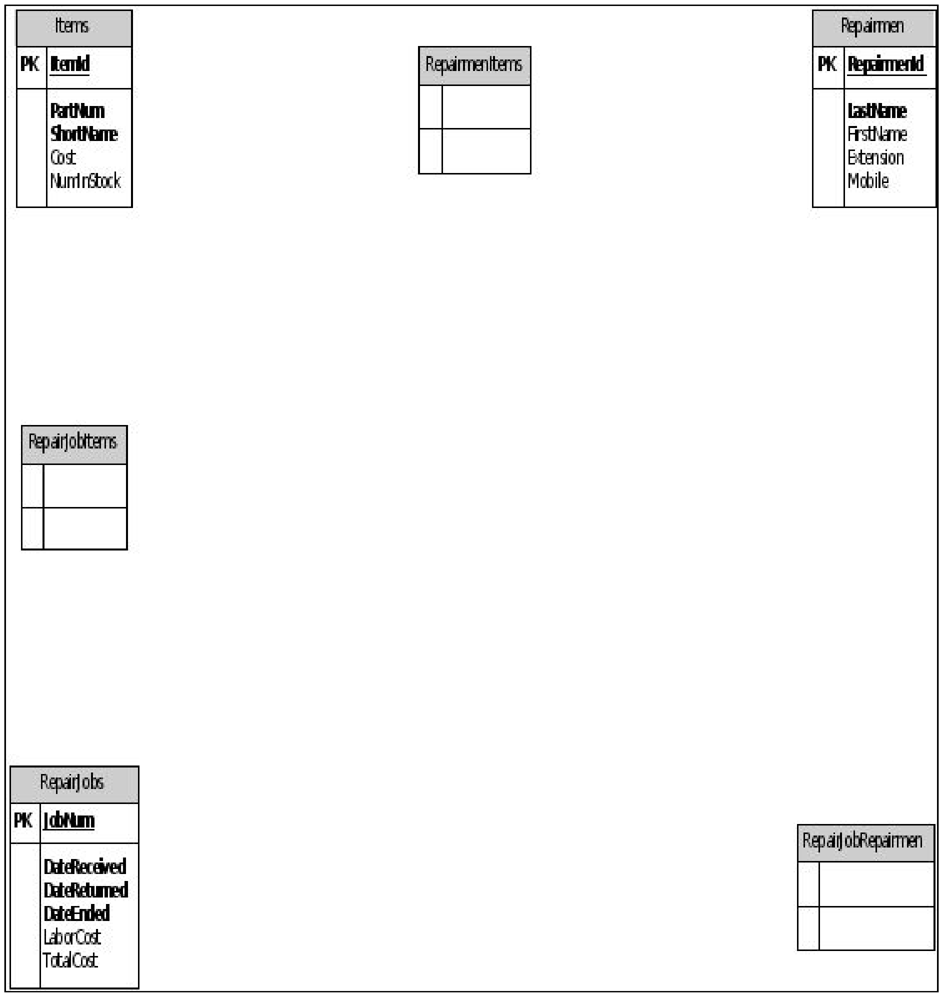

Figure 7.11 – New Relations for Case Study 2

Third:

The diagram below shows the new Relations with new relationships, new primary key, and foreign keys for Case Study 2:

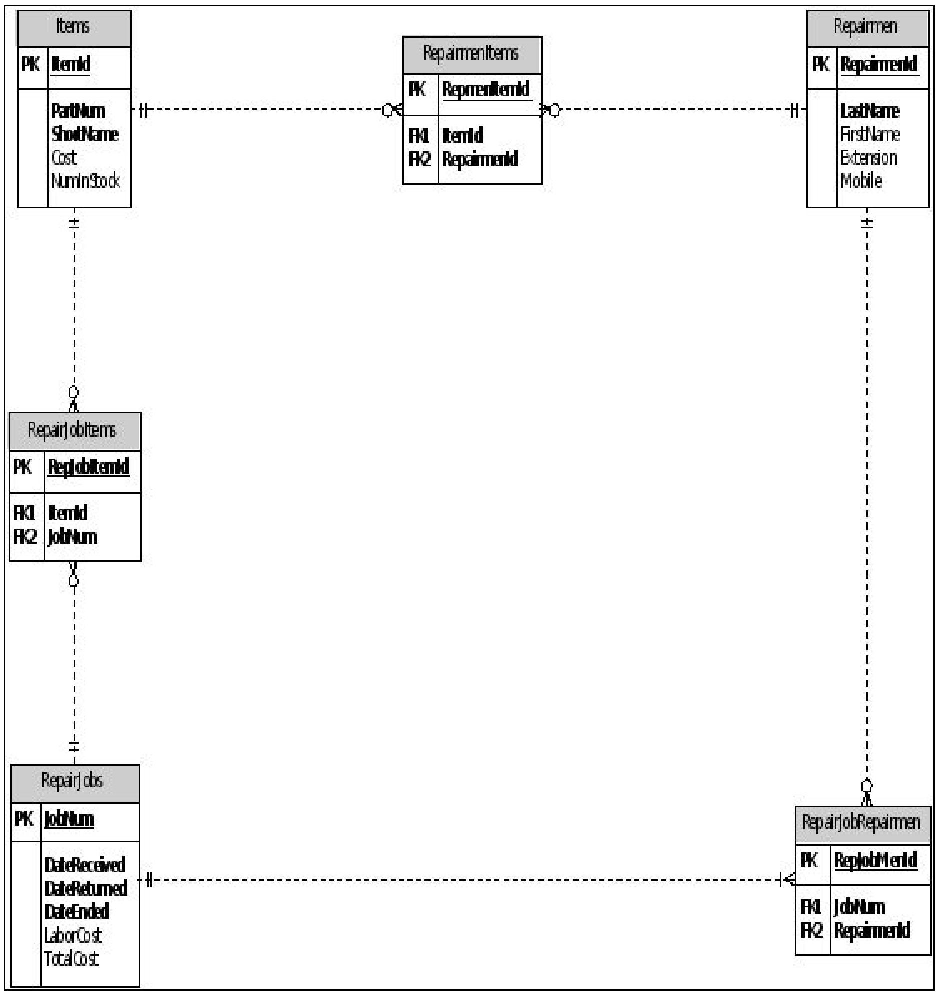

Figure 7.12 – New Relations and relationships for Case Study 2

Fourth:

The diagram below shows the completed many-to-many transformation for Case Study 2, including added attributes:

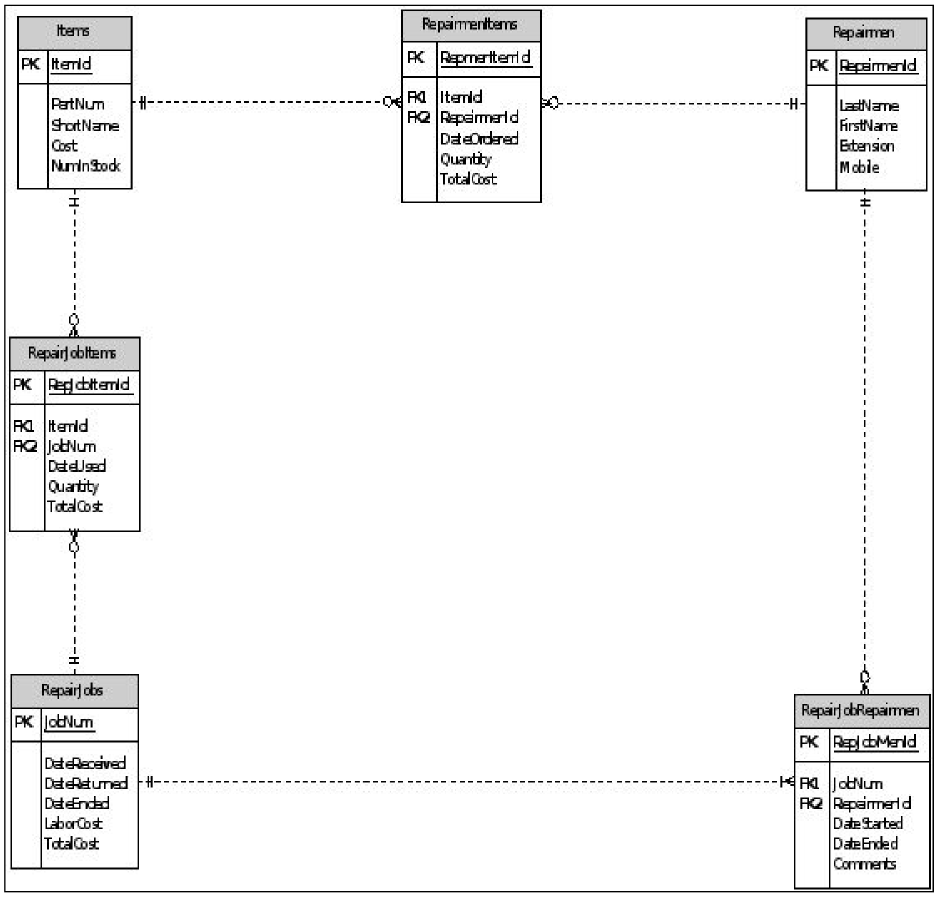

Figure 7.13 – Completed many-to-many transformation for Case Study 2

In the *RepairJobRepairmen* Relation, the *DateStarted* and *DateEnded* attributes represent the date the repairman identified by *ReparimenId* started and ended work on the job identified by *JobNum*. If only one repariman worked on the computer then these dates would be the same as those in the *RepairJob* Relation.

Remember: Verify that each new Relation contains two foreign keys. Also check to see whether there are possible attributes for the new Relations, and if there are, ensure that these new attributes depend on both foreign keys of the new Relation and not on any one foreign key individually.

Case Study 3

First:

The diagram below shows the many-to-many relationships for Case Study 3:

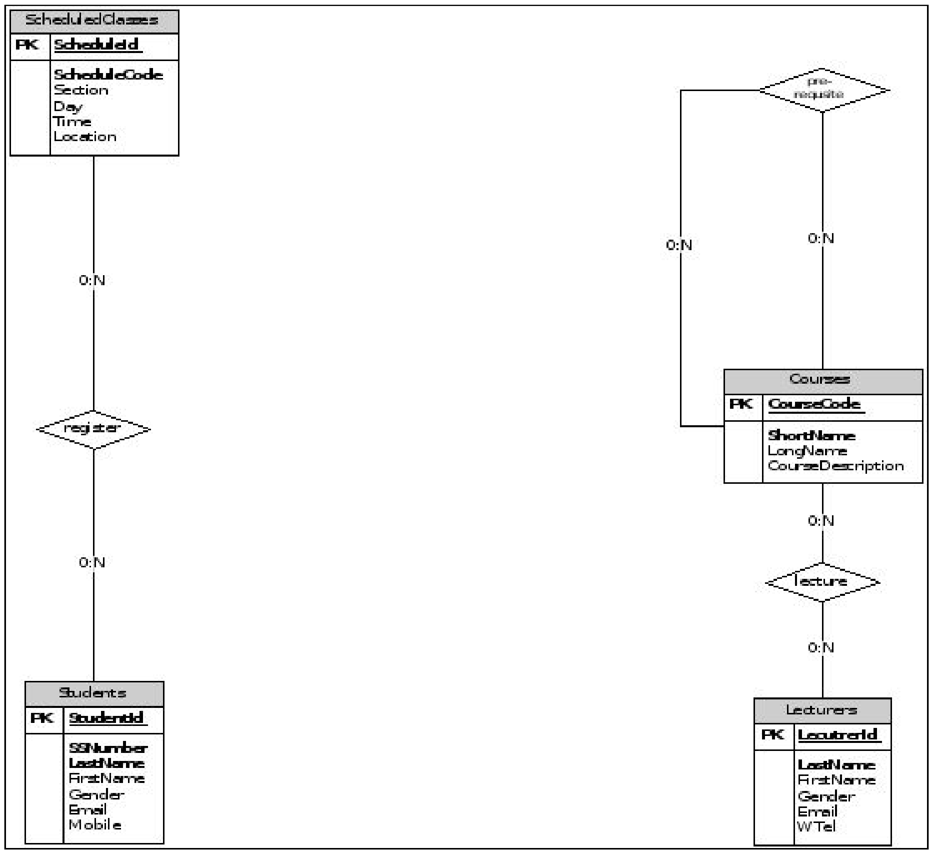

Figure 7.14 – Many-to-many relationships for Case Study 3

Second:

The diagram below shows the new Relations replacing the old relationships and connectors for Case Study 3:

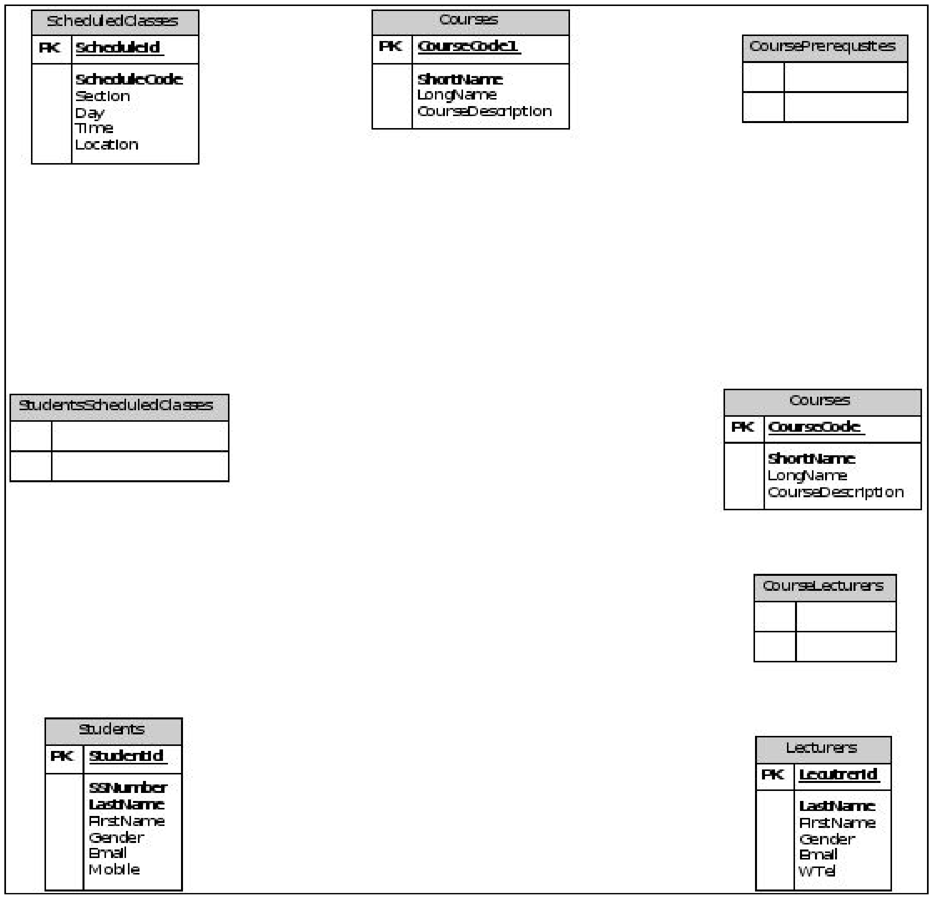

Figure 7.15 – New Relations for Case Study 3

Note: In order to clearly represent the many-to-many relationship *prerequisites*, the *Courses* Relation appears twice, with the primary key of one Relation slightly modified to differentiate between the two.

Third:

The diagram below shows the new Relations with new, new primary key, and foreign keys for Case Study 3:

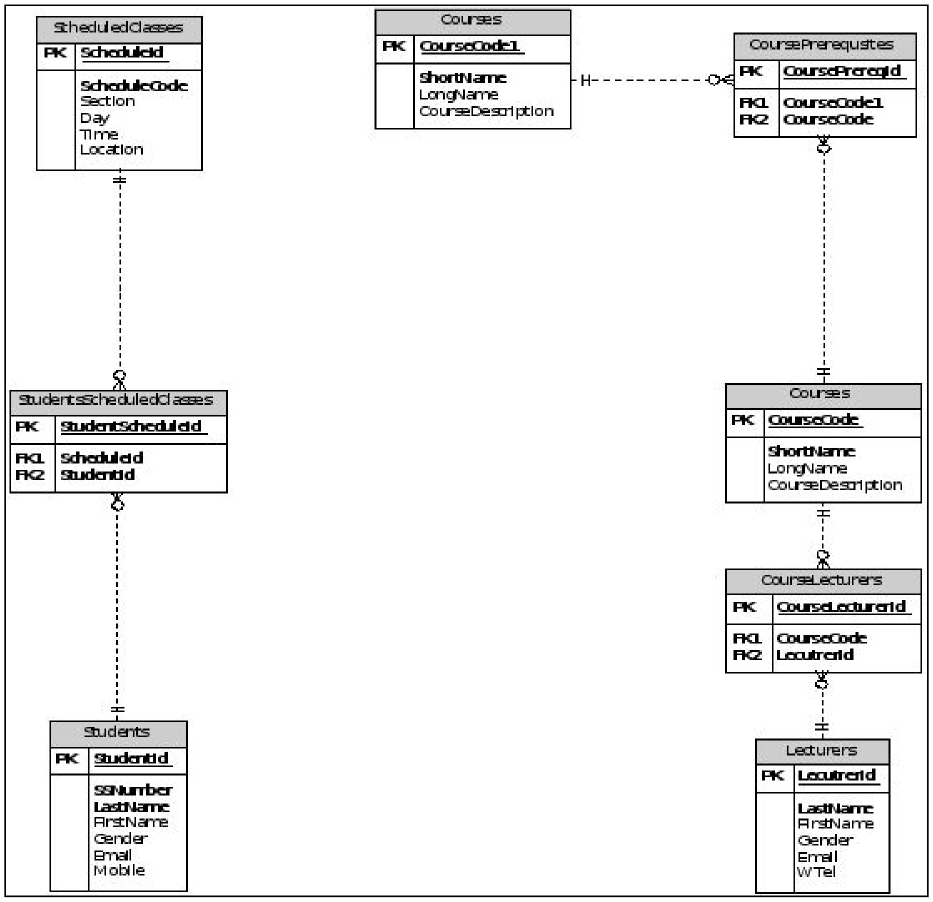

Figure 7.16 – New Relations and relationships for Case Study 3

Note: In the new Relation *CoursePrerequisites*, there are two foreign keys, but each foreign key references a primary key with the same name from the same Relation. One foreign key represents the course and the other represents the prerequisite course for that course.

Fourth:

The diagram below shows the completed many-to-many transformation for Case Study 3, including added attributes:

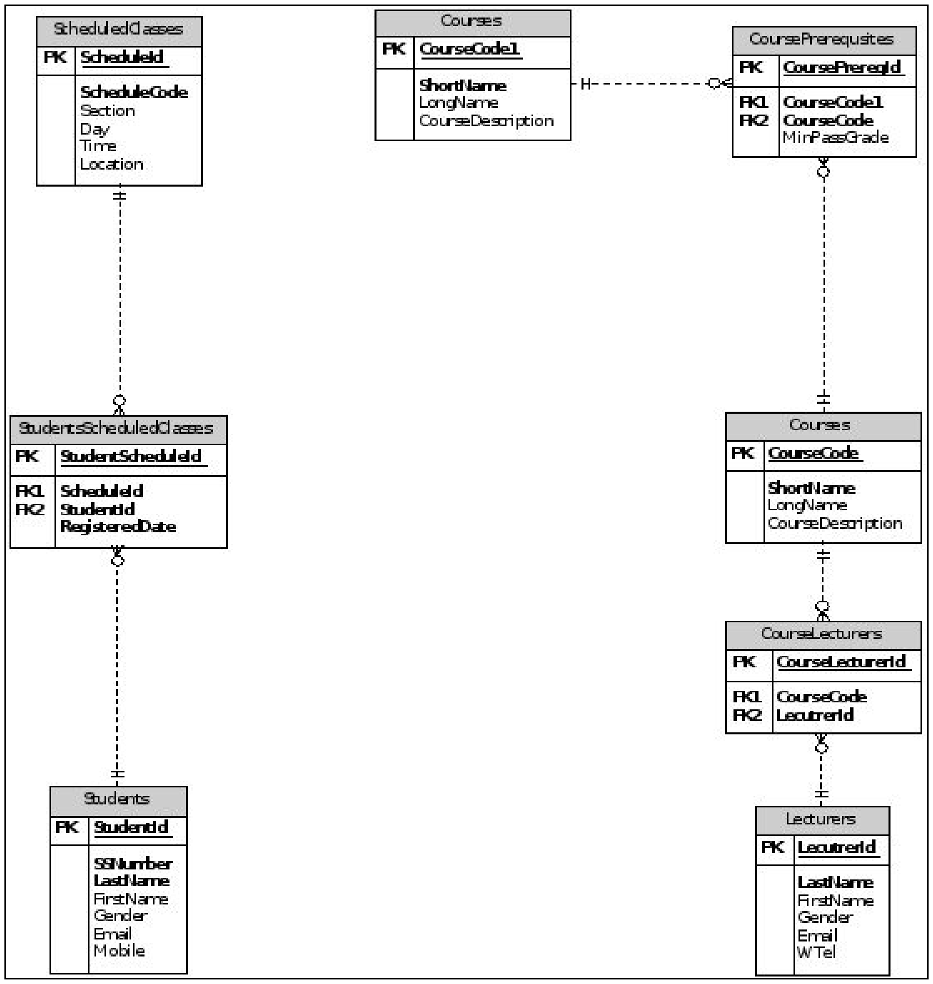

Figure 7.17 – Completed many-to-many transformation for Case Study 3

Remember: Verify that each new Relation contains two foreign keys. Also check to see whether there are possible attributes for the new Relations, and if there are, ensure that these new attributes depend on both foreign keys of the new Relation and not on any one foreign key individually.

Step 6-2: One-to-many relationships

This step will transform one-to-many relationships on the detailed E-R diagram into one-to-many relationships on the R-M diagram. One-to-many relationships on the detailed E-R diagram are identified by a cardinality of *N (many or at least one)* on one side of the relationship (diamond) and a cardinality of *1* (*only one*) on one side, as shown in the example below:

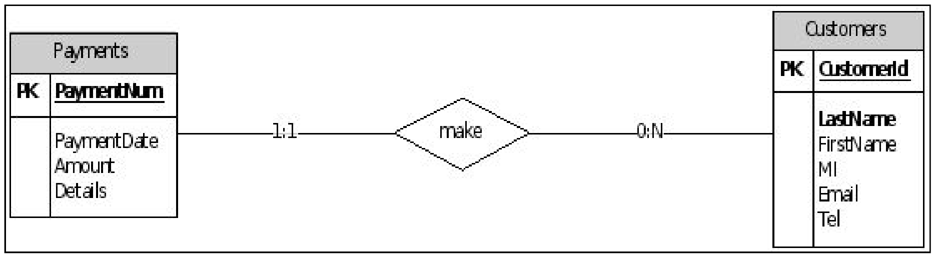

Figure 7.18 – One-to-many relationship on detailed E-R diagram

Here is what to do to transform these relationships into one-to-many relationships on an R-M diagram:

First:

Identify all one-to-many relationships. These relationships are identifiable by having a cardinality of *N (many or at least one)* on one side of the relationship (diamond) and a cardinality of *1 (only one)* on one side, as shown in figure 7.18.

Second:

Group together the Relation and relationship on the side of the relationship where the cardinality is *1 (only one)*, as shown in figure 7.19:

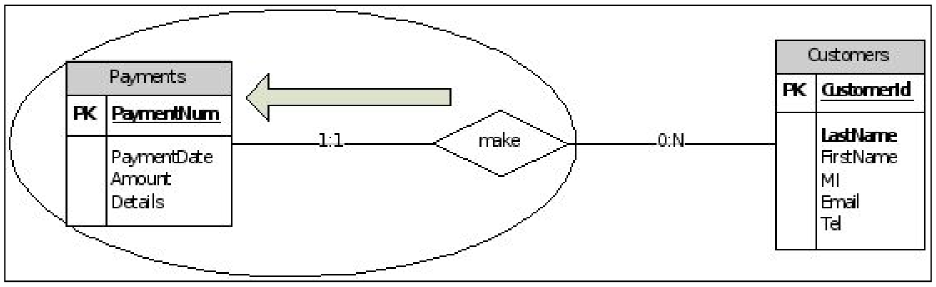

Figure 7.19 – One-to-many relationship grouping on detailed E-R diagram

This Relation will *absorb* the old relationship and create a new one-to-many relationship in its place.

Third:

Remove both the relationship and the connectors to that relationship, and create a new *1 to 0 or more*, or *1 to 1 or more*, relationship that connects the two Relations. Use a *1 to 0 or more* relationship if the optionality on the *N (many)* side of the old relationship was *0 (can)* and a *1 to 1 or more* if the optionality was *1 (must)*.

The many side (Crow's Foot) of the new relationship should be on the Relation that *absorbed* the old relationship, and the primary key of the *non-absorbing* Relation becomes a foreign key in the *absorbing* Relation.

Here is the resulting representation of the one-to-many relationship on the Crow's Foot Relational Model diagram:

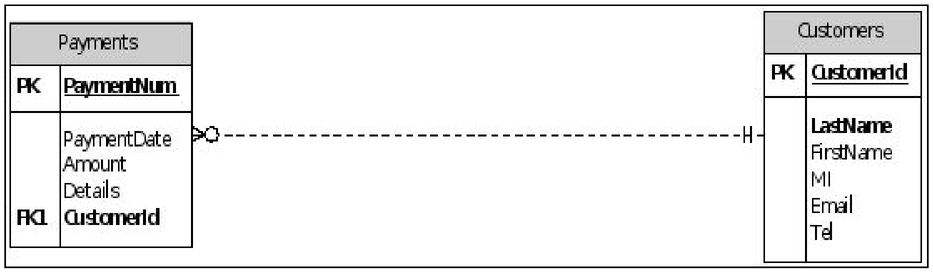

Figure 7.20 – One-to-many relationship on the Crow's Foot R-M diagram

Case Study 1

First:

The diagram below shows the one-to-many relationships for Case Study 1:

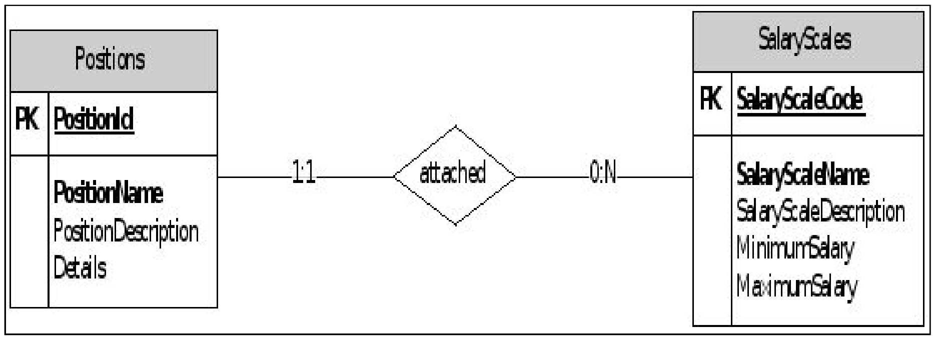

Figure 7.21 – One-to-many relationships for Case Study 1

Second:

The diagram below shows the grouping together of the Relations and relationships on the side of the relationship where the cardinality is *1 (only one)* for Case Study 1:

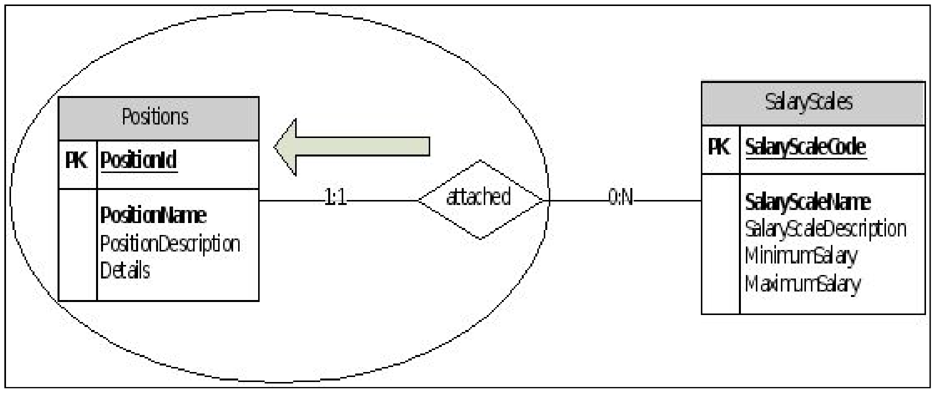

Figure 7.22 – Grouping together of Relations and relationships for Case Study 1

Third:

The diagram below shows the resulting Relations and relationships after the old relationship has been *absorbed* and replaced with a new one for Case Study 1:

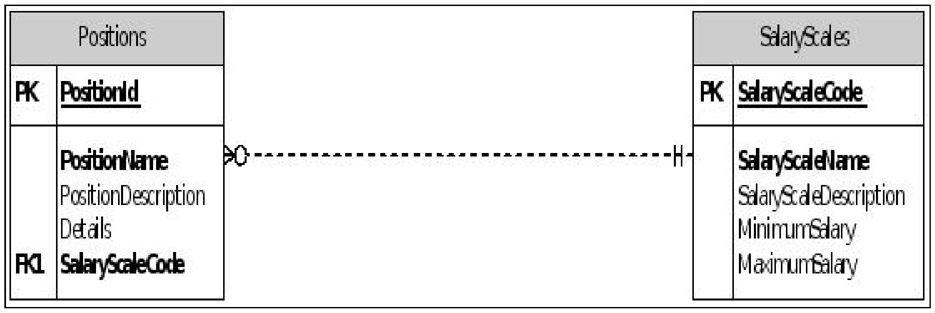

Figure 7.23 – Resulting Relations and relationships for Case Study 1

Remember: Verify that the foreign key and Crow's Foot are in the correct location – the foreign key is in the Relation that absorbed the old relationship, and the Crow's Foot is on that Relation also.

Case Study 2

First:

The diagram below shows the one-to-many relationships for Case Study 2:

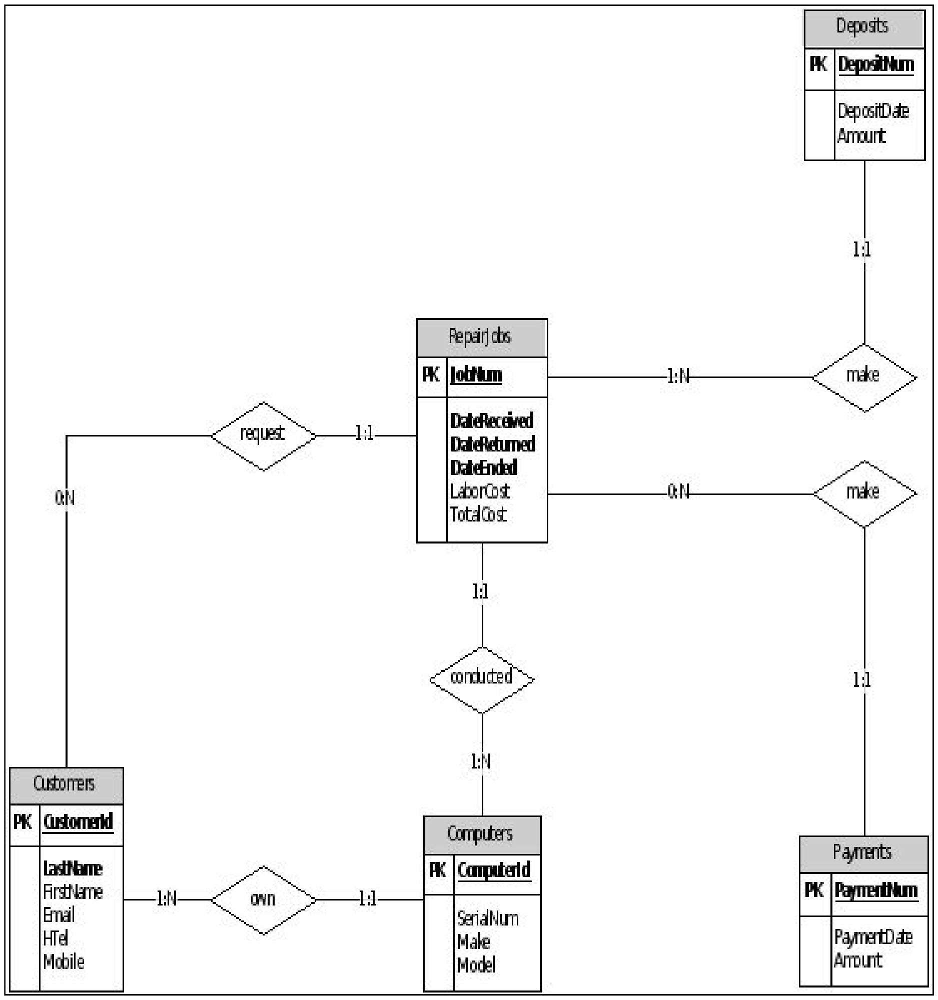

Figure 7.24 – One-to-many relationships for Case Study 2

Second:

The diagram below shows the grouping together of the Relations and relationships on the side of the relationship where the cardinality is *1 (only one)* for Case Study 2:

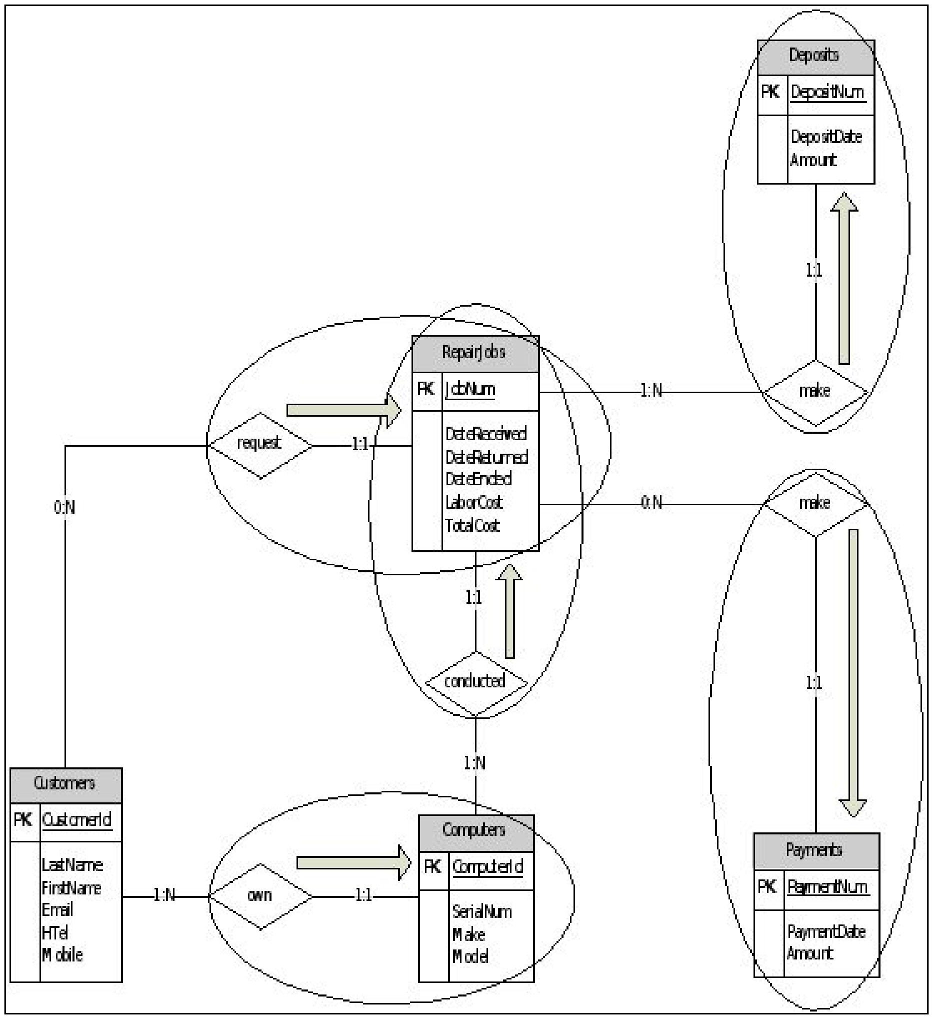

Figure 7.25 – Grouping together of Relations and relationships for Case Study 2

Third:

The diagram below shows the resulting Relations and relationships after the old relationship has been *absorbed* and replaced with a new one for Case Study 2:

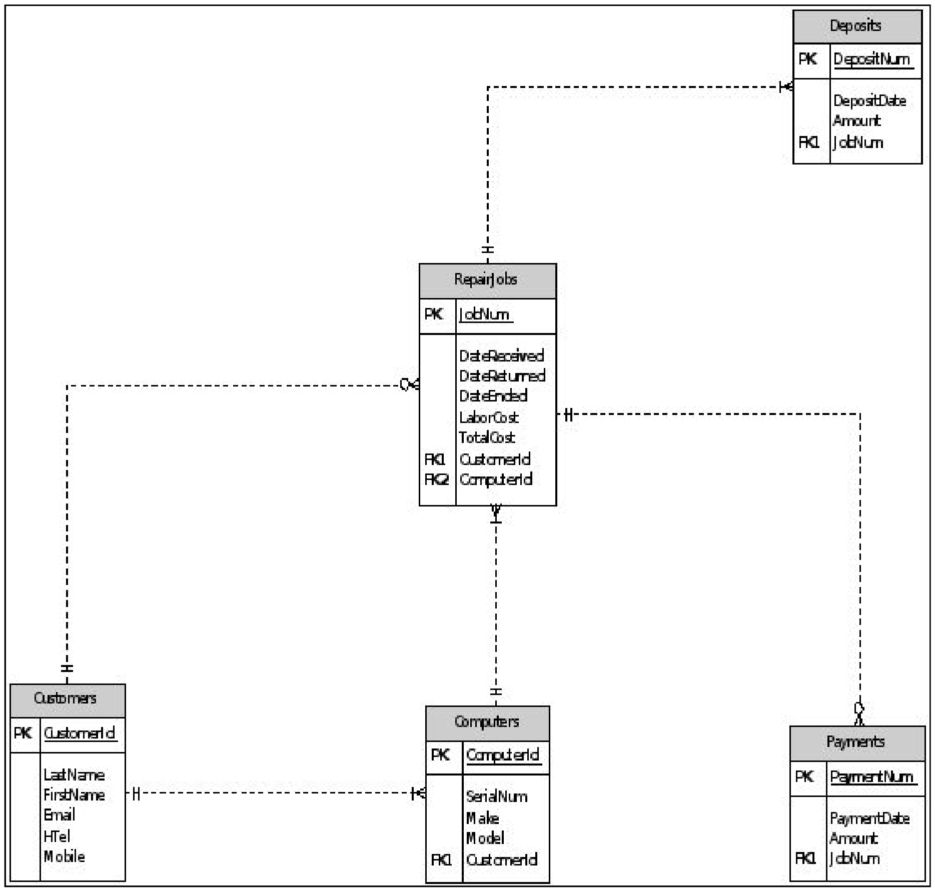

Figure 7.26 – Resulting Relations and relationships for Case Study 2

Remember: Verify that the foreign key and Crow's Foot are in the correct location – the foreign key is in the Relation that absorbed the old relationship, and the Crow's Foot is on that Relation also.

Case Study 3

First:

The diagram below shows the one-to-many relationships for Case Study 3:

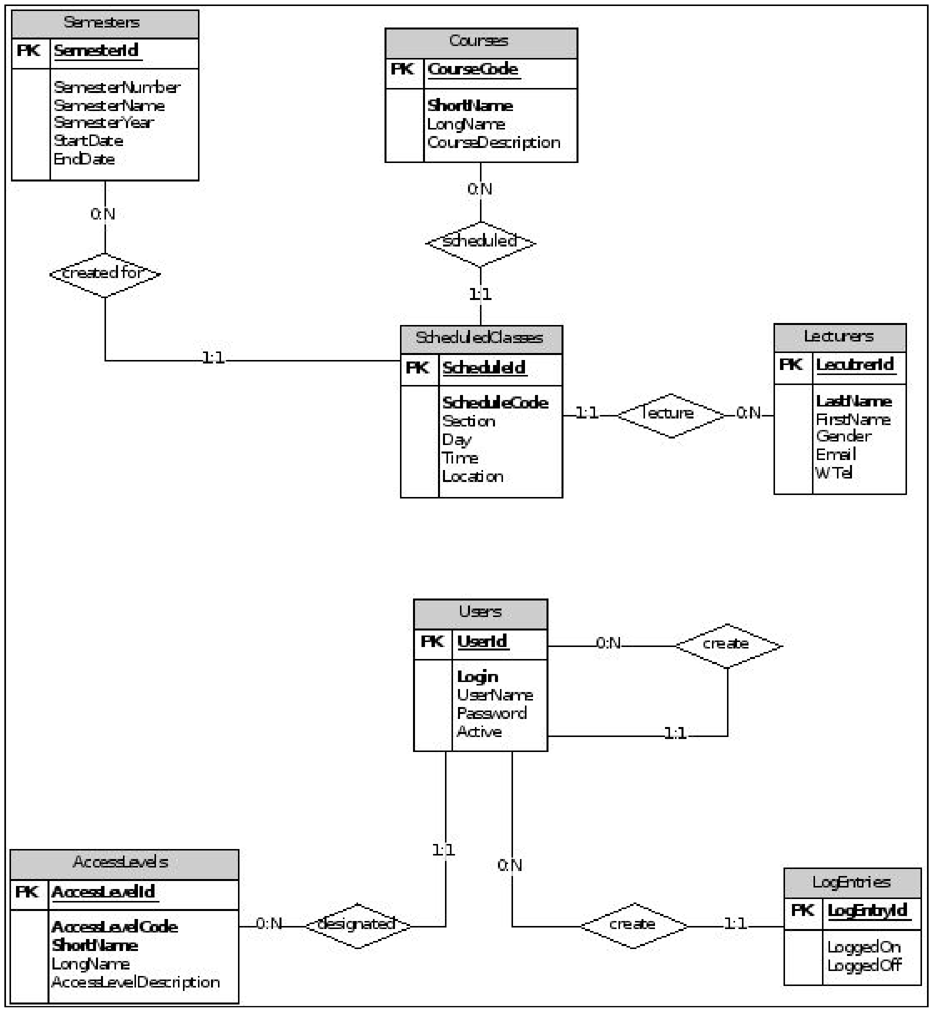

Figure 7.27 – One-to-many relationships for Case Study 3

Second:

The diagram below shows the grouping together of the Relations and relationships on the side of the relationship where the cardinality is *1 (only one)* for Case Study 3:

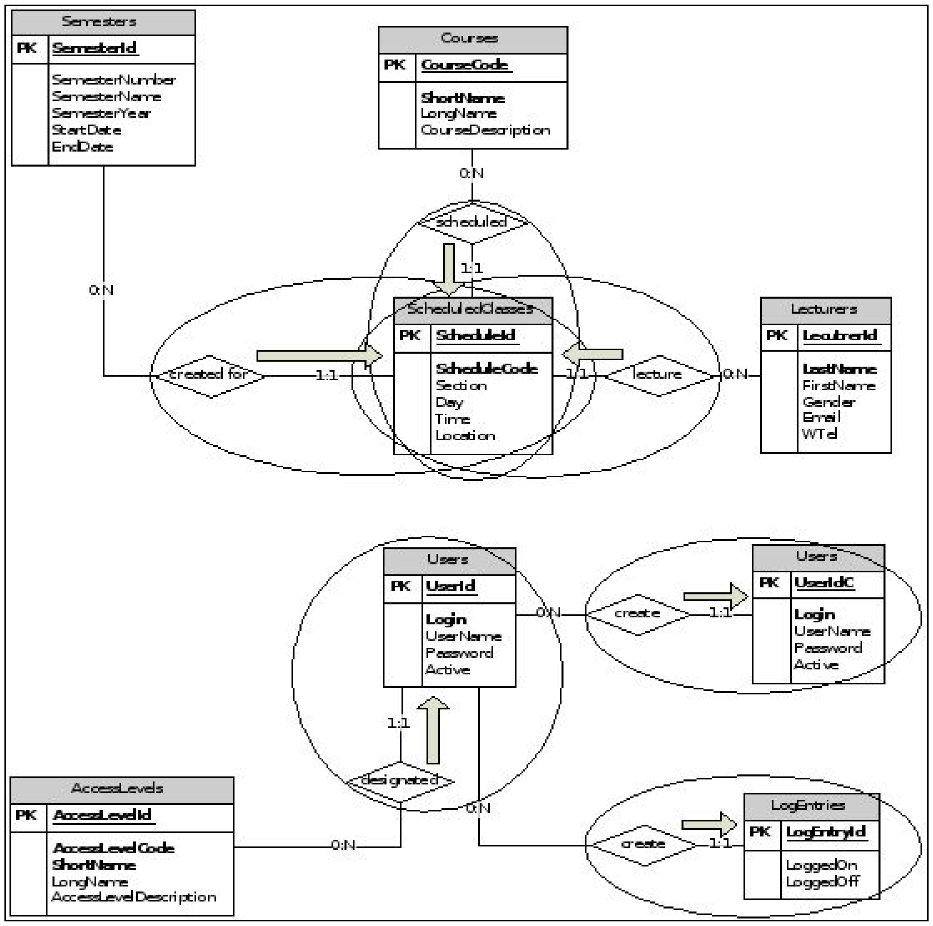

Figure 7.28 – Grouping together of Relations and relationships for Case Study 3

Note: In order to clearly represent the unary relationship *create*, the *Users* Relation appears twice, with the primary key of one Relation slightly modified to differentiate between the two.

Third:

The diagram below shows the resulting Relations and relationships after the old relationship has been *absorbed* and replaced with a new one for Case Study 3:

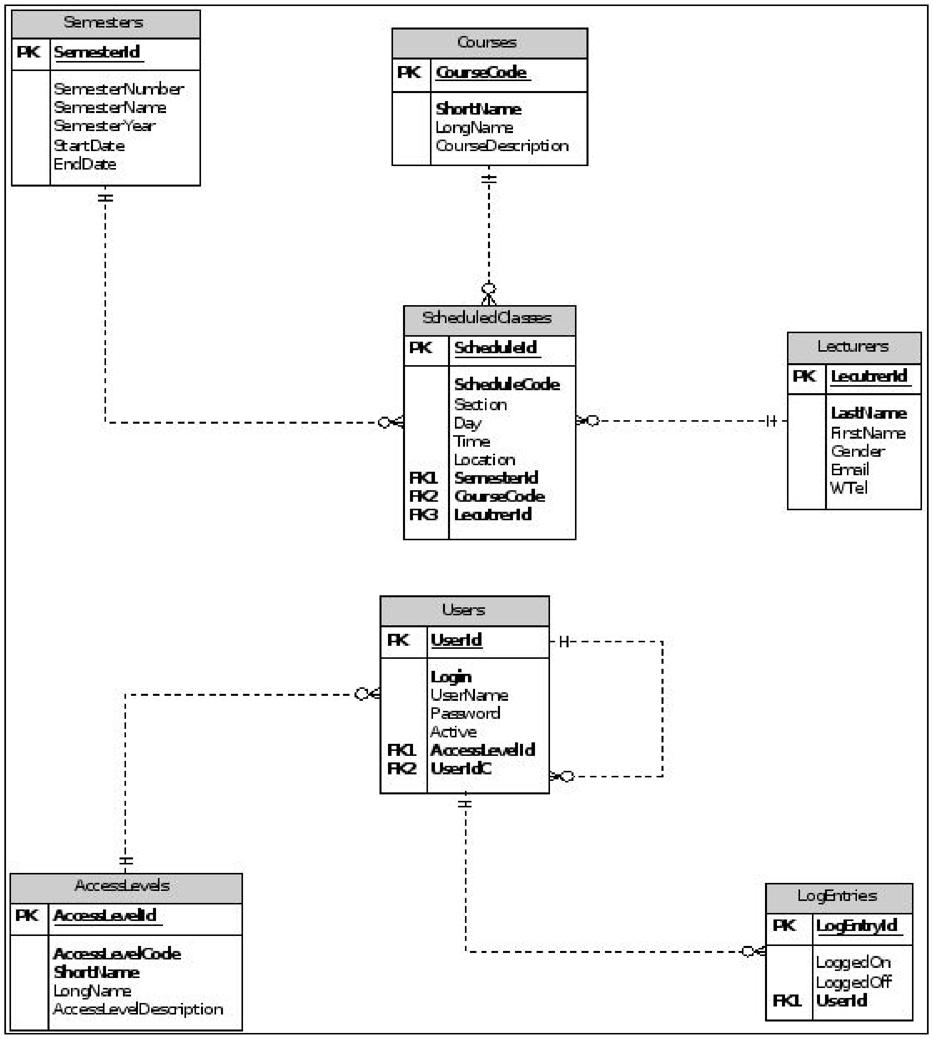

Figure 7.29 – Resulting Relations and relationships for Case Study 3

Remember: Verify that the foreign key and Crow's Foot are in the correct location – the foreign key is in the Relation that absorbed the old relationship, and the Crow's Foot is on that Relation also.

Step 6-3: One-to-one relationships

This step will transform one-to-one relationships on the detailed E-R diagram into one-to-one relationships on the R-M diagram. One-to-one relationships in the E-R diagram are identified by a cardinality of *1 (only one)* on *both* sides of the relationship (diamond), as shown in the example below:

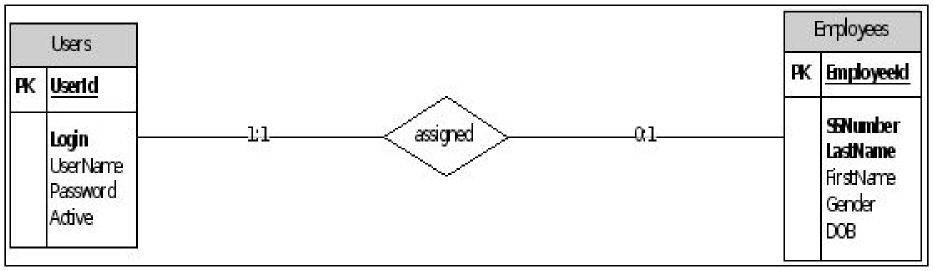

Figure 7.30 – One-to-one relationship on detailed E-R diagram

Here is what to do to transform these relationships into one-to-one relationships on an R-M diagram:

First:

Identify all one-to-one relationships. These relationships are identifiable by having a cardinality of *1 (only one)* on *both* sides of the relationship, as shown in figure 7.30.

Second:

Group together the Relation and relationship on the side of the relationship where the optionality is *1 (must)*, as shown in figure 7.31:

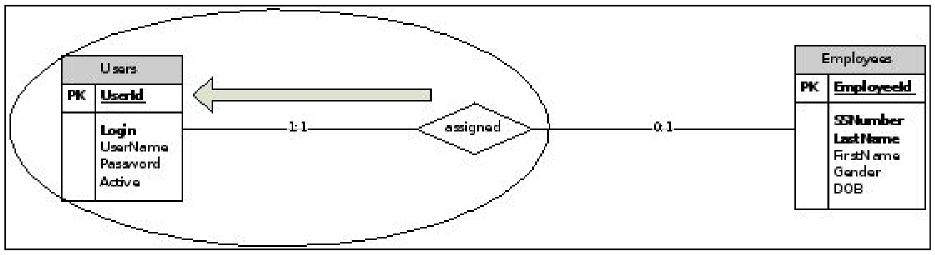

Figure 7.31 – One-to-one relationship grouping on detailed E-R diagram

This Relation will *absorb* the old relationship and create a new one-to-one relationship in its place.

In most cases, one side of the old relationship will have an optionality of *1 (must),* and the other an optionality of *0 (can)*. If both sides have an optionality of *1 (must)* then the old relationship can be absorbed by either Relation. Which Relation absorbs the old relationship should be based on which would be more efficient to implement in the RDBMS. Chapter 9 discusses this further.

Third:

Remove both the relationship and the connectors to that relationship, and create a new *1 to 1* relationship that connects the two Relations.

The primary key of the *non-absorbing* Relation becomes a foreign key in the *absorbing* Relation.

Here is the resulting representation of the one-to-one relationship on the Crow’s Foot Relational Model diagram:

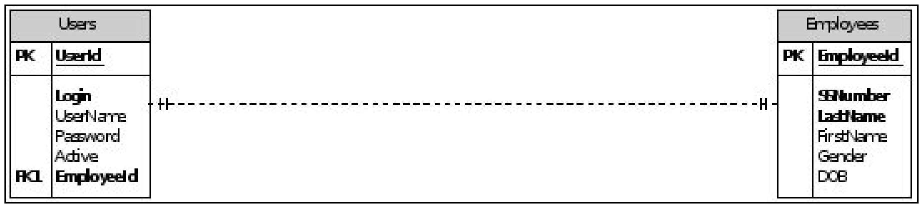

Figure 7.32 – One-to-one relationship on the Crow’s Foot R-M diagram

Case Study 1

First:

The diagram below shows the one-to-one relationships for Case Study 1:

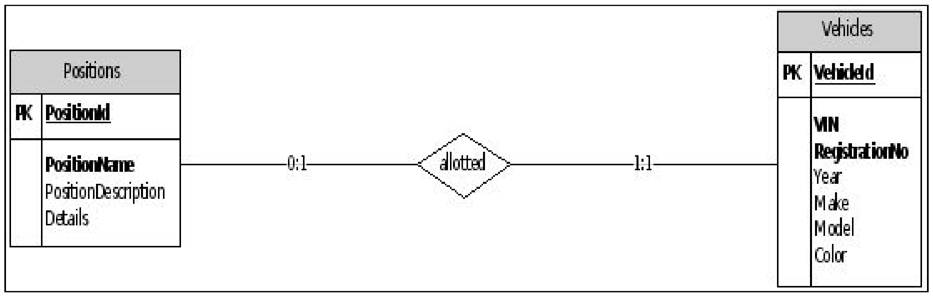

Figure 7.33 – One-to-one relationships for Case Study 1

Second:

The diagram below shows the grouping together of the Relations and relationships on the side of the relationship where the optionality is *1 (must)* for Case Study 1:

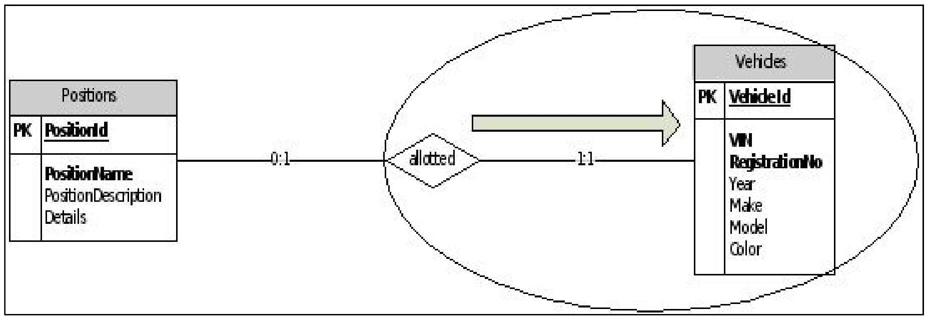

Figure 7.34 – Grouping together of Relations and relationships for Case Study 1

Third:

The diagram overleaf shows the resulting Relations and relationships after the old relationship has been *absorbed* and replaced with a new one for Case Study 1:

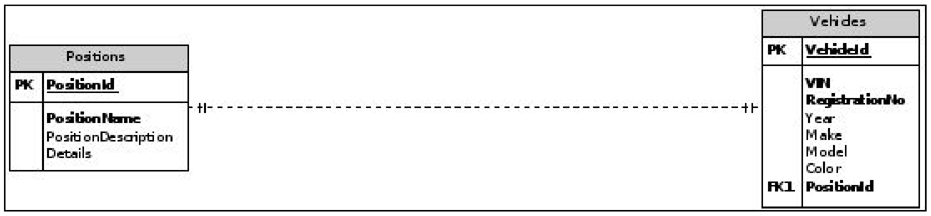

Figure 7.35 – Resulting Relations and relationships for Case Study 1

Remember: Verify that the foreign key is in the correct Relation – the Relation that absorbed the old relationship.

Case Study 2

There are no one-to-one relationships for Case Study 2.

Case Study 3

First:

The diagram below shows the one-to-one relationships for Case Study 3:

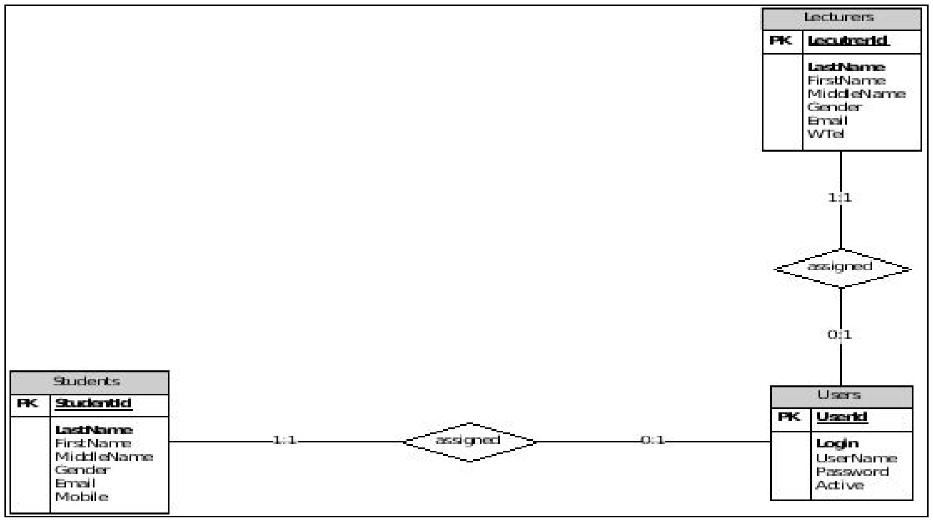

Figure 7.36 – One-to-one relationships for Case Study 3

Second:

The diagram below shows the grouping together of the Relations and relationships on the side of the relationship where the optionality is *1 (must)* for Case Study 3:

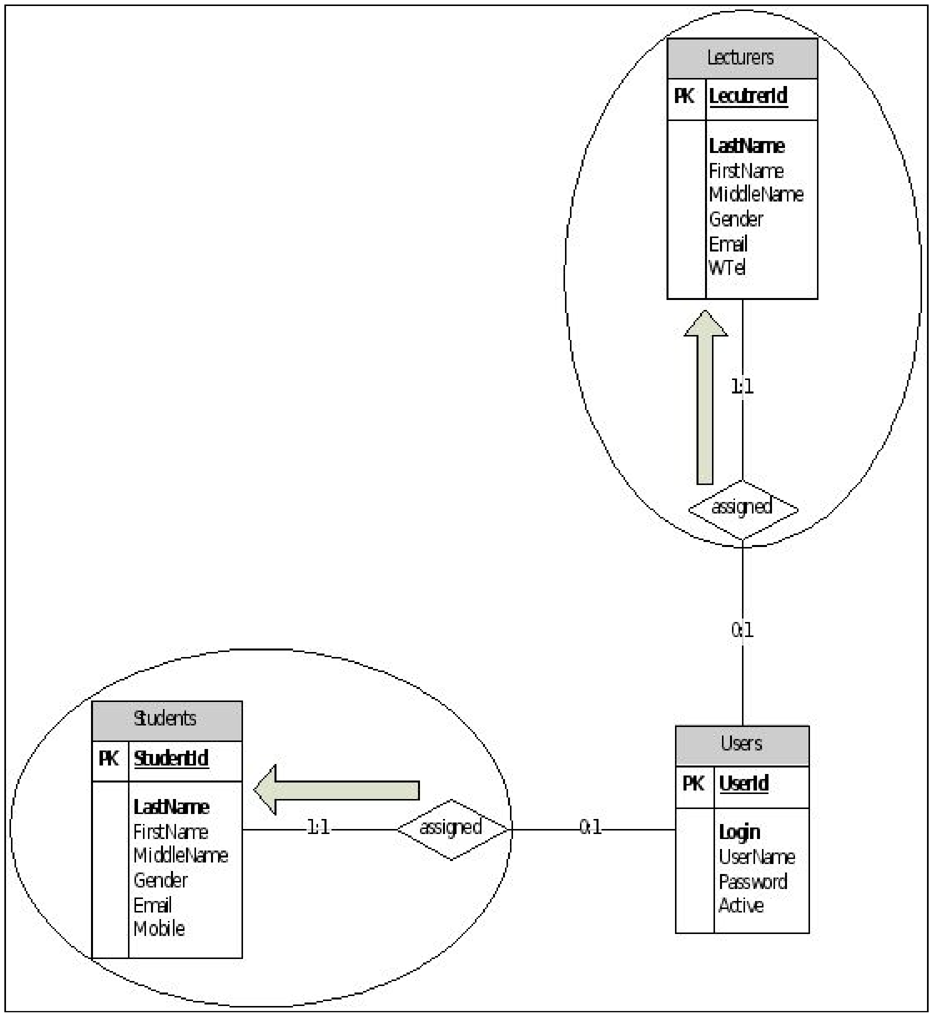

Figure 7.37 – Grouping together of Relations and relationships for Case Study 3

Third:

The diagram below shows the resulting Relations and relationships after the old relationship has been *absorbed* and replaced with a new one for Case Study 3:

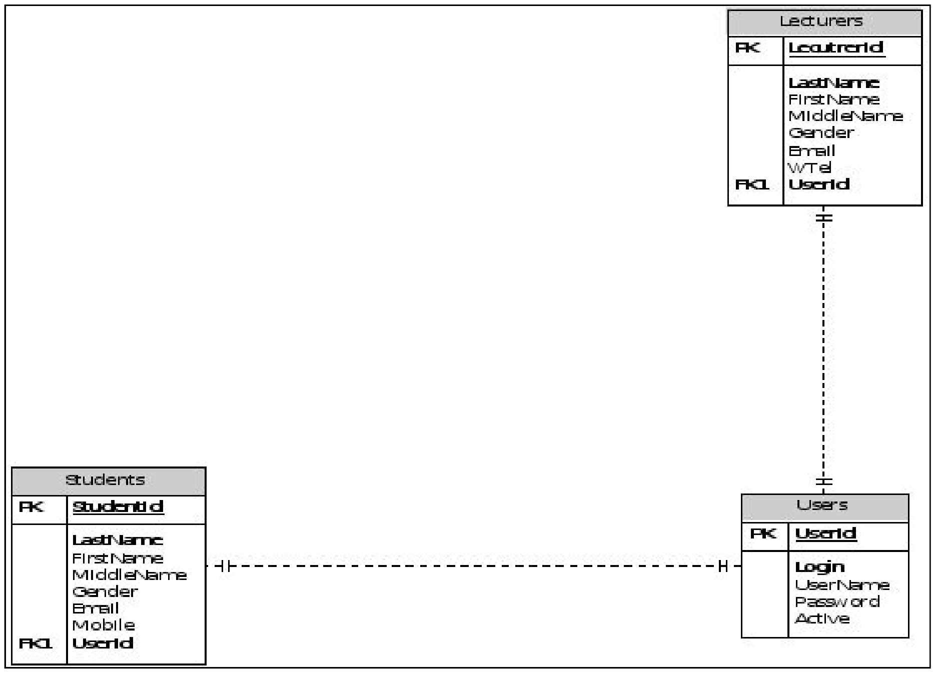

Figure 7.38 – Resulting Relations and relationships for Case Study 3

Remember: Verify that the foreign key is in the correct Relation – the Relation that absorbed the old relationship.

Step 6-4: Combine results to create R-M diagram

Combining the diagrams created in Steps 6-1 to 6-3 creates the final Crow’s Foot Relational Model diagram. Depending on the nature and complexity of the problem, more than one diagram may be required to represent the database design.

Here are the resulting R-M diagrams for each of the case studies.

Case Study 1

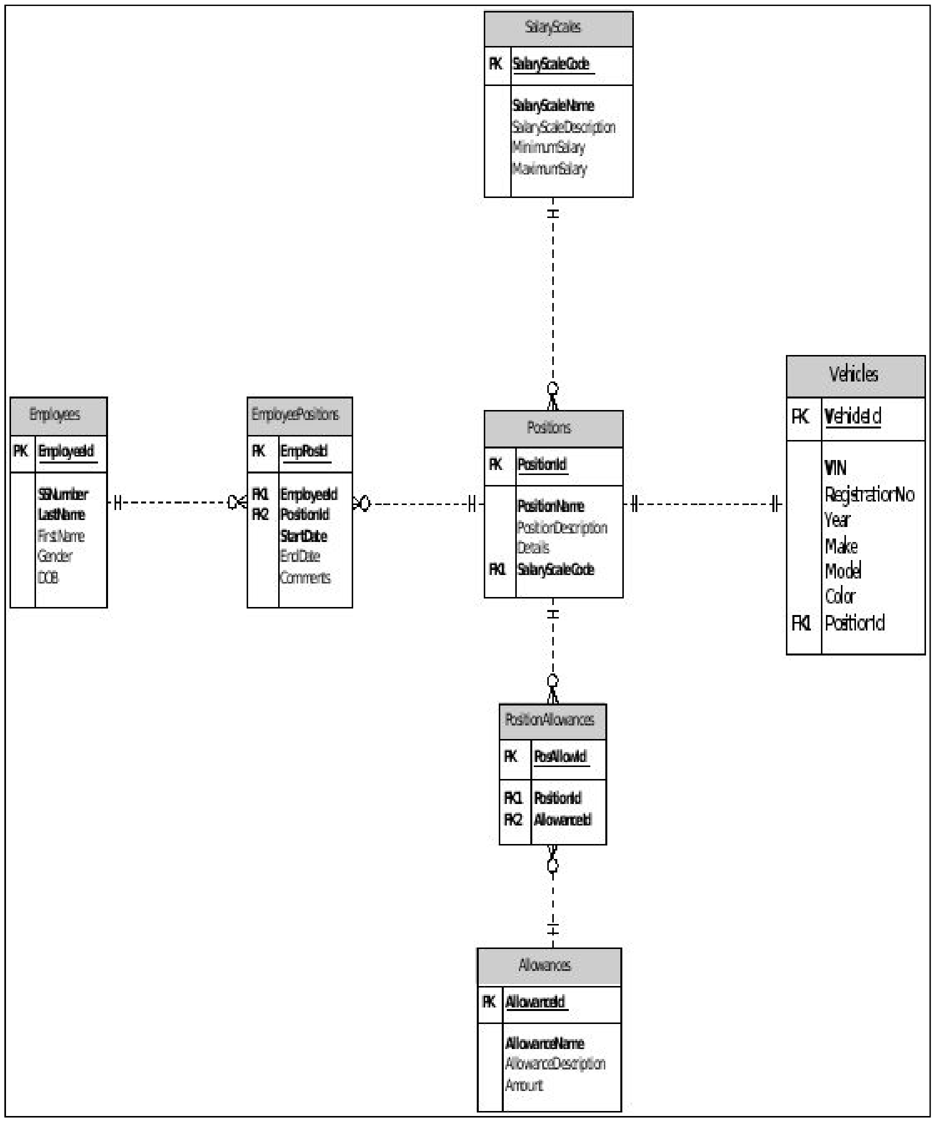

Figure 7.39 – Final Crow’s Foot R-M diagram for Case Study 1

Case Study 2

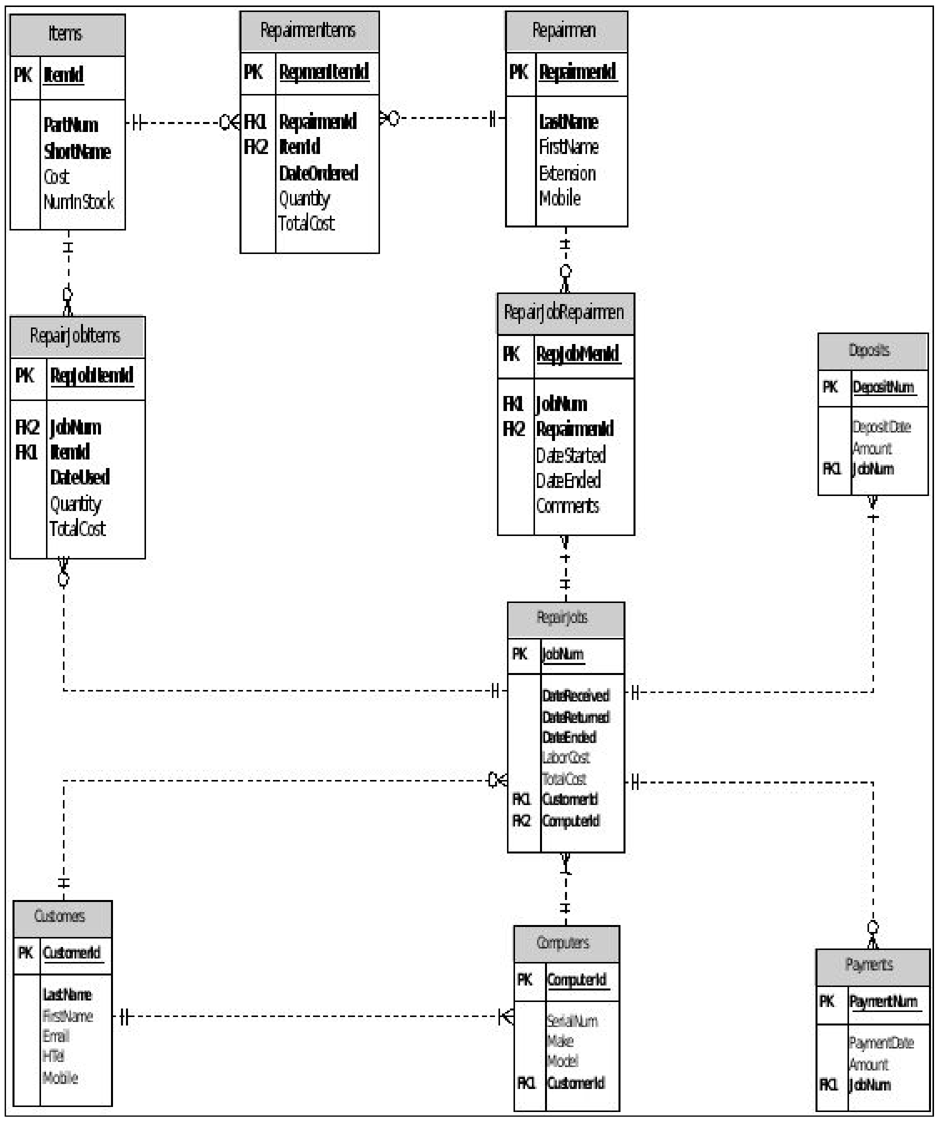

Figure 7.40 – Final Crow’s Foot R-M diagram for Case Study 2

Case Study 3

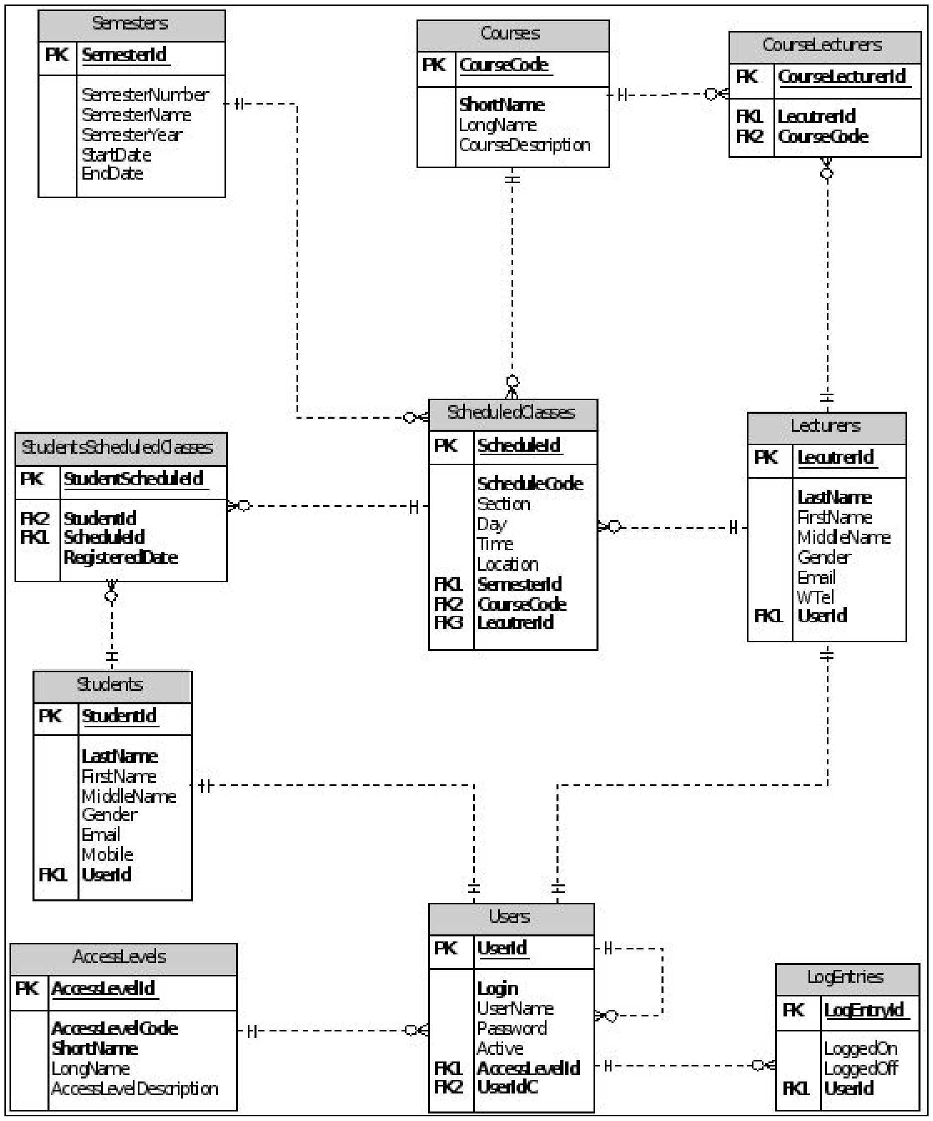

Figure 7.41 – Final Crow’s Foot R-M diagram for Case Study 3 (Part 1)

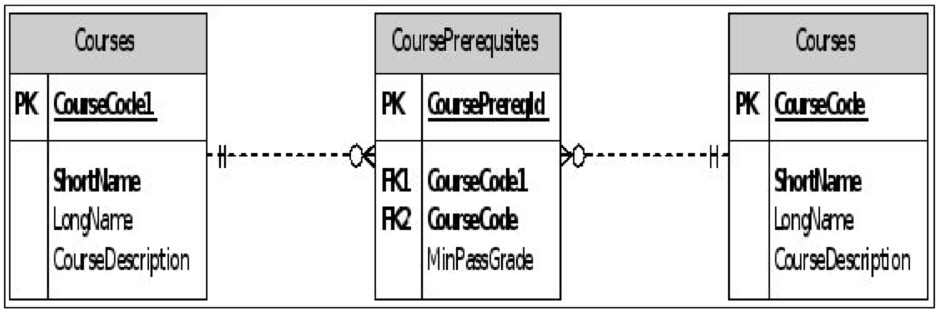

Figure 7.42 – Final Crow’s Foot R-M diagram for Case Study 3 (Part 2)

Summary

This chapter focused on how to transform the detailed E-R diagram, created in the previous step, into a Crow's Foot Relational Model diagram that can be easily implemented on any RDBMS. Here is a list of steps to transform the E-R diagram into an R-M diagram:

Step 6-1: Many-to-many relationships

This step will transform many-to-many relationships on the detailed E-R diagram into many-to-many relationships on the R-M diagram. Many-to-many relationships in the E-R diagram are identified by a cardinality of *N (many or at least one)* on both sides of the relationship (diamond).

First: Identify all many-to-many relationships. These relationships are identified by a cardinality of *N (many/at least one)* on both sides of the relationship.

Second: Remove both the relationship and the connectors to that relationship. Replace the relationship with a new Relation. The name of this new Relation should be a combination of the names of the two Relations that are on either side of the removed relationship.

Third: Create new *1 to 0 or more*, or *1 to 1 or more* relationships that connect the two existing Relations to the new Relation. Use a *1 to 0 or more* relationship if the optionality of the old relationship was *0 (can)*, and a *1 to 1 or more* if the optionality was *1 (must)*. The many side (Crow's Foot) of the new relationship should be on the new Relation. Ensure that the primary key for each of the two existing Relations becomes a foreign key in the new Relation, and create a new and separate primary key for the new Relation, such as a unique identifier (Id).

Fourth: Explore whether attributes exist for the newly created Relation. In most cases they will not, but in some cases they will. These attributes must be valid for both foreign keys and not for any one foreign key individually; otherwise that attribute belongs in one of the relations for the foreign keys and not in this new relation.

Remember: Verify that each new Relation contains two foreign keys. Also check to see whether there are possible attributes for the new Relations, and if there are, ensure that these new attributes depend on both foreign keys of the new Relation and not on any one foreign key individually.

Step 6-2: One-to-many relationships

This step will transform one-to-many relationships on the detailed E-R diagram into one-to-many relationships on the R-M diagram. One-to-many relationships on the E-R diagram are identified by a cardinality of *N (many or at least one)* on one side of the relationship (diamond) and a cardinality of *1 (only one)* on one side.

First: Identify all one-to-many relationships. These relationships are identified by a cardinality of *N (many or at least one)* on one side of the relationship (diamond) and a cardinality of *1 (only one)* on one side.

Second: Group together the Relation and relationship on the side of the relationship where the cardinality is *1 (only one)*.

Third: Remove both the relationship and the connectors to that relationship, and create a new *1 to 0 or more*, or *1 to 1 or more*, relationship that connects the two Relations. Use a *1 to 0 or more* relationship if the optionality on the *N (many)* side of the old relationship was *0 (can)*, and a *1 to 1 or more* if the optionality was *1 (must)*. The many side (Crow’s Foot) of the new relationship should be on the Relation that *absorbed* the old relationship, and the primary key of the *non-absorbing* Relation becomes a foreign key in the *absorbing* Relation.

Remember: Verify that the foreign key and Crow’s Foot are in the correct location – the foreign key is in the Relation that absorbed the old relationship, and the Crow’s Foot is on that Relation also.

Step 6-3: One-to-one relationships

This step will transform one-to-one relationships on the detailed E-R diagram into one-to-one relationships on the R-M diagram. One-to-one relationships in the E-R diagram are identified by a cardinality of *1 (only one)* on *both* sides of the relationship (diamond).

First: Identify all one-to-one relationships. These relationships are identifiable by having a cardinality of *1 (only one)* on *both* sides of the relationship (diamond).

Second: Group together the Relation and relationship on the side of the relationship where the optionality is *1 (must)*. In most cases, one side of the old relationship will have an optionality of *1 (must)* and the other an optionality of *0 (can)*. If both sides have an optionality of *1 (must)*, then the old relationship can be absorbed by either Relation.

Third: Remove both the relationship and the connectors to that relationship, and create a new *1 to 1* relationship that connects the two Relations. The primary key of the *non-absorbing* Relation becomes a foreign key in the *absorbing* Relation.

Remember: Verify that the foreign key is in the correct Relation – the Relation that absorbed the old relationship.

Step 6-4: Combine results to create R-M diagram

Combining the diagrams created in Step 6-1 to Step 6-3 creates the final Crow's Foot Relational Model diagram. Depending on the nature and complexity of the problem, more than one diagram may be required to represent the database design.

Review Questions

1. State the two types of models in relational database design, and give an example of each type of model.

2. What two types of relationships are Crow’s feet used to represent?

3. How are many-to-many relationships created using Crow’s Foot notation?

4. State the steps necessary to convert a detailed E-R Diagram into a Crow’s Foot R-M Diagram.

5. Using appropriate examples, describe how the following relationships are represented on a detailed E-R Diagram:

    a. one-to-one
    b. one-to-many
    c. many-to-many

6. Using appropriate examples, describe how the following relationships are represented on a Crow’s Foot R-M Diagram:

    a. one-to-one
    b. one-to-many
    c. many-to-many

7. Using appropriate examples, describe how the following relationships on a detailed E-R Diagram are transformed into their corresponding counterparts on a Crow’s Foot R-M Diagram:

    a. one-to-one
    b. one-to-many
    c. many-to-many

Exercises

Consider the following detailed E-R diagram for an online insurance policy tracking system:

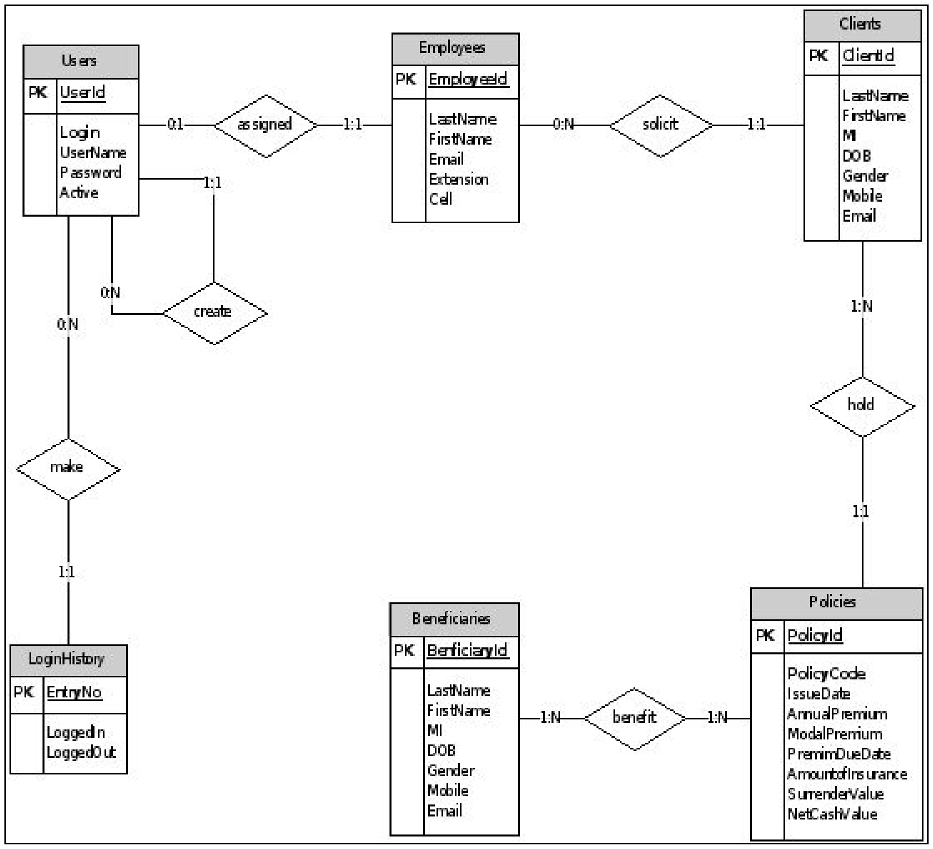

Figure 7.43 – Detailed E-R diagram for insurance policy tracking system

1. Transform the detailed E-R diagram in figure 7.43 above into an implementable R-M diagram.

Chapter 8

Case Studies

This chapter goes through the three case studies that have been used throughout this book, applying the six-step process to each of them. It eliminates the detailed explanations and sub-steps, applying the six-step process to each case study. It is intended to show the six-step process in action and can be used by experienced programmers, experienced database designers, experienced database developers, and students to get a quick synopsis of the six-step process.

Case Study 1

Problem Specification

A small accounting firm wants a simple HR application that will help it to keep track of its employees, their positions, allowances, salary scales, and which company vehicles their employees drive.

The application must keep track of all the positions at the firm, the employees filling these positions, the allowances for these positions, the salary scales for these positions, and the company vehicles assigned to these positions.

Step 1: Discover entities and assign attributes

Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
(PK) EmployeeId	**(PK) PositionId**	**(PK) AllowanceId**	**(PK) SalaryScaleCode**	**(PK) VehicleId**
SSNumber	**PositionName**	**AllowanceName**	**SalaryScaleName**	**VIN**
LastName	PositionDescription	AllowanceDescription	SalaryScaleDescription	**RegistrationNo**
FirstName	Details	Amount	MinimumSalary	Year
MiddleName			MaximumSalary	Make
Gender				Model
DOB				Color
Email				
Mobile				
HTel				
AddressLine1				
AddressLine2				
City				
State				
PostCode				

Step 2: Derive unary and binary relationships

	Employees	Positions	Allowances	SalaryScales	Vehicles
Employees					
Positions	fill				
Allowances		allocated			
SalaryScales		attached			
Vehicles		allotted			

Step 3: Create simplified Entity-Relationship diagram

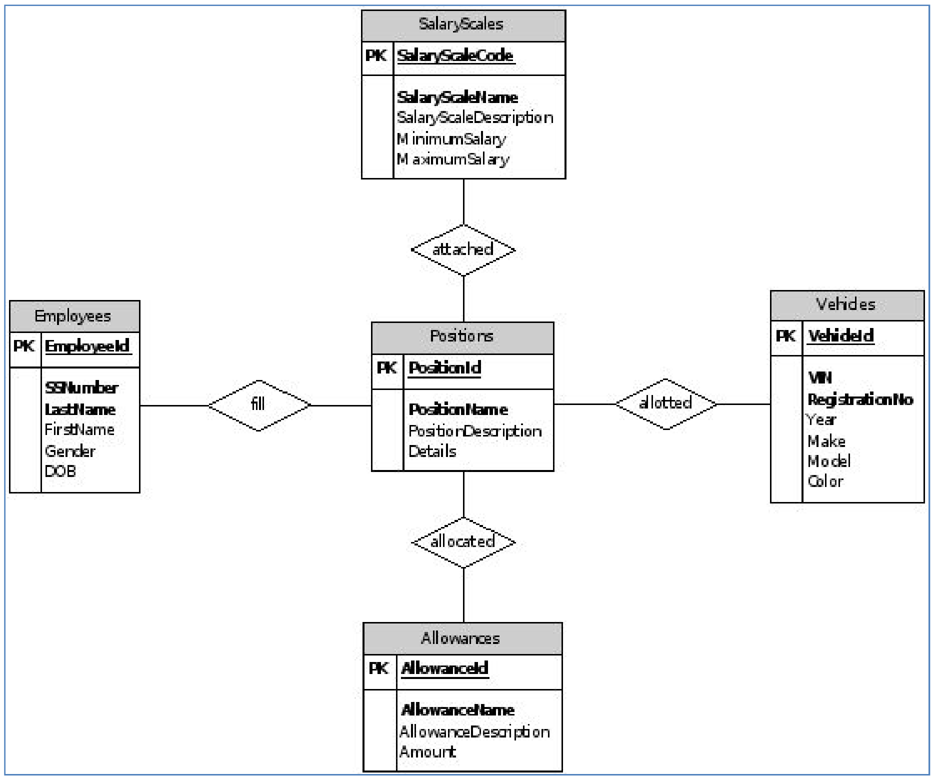

Figure 8.1 – Simplified E-R diagram for Case Study 1

Step 4: List assertions for all relationships

- A Position can be allocated many Allowances
- Each Allowance can be allocated to many Positions
- A Position can be allotted only one Vehicle
- Each Vehicle must be allotted to only one Position
- A Position must be attached to only one SalaryScale
- Each SalaryScale can be attached to many Positions
- A Position can be filled by many Employees
- Each Employee can fill many Positions

Step 5: Create detailed E-R diagram using assertions

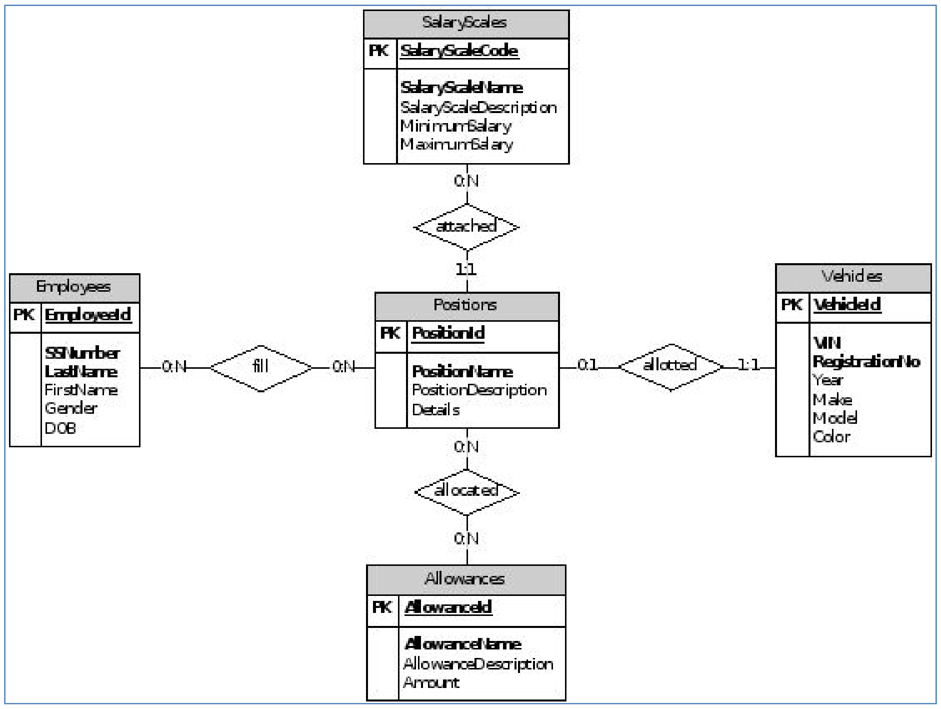

Figure 8.2 – Detailed E-R diagram for Case Study 1

Step 6: Transform the detailed E-R diagram into an implementable R-M diagram

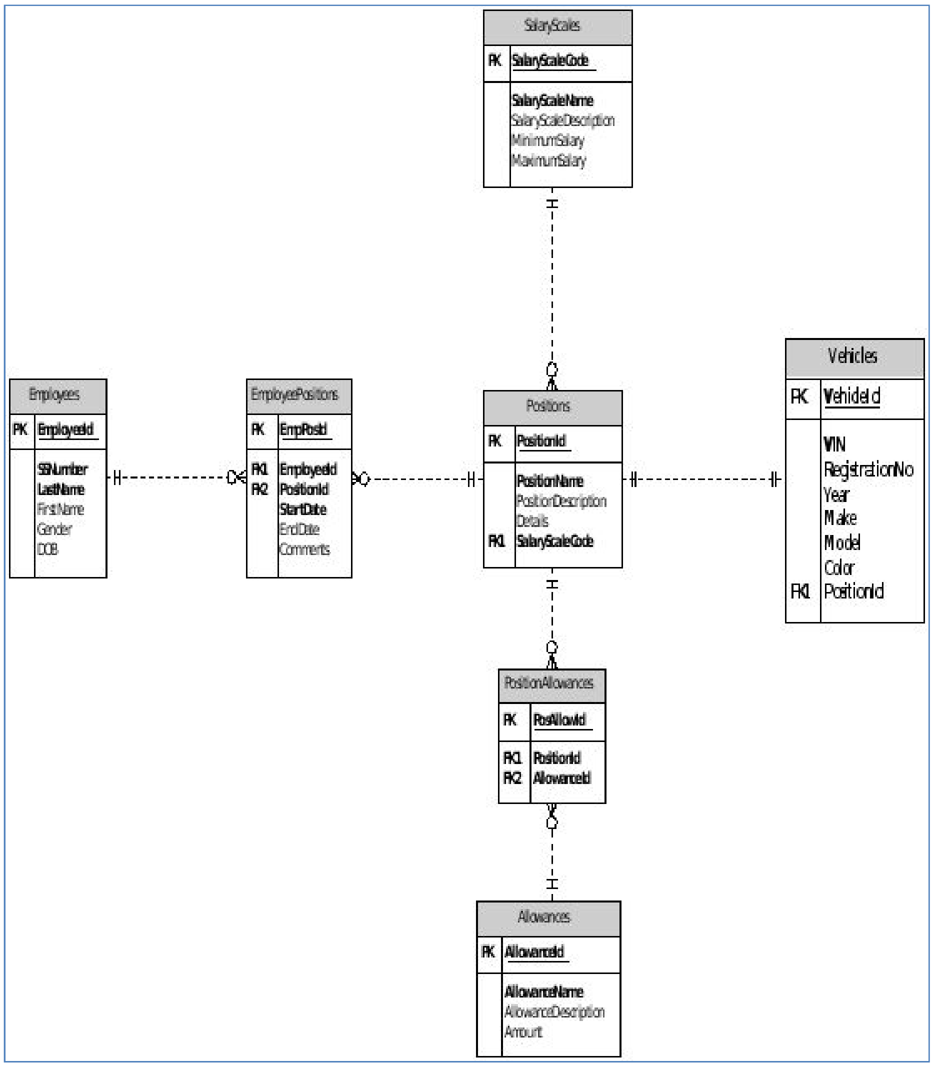

Figure 8.3 – Crow’s Foot R-M diagram for Case Study 1

Case Study 2

Problem Specification

The owners of a small computer repair shop would like to keep track of the repair jobs for computers they repair, the items used for each repair job, the labor costs for each repair job, the repairmen performing each repair job, and the total cost of each repair job.

When customers bring their computers in to be repaired, they make a deposit on the repair job and are given a date to return and uplift their computer. Repairmen then perform repairs on the customers' computers based on the repair job, and detail the labor costs and the items used for each repair job.

When customers return they pay the total cost of the repair job less the deposit, collect a receipt for their payment, and uplift the repaired computer using this payment receipt.

Step 1: Discover entities and assign attributes

RepairJob	**Computers**	**Items**	**Repairmen**	**Customers**
(PK) JobNum	**(PK) ComputerId**	**(PK) ItemId**	**(PK) RepairmenId**	**(PK) CustomerId**
DateReceived	**SerialNum**	**PartNum**	**LastName**	**LastName**
DateToReturn	Make	**ShortName**	FirstName	FirstName
DateReturned	Model	ItemDescription	MI	MI
DateStarted	ComputerDescription	Cost	Email	Email
DateEnded		NumInStock	Mobile	Mobile
RepairDetails		ReorderLow	HTel	HTel
LaborDetails			Extension	AddressLine1
LaborCost				AddressLine2
TotalCost				City
PaidInFull				State
AdditionalComments				**PostCode**

Deposits	**Payments**
(PK) DepositNum	**(PK) PaymentNum**
DepositDate	PaymentDate
Amount	Amount

Step 2: Derive unary and binary relationships

	RepairJobs	Computers	Items	Repairmen	Customers	Deposits	Payments
RepairJobs							
Computers	conducted						
Items	use						
Repairmen	perform		order				
Customers	request	own					
Deposits	make						
Payments	make						

Step 3: Create simplified Entity-Relationship diagram

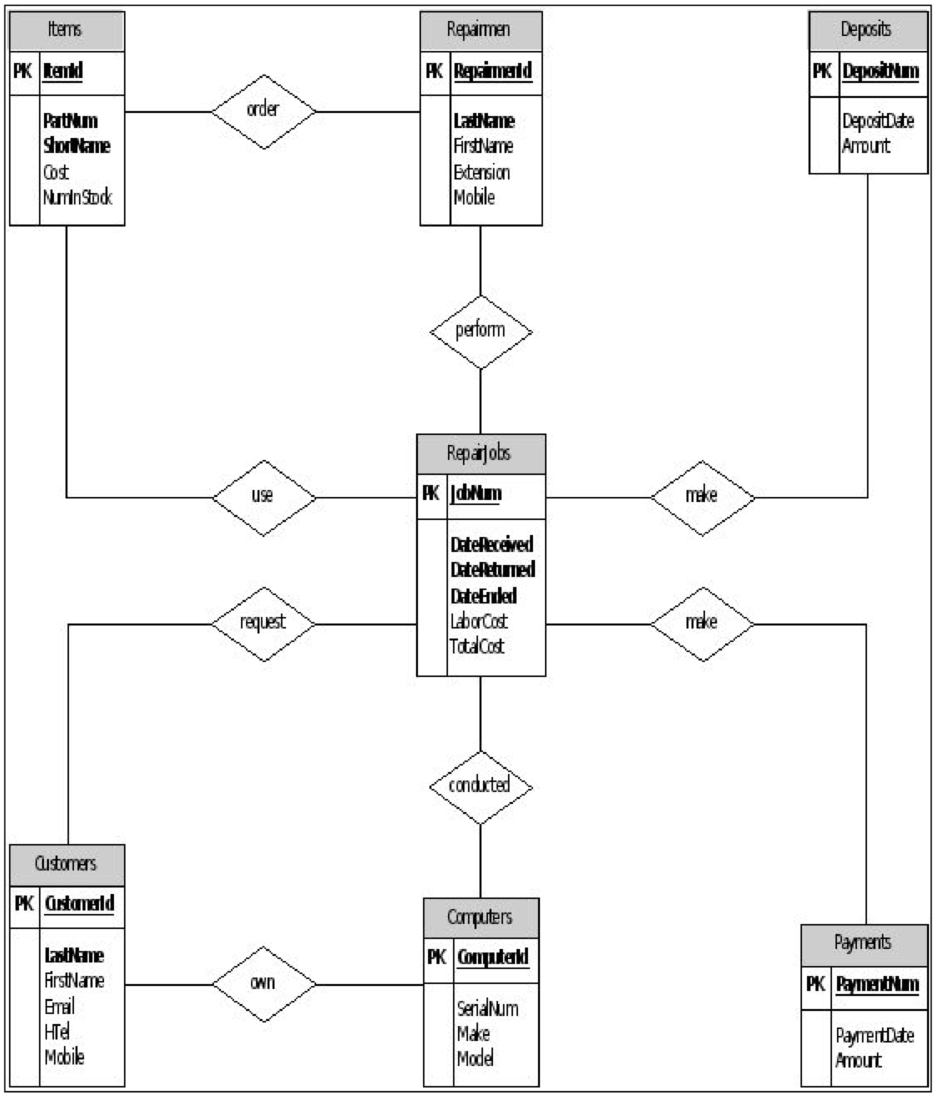

Figure 8.4 – Simplified E-R diagram for Case Study 2

Step 4: List assertions for all relationships

- A RepairJob must be conducted on only one Computer
- Each Computer must have conducted at least one RepairJob
- A RepairJob must be requested by only one Customer
- Each Customer can request many RepairJobs
- A RepairJob can use many Items
- Each Item can be used in many RepairJobs
- A RepairJob must be performed by at least one Repairman
- Each Repairman can perform many RepairJobs
- A RepairJob must have made at least one Deposit
- Each Deposit must be made for only one RepairJob
- A RepairJob can have made many Payments
- Each Payment must be made for only one RepairJob
- An Item can be ordered by many Repairmen
- Each Repairman can order many Items
- A Customer must own at least one Computer
- Each Computer must be owned by only one Customer

Step 5: Create detailed E-R diagram using assertions

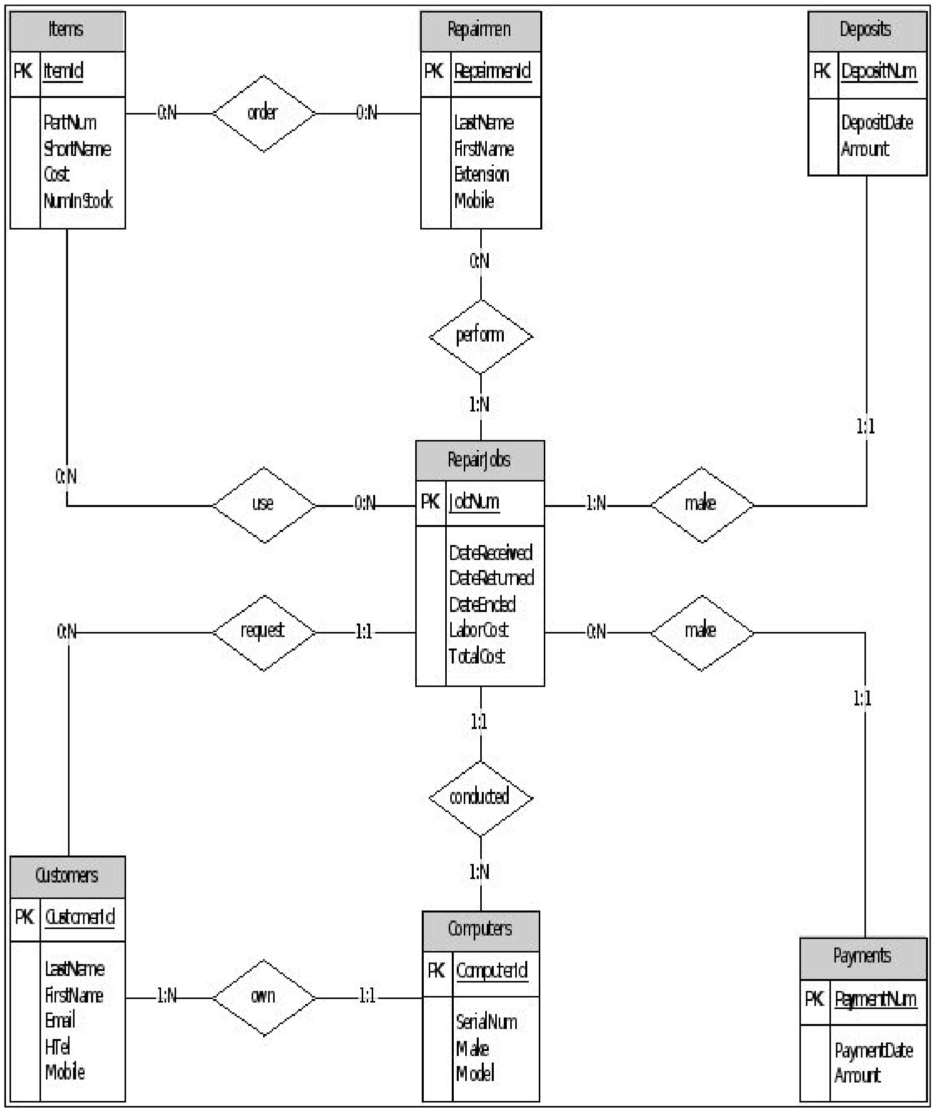

Figure 8.5 – Detailed E-R diagram for Case Study 2

Step 6: Transform the detailed E-R diagram into an implementable R-M diagram

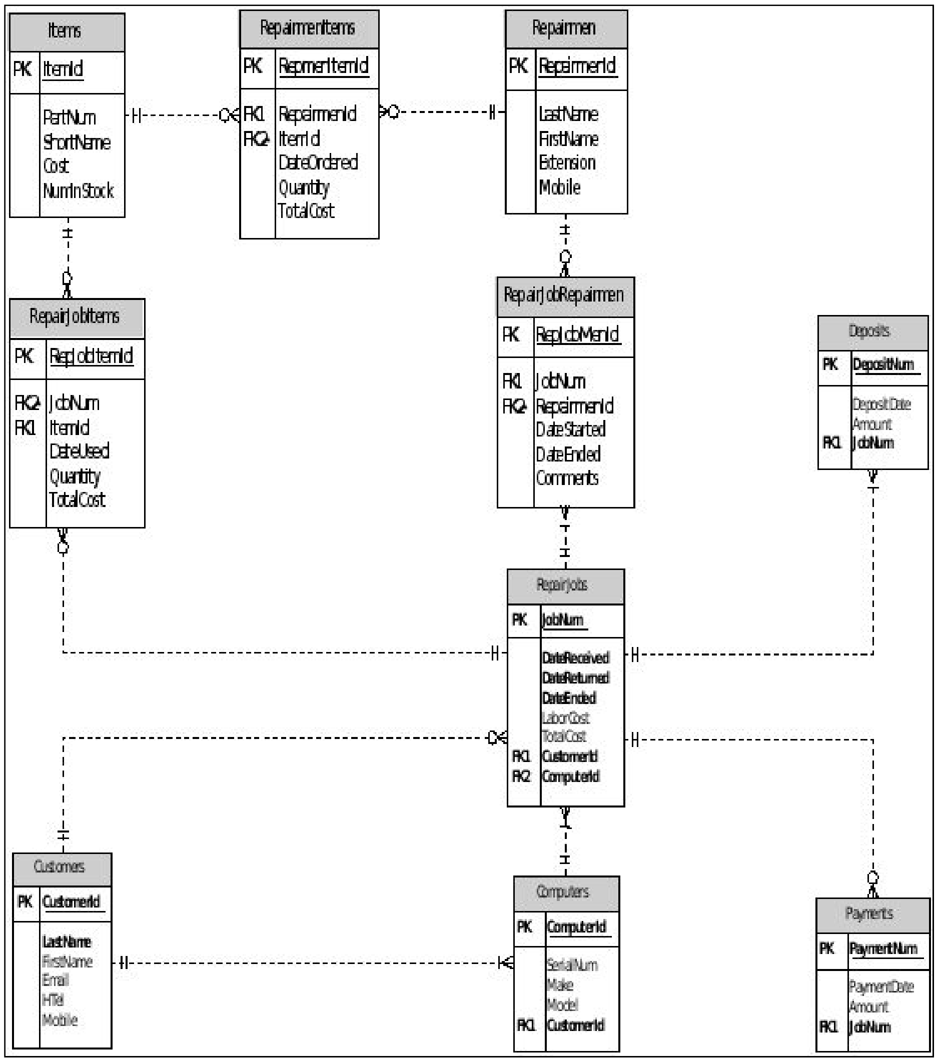

Figure 8.6 – Crow’s Foot R-M diagram for Case Study 2

Case Study 3

Problem Specification

The registrar at a small college wants an application that will help their department keep track of the scheduled classes, the courses and lecturers appearing in the schedule, and the students registering for courses according to the schedule.

Courses are scheduled every semester and this is documented in the schedule of classes, which also documents the lecturers assigned to each schedule of a class. Students register for courses according to the list of scheduled classes.

Users (students, lecturers, and other college staff) must login to the application to gain access, and the application must keep track of user logins/logouts. In addition, users must have different levels of access, which will determine their access to different parts of the application.

Step 1: Discover entities and assign attributes

Students	**Courses**	**ScheduledClasses**	**Semesters**
(PK) StudentId	**(PK) CourseCode**	**(PK) ScheduleId**	**(PK) SemesterId**
SSNumber	**ShortName**	**ScheduleCode**	SemesterNumber
LastName	LongName	Section	SemesterName
FirstName	CourseDescription	Day	SemesterYear
MiddleName		Time	StartDate
Gender		Location	EndDate
DOB			
Email			
Mobile			
HTel			
WTel			
AddressLine1			
AddressLine2			
City			
State			
PostCode			

Lecturers	**Users**	**AccessLevels**	**LogEntries**
(PK) LecturerId	**(PK) UserId**	**(PK) AccessLevelId**	**(PK) LogEntryId**
SSNumber	**Login**	**AccessLevelCode**	LoggedOn
LastName	UserName	**ShortName**	LoggedOff
FirstName	Password	LongName	
MiddleName	Active	AccessLevelDescription	
Gender			
Email			
Mobile			
HTel			
WTel			
About			

Step 2: Derive unary and binary relationships

	Students	Courses	Scheduled Classes	Semesters	Lecturers	Users	Access Levels	Log Entries
Students								
Courses		pre-requisite						
Scheduled Classes	register	scheduled						
Semesters			created for					
Lecturers		lecture	lecture					
Users	assigned				assigned	create		
Access Levels						designated		
Log Entries						create		

Step 3: Create simplified Entity-Relationship diagram

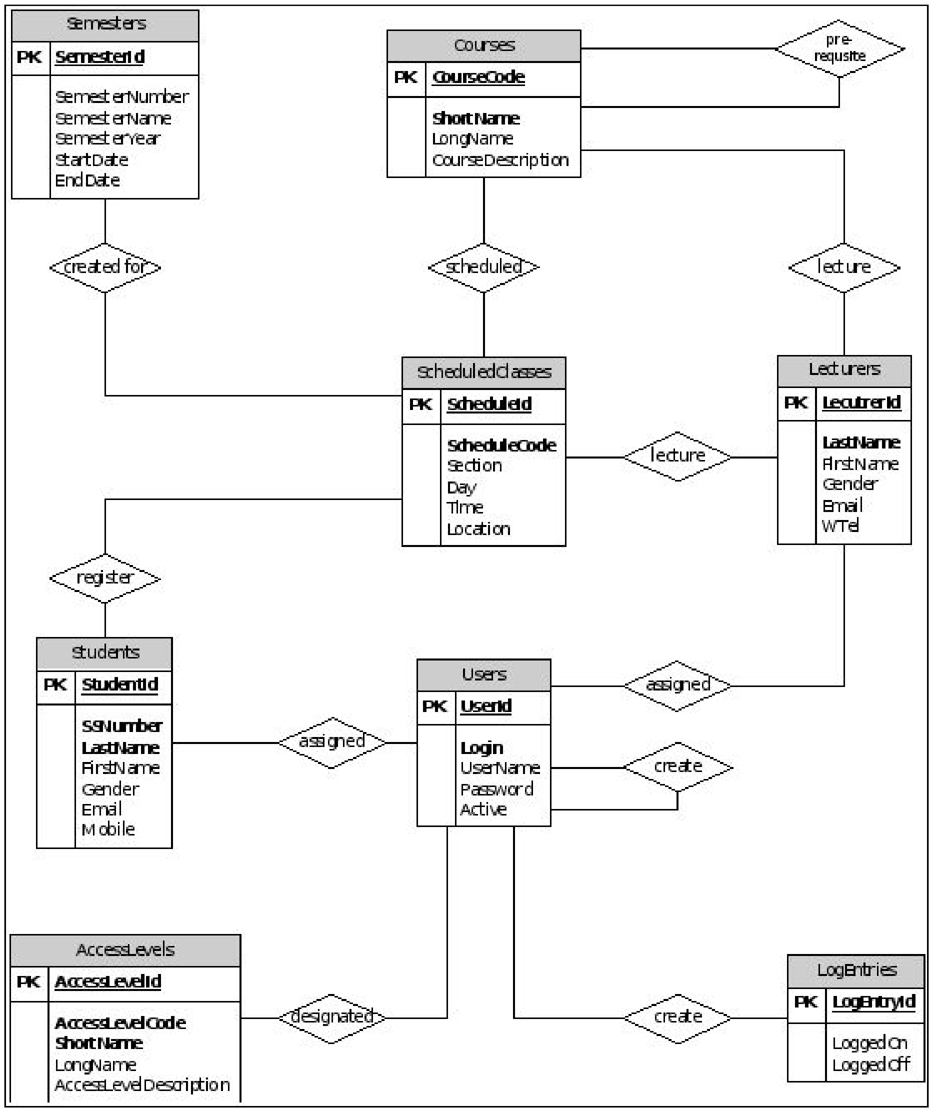

Figure 8.7 – Simplified E-R diagram for Case Study 3

Step 4: List assertions for all relationships

- A User can create many Users
- Each User must be created by only one [other] User
- A User can create many LogEntries
- Each LogEntry must be created by only one User
- An AccessLevel can be designated to many Users
- Each User must be designated only one AccessLevel
- A Student must be assigned to only one User [login]
- Each User [login] can be assigned to only one Student
- A Lecturer must be assigned to only one User [login]
- Each User [login] can be assigned to only one Lecturer
- A Course can be the prerequisite for many [other] Courses
- Each Course can have as prequisites many [other] Courses
- A Lecturer can lecture many Courses
- Each Course can be lectured by many Lecturers
- A Lecturer can lecture many ScheduledClasses
- Each ScheduledClass must be lectured by only one Lecturer
- A Course can be scheduled many times in the Scheduled [of classes]
- Each ScheduledClass must be scheduled for only one Course
- A Semester can be created for many ScheduledClasses
- Each ScheduledClass must be created for only one Semester
- A Student can register for many ScheduledClasses
- Each ScheduledClass can be registered for by many Students

Step 5: Create detailed E-R diagram using assertions

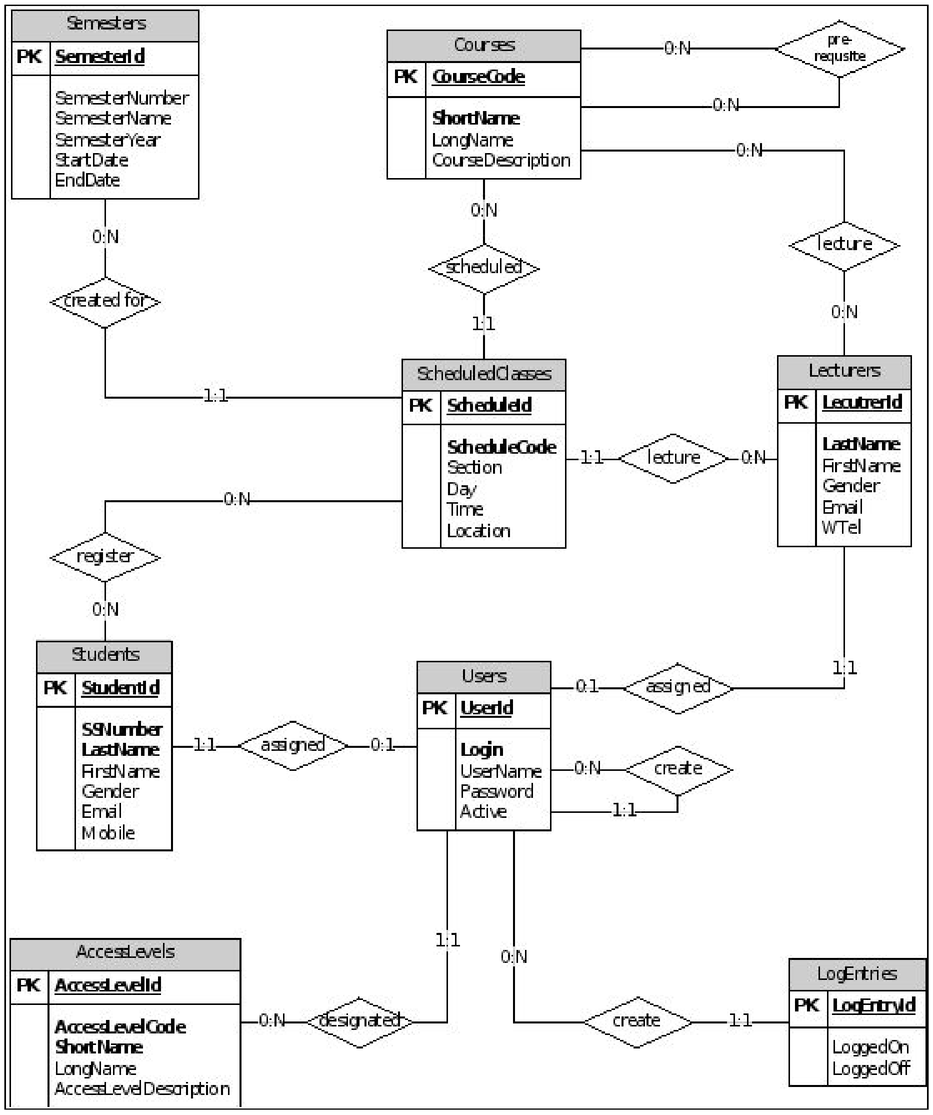

Figure 8.8 – Detailed E-R diagram for Case Study 3

Step 6: Transform the detailed E-R diagram into an implementable R-M diagram

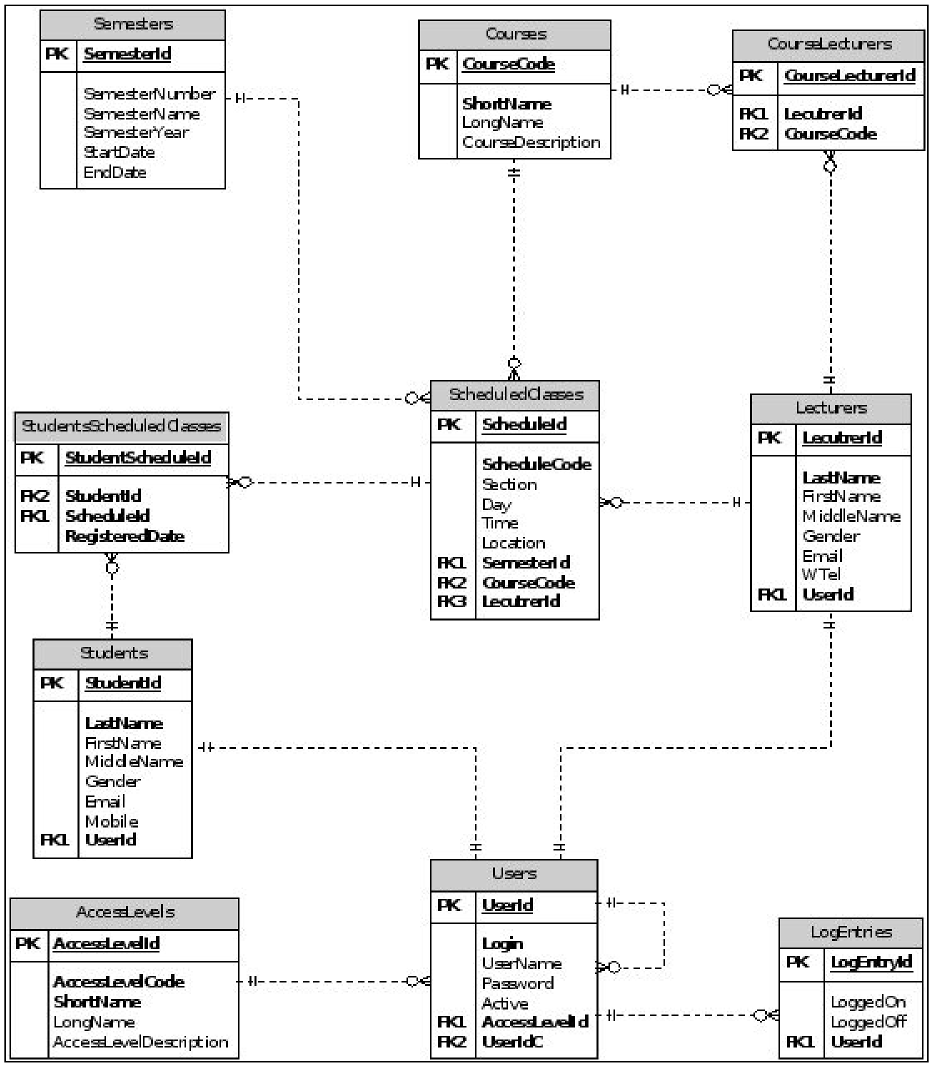

Figure 8.9 – Crow’s Foot R-M diagram for Case Study 3 (Part 1)

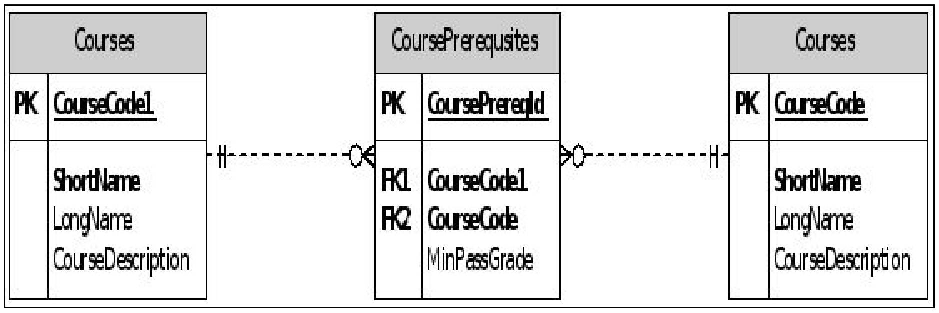

Figure 8.10 – Crow’s Foot R-M diagram for Case Study 3 (Part 2)

Summary

This chapter goes through the three case studies that have been used throughout this book, applying the six-step process to each of them. It eliminates the detailed explanations and sub-steps, applying the six-step process to each case study:

Step 1: Discover entities and assign attributes

Identify all the collective nouns and nouns in the statement of the problem that represent *objects of interest* from the problem domain. List the discovered objects of interest.

For each entity, list the possible properties and/or characteristics that are recorded in the problem domain and are relevant to the client or end user. Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it has been placed and not in any other entity or entities, and that it is not shared between or among entities.

Step 2: Derive unary and binary relationships

An E-E Matrix is used to derive the unary and binary relationships between all possible pairs of entities discovered in Step 1 of the six-step process. It is a table consisting of an equal number of rows and columns, with each entity discovered heading a row and a column. The intersection of the rows and columns represents the relationships that may exist between the entities.

Go through each cell in the Matrix, asking the question is [Entity in Row Heading] related to [Entity in Column Heading]? If a relationship [or relationships] exists, place a verb in the cell for each relationship. Each verb that is placed in the cell represents a relationship that exists between the entity on the row heading and the entity on the column heading.

Ignore the top half of the table drawn down the diagonal from the top left of the table to the bottom right of the table, since it is a mirror image of the bottom half.

Step 3: Create simplified Entity-Relationship diagram

Each of the entities derived in Step 1 the six-step process is represented by a rectangle, clearly indicating the primary key and important attributes. Each of the relationships derived in Step 2 of the six-step process is represented by a diamond with the name of the relationship in the diamond.

Create the entities (rectangles), clearly indicating the primary key and other important attributes. Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix.

Step 4: List assertions for all relationships

Each relationship on the simplified E-R diagram will yield two assertions because the relationship is looked at from two directions: from *Entity A* to *Entity B*, then in the other direction from *Entity B* to *Entity A*. Therefore, there will be twice as many assertions as there are relationships.

Look at each relationship from *Entity A* to *Entity B*, and write out the relationship in words, using the entities involved in the relationship, the optionalities and cardinalities. This is how assertions should be written:

Entity Occurrence ***optionality*** *relationship* ***cardinality*** *Entity Occurrence or Entity Class*

Look at each relationship in reverse, from *Entity B* to *Entity A*, and write out the relationship in words, using the entities involved in the relationship, the optionalities, and cardinalities.

Step 5: Create detailed E-R diagram using assertions

Each relationship on the simplified E-R diagram yielded two assertions. These two assertions will now appear on either side of the relevant relationship. Go through each relationship and its associated assertions one at a time until they have all been inserted on the diagram.

List the assertions and include **(optionality:cardinality)** at the end of each assertion, then insert the generated assertions as **optionality:cardinality** one at a time on the simplified E-R diagram, in the correct position, creating the detailed E-R diagram. Do this one relationship at a time, one assertion at a time.

Step 6: Transform the detailed E-R diagram into an implementable R-M diagram

Identify all many-to-many relationships. These relationships are identified by a cardinality of *N (many/at least one)* on both sides of the relationship. Remove both the relationship and the connectors to that relationship. Replace the relationship with a new Relation. Create new *1 to 0 or more*, or *1 to 1 or more* relationships that connect the two existing Relations to the new Relation. Ensure that the primary key for each of the two existing Relations becomes a foreign key in the new Relation, and create a new and separate primary key for the new Relation, such as a unique identifier (Id). Explore whether attributes exist for the newly created Relation. In most cases they will not, but in some cases they will.

Identify all one-to-many relationships. These relationships are identified by a cardinality of *N (many or at least one)* on one side of the relationship (diamond) and a cardinality of *1 (only one)* on one side. Group together the Relation and relationship on the side of the relationship where the cardinality is *1 (only one)*. Remove both the relationship and the connectors to that relationship, and create a new *1 to 0 or more*, or *1 to 1 or more*, relationship that connects the two Relations. The many side (Crow's Foot) of the new relationship should be on the Relation that *absorbed* the old relationship, and the primary key of the *non-absorbing* Relation becomes a foreign key in the *absorbing* Relation.

Identify all one-to-one relationships. These relationships are identifiable by having a cardinality of *1 (only one)* on *both* sides of the relationship (diamond). Group together the Relation and relationship on the side of the relationship where the optionality is *1 (must)*. If both sides have an optionality of *1 (must)*, then the old relationship can be absorbed by either Relation. Remove both the relationship and the connectors to that relationship, and create a new *1 to 1*, relationship that connects the two Relations. The primary key of the *non-absorbing* Relation becomes a foreign key in the *absorbing* Relation.

Review Questions

1. What is the purpose of the “Problem Specification”?

2. State the six steps in six-step relational database design.

3. Using an appropriate example, describe the six-step relational database design process.

Exercises

Use the problem specification below, initially given at the end of Chapter 2, to answer the questions that follow.

> The librarian of a small town's library wants a database for the library that will help it to keep track of its collections (e.g. fiction, non-fiction, and journals), the books or items in those collections, the physical location in the library of these collections, the members of the library and the books they borrow from the various collections.
>
> The database must keep track of which book or item belongs to which collection, which collection is located where in the library, and which members borrowed books or items from which collections.

1. Complete Step 1 of the six-step relational database design process as outlined in this book. State any assumptions you make regarding the client's needs.

2. Complete Step 2 of the six-step relational database design process as outlined in this book. State any assumptions you make regarding the client's needs.

3. Complete Step 3 of the six-step relational database design process as outlined in this book.

4. Complete Step 4 of the six-step relational database design process as outlined in this book. State any assumptions you make regarding the client's needs.

5. Complete Step 5 of the six-step relational database design process as outlined in this book.

6. Complete Step 6 of the six-step relational database design process as outlined in this book.

Chapter 9

Implementing the design

This chapter describes some of the considerations that need to be taken into account when implementing relational databases. It also describes how designs derived using the six steps outlined previously can be implemented on existing relational database management systems (RDBMSs). The chapter ends by giving ANSI SQL commands to implement the designs of the three case studies covered in the previous chapters.

Relational Database Management Systems (RDBMSs)

A relational database management system (RDBMS) is an administration and management system for relational databases. In relational databases all of the data is held in tables, and these tables are collectively or individually stored in files on a server or computerized system. The role of the RDBMS is to control access to these files, ensuring that they are accessed in a manner that preserves the integrity of the data in those files. In essence, it administers and manages relational databases by controlling access to the database files containing the raw data.

The RDBMS sits between the database files and the users and applications, and controls all access to these files. This is shown in figure 9.1 below:

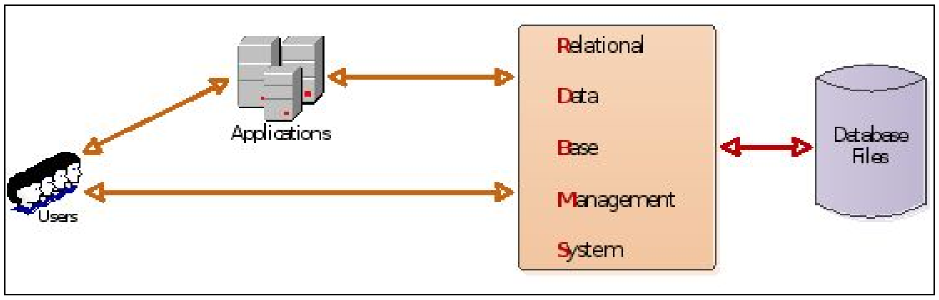

Figure 9.1 – Relational Database Management Systems

RDBMSs can be seen as great data integrity preservers. This is because in controlling access to the database files they ensure that the integrity of the data in the database files is preserved according to the rules of relational databases and those defined for the data by the creator of that data.

Relational databases are not created independently or in isolation, but by an RDMBS. After creation, the database is inextricable linked to the RDBMS that created it, and reliable access to data in that database can only be had through the RDBMS that created it or by a tool (or plugin) provided for that RDBMS. That is why databases are referred to by the name of the management system that created it – an Oracle database or a SQL server database or a MySQL database.

Transactions and ACID Compliance

A database transaction is a logical unit of *database operations* that are executed as one, all together or none at all. Each database transaction expects to find the database in a consistent and reliable state before it is processed and must leave the database in a consistent and reliable state for the next transaction after it has been processed. The RDBMS ensures that each transaction finds the database in a consistent and reliable state, processes reliably, and then leaves the database in a reliable and consistent state for the next transaction after processing.

Database transactions must be **A**tomic, **C**onsistent, **I**solated, and **D**urable (ACID). ACID compliance guarantees the integrity of the data in the tables of an RDBMS database each time a database transaction is processed:

Atomicity: Each database transaction is treated and processed as a single unit, and either all of the database operations that are part of the transaction are successfully processed, or none of the operations are processed at all.

Consistency: Each database transaction must find and leave the database in a consistent state. This state must be consistent with all of the rules of relational databases, which may be implicitly defined and implemented by the RDBMS or explicitly defined by the database's designer.

Isolation: Each database transaction is processed in isolation, and is immune to the *side effects* of other concurrent transactions and does not itself cause unintended side effects while being processed. This is also referred to as *serialization* because the transactions must *appear* to be performed serially, with each transaction receiving the data in a reliable state and leaving it in a reliable state for the next transaction.

Durability: Each database transaction's changes to the data are permanent and persist even if the system were to fail immediately after the transaction. The RDBMS is responsible for ensuring that after a transaction has been successfully processed and *committed*, the transaction's changes to the database are recoverable in the event of a system failure.

Example

Here is the Crow's Foot design for Case Study 2:

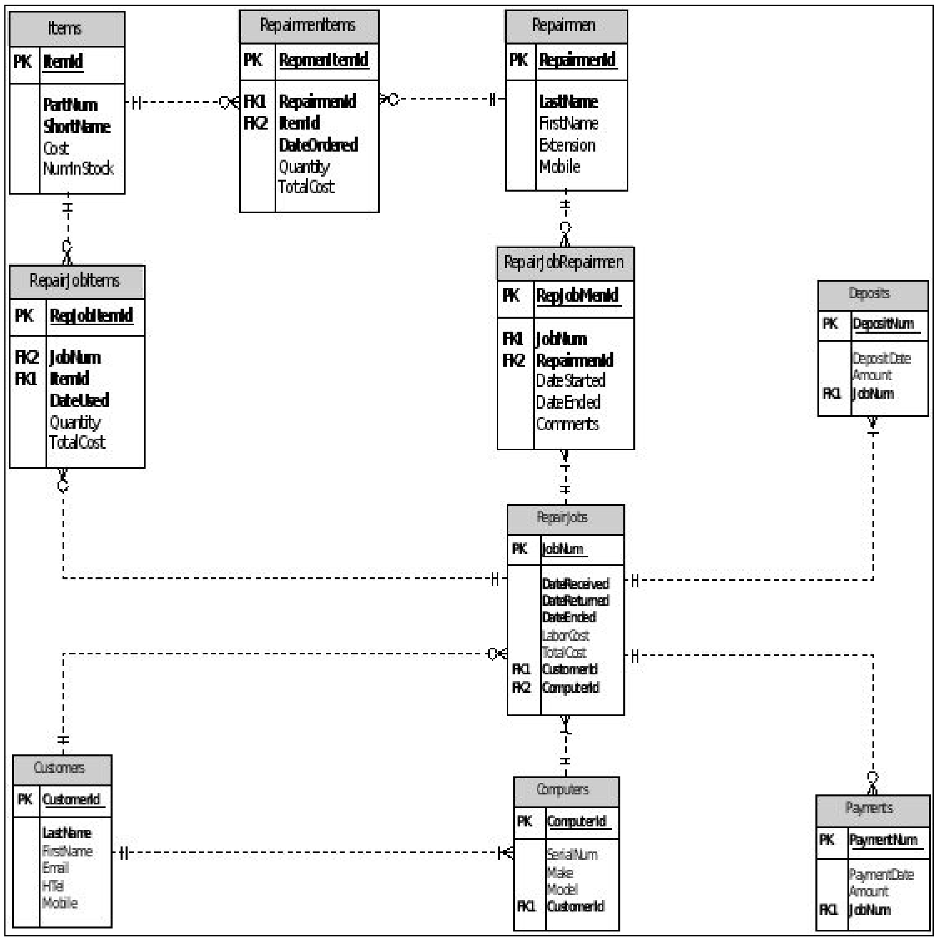

Figure 9.2 – Crow's Foot R-M diagram for Case Study 2

Now consider the following scenario for Case Study 2:

The database is implemented so that many users can access the database at the same time through a single application or multiple applications.

Item with *ItemId* 125 has a *NumInStock* value of 10 (there are 10 of item 125 in stock). Two repairmen need this item to conduct repairs to a computer, one needs 7 and the other needs 5, and both repairmen put their request in the database at exactly the same time.

In essence, the *Items* table is checked to see if there are sufficient in stock and if there are sufficient in stock the *RepairJobItems* table is updated to reflect that the specified *Items* are used in the specified *RepairJob* and the number of items in stock (*NumInStock*) for that *Item* is reduced.

What happens if both repairmen *read* the *Items* table and see that there are 10 of *Item* 125 in stock and then proceed to decrease the *NumInStock* and update the *RepairJobItems* table? This would lead to there being -2 items in stock and the *RepairJobItems* table containing erroneous data, an undesirable situation.

Implementation Considerations

A relational database management system has to be chosen that would allow multiple users to access the database simultaneously. In addition, the RDBMS would have to be able to resolve the above scenario, or at least provide the developer with a means of resolving the conflict and preserving the integrity of the data in the tables of the database.

The selected RDBMS would need to support transactions and be ACID compliant. Most RDBMSs would claim to support transactions and be ACID compliant, but as a developer you would need to investigate the levels of transaction isolation that are possible, and the level ACID compliance:

- Are transactions processed concurrently?
- Is transaction serialization possible?
- What types of locks are available to be used on tables and rows?
- Are constraints on columns possible?
- Are *dirty reads* possible?
- Are *phantom reads* possible?
- Are *repeatable reads* possible?

To solve the problem outlined in the scenario above, your RDBMS should be able to do the following:

- Prevent erroneous values from being entered into columns in tables. For example, negative values should not be allowed to be entered in the *NumInStock* column.
- Process transactions. You should be able to create a transaction that could read the value of *NumInStock* and update its value if necessary (part of an all or none proposition).
- Isolate transactions appropriately. You should be able to prevent transactions from reading the value of *NumInStock* while it is being used by another transaction in which it may be updated, has been marked to be updated, or has been updated and not committed.

RDBMSs would offer many flavors of the tools and techniques needed to provide a solution to the above scenario. However, there are tradeoffs that may have to be made to preserve the integrity of the data in the tables of your database.

Normalization

Normalization is the process of organizing the Relations (tables) in a database so that they reduce data redundancy and prevent inconsistent data dependencies. It is a very important part of database design, and RDBMSs must provide tools and techniques for implementing normalized databases.

There are at least three normal forms associated with normalization: first normal form (1NF), second normal form (2NF), and third normal form (3NF). Other higher forms, such as Boyce-Codd Normal Form (BCNF), fourth, and fifth normal forms do exist, but 1NF, 2NF, and 3NF are usually sufficient to *“reduce data redundancy and prevent inconsistent data dependencies.”*

First Normal Form (1NF)

Any Relation (table) that is in first normal form exhibits *domain integrity*. A domain is the set of all possible values of a given attribute, and domain integrity means that all the possible values of an attribute are legitimate ones. Here is what needs to be done to achieve domain integrity:

- Every table must have a primary key. This is to ensure that each row is unique.
- Every column within a table must have a unique name, and represent a single value. Repeating groups and multi-valued columns are not allowed.

Second Normal Form (2NF)

Any table that is in second normal form exhibits *entity integrity* and is also, by definition, in first normal form. Entity integrity ensures that each row in the table is a unique and genuine entity, and that every other column in that table depends solely on the primary key. Here is what needs to be done to achieve entity integrity:

- Every column in a table must depend solely on the primary key, and if the primary key is a composite primary key every column must depend on all the columns that make up the primary key and not on a subset of the composite primary key.

Third Normal Form (3NF)

Any table that is in third normal form exhibits *referential integrity* and is also, by definition, in first and second normal forms. Referential integrity ensures that all references to values in columns in other tables are authentic – authentic foreign key values. Here is what needs to be done to achieve referential integrity:

- Every reference made in one table to the primary key of another table must be valid. That is, all foreign key values must be valid existing primary key values in another table.
- If the value of a primary key changes in its defined table, all references to that primary key value must change consistently throughout the database in all tables that reference that primary key value.

Cascade Updates and Cascade Deletes

Cascade updates and cascade deletes are mechanisms used by RDBMSs and developers, either implicitly or explicitly, to help ensure referential integrity. Referential integrity is a characteristic of third normal form and ensures that foreign keys reference valid primary keys. Consider the example below, taken from Case Study 2, figure 9.2:

> *JobNum* is a foreign key in the *Deposits* [and *Payments*] table that references the *JobNum* primary key in the *RepairJobs* table. If *JobNum* changes from 2 to 3 in the *RepairJobs* table then it must also change from 2 to 3 in the *Deposits* [and *Payments*] table to preserve the integrity of the data in the table (cascade updates).
>
> If the record (row) where *JobNum* has the value of 2 is deleted from the *RepairJobs* table and there is a corresponding *JobNum* in the *Deposits* [and *Payments*] table then the corresponding record (row) in the *Deposits* [and *Payments*] table must also be deleted to preserve the integrity of the data in the table (cascade deletes).

Cascade updates ensures referential integrity by updating all references to a primary key value which has changed, by updating all corresponding foreign key values for that primary key. Some RDBMSs prevent the update of auto-generated primary keys, making cascade updates unnecessary if all the primary keys in your database are auto-generated. However, it is best to always explicitly define foreign keys as cascade update when creating the table, if possible.

Cascade deletes ensures referential integrity by deleting all references to a primary key record (row) which has been deleted, by deleting all corresponding foreign key records (rows) for that primary key. Cascade deletes maintains referential integrity, but usually has many unintended *side effects* as demonstrated in the example below.

It is very import to know how cascade updates and cascade deletes are made available to you as a developer, and what the defaults are for the RDBMS you are implementing your database on. There is no harm in explicitly specifying, if you can, cascade updates and/or cascade deletes as necessary when you create the tables for your database.

Example

Here is a partial Crow's Foot design for Case Study 2:

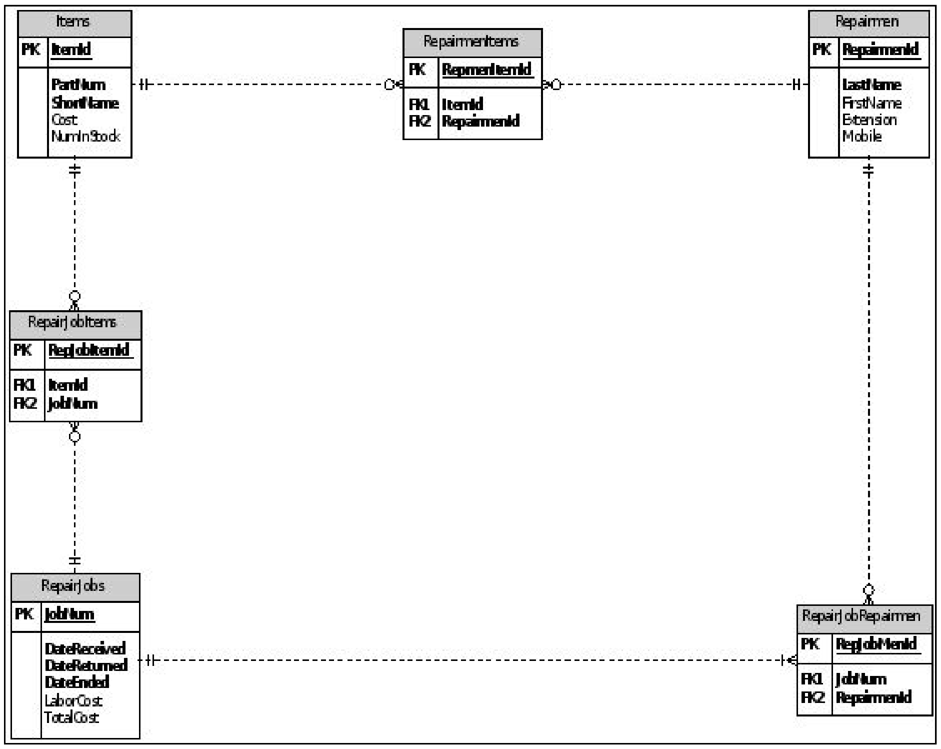

Figure 9.3 – Partial Crow's Foot R-M diagram for Case Study 2

Now consider the following scenario for Case Study 2:

Item with *ItemId* 973 is no longer being manufactured and supplied, so Inventory wants to remove this item from the list of available items for *RepairJobs* and that *Repairmen* can order.

What would happen to existing records in the *RepairJobItems* and *RepairmenItems* table that still have *ItemId* 973? If *ItemId* 973 is deleted then it is possible that there will be invalid references to it in the *RepairJobItems* and *RepairmenItems* table, an undesirable situation.

Implementation Considerations

Most RDBMSs would, by default, prevent the deletion of a record (row) that has foreign key values associated with the primary key value of the record being deleted. However, by enabling cascade deletes when you create the table the deletion would go through, but would also delete all related records (rows). Some RDBMSs might have settings that you can change to allow such deletions, but it is ill advised to manipulate these settings as this can lead to inadvertent data loss in your database.

In most instances cascade deletes, such as the one above, can have disastrous *side effects*. If the deletes were allowed to cascade, preserving referential integrity, then existing *RepairJobItems* that were used in previous repairs will no longer be part of the database, and the *TotalCost* on *RepairJobs* that were generated before the delete would be incorrect because they would have included *RepairJobItems* that now do not exist.

Great caution should always be taken when deleting records from a table in a relational database. The trend nowadays is to mark the record as deleted by creating a column called *deleted* making it a *yes/no* column with a value of *yes (1)* for deleted, and sometime in the future you can go to your *trash* and really delete that record, or undelete it if it was deleted in error. In these instances you may be prompted to delete related records first if cascade deletes was not enabled for related tables.

It is important to note that the six-step process outlined in this book creates normalized Relations (tables) that are in third normal form. However, there may be instances where deviation from the process outlined in this book may be advantageous or necessary, resulting in de-normalized Relations (tables).

Indexes

An index (key) is a data structure that is created for a database table but is external to that table, and if well selected and implemented can result in faster searches for data in that table. However, indexes do not guarantee that data in a table will be accessed faster or that it would speed up searches for data in that table. If poorly implemented, indexes can significantly reduce database performance and take up as much or even more space than the data in the tables themselves.

Indexes have two main architectures: *clustered* and *non-clustered*. Clustered indexes store the data in the database file(s) in the same order as the index, while non-clustered indexes do not. This means that each table can only have one clustered index, which is usually the primary key.

Indexes are usually implemented as binary trees (B trees), B+ trees, and hashes. B trees and B+ trees are used to implement clustered indexes, and hashes are used to implement non-clustered indexes.

Most RDBMSs will allow you to select which columns in a table you want to use as indexes, but will not allow you to select the architecture and implementation of those indexes. ANSI SQL provides the CREATE INDEX statement to create indexes on table columns, but the RDBMS usually determines the flexibility given to developers to choose the type and architecture of the index created.

Primary Keys and Foreign Keys

As mentioned in Chapter 1, the primary key for a table *“is a unique identifier that is used to distinguish or identify rows in a Relation (table)”*, and a foreign key is *“a primary key in one Relation (table) that appears as a column in another Relation (table) and is used to join the two Relations (tables) together in a relationship.”* Every table in a normalized relational database *must* have a primary key.

Primary keys and foreign keys play an important role in storing, organizing, and finding data stored in these databases. As a result, primary keys and foreign keys should **always** be indexed.

Once a column is defined as the primary key of a table, the RDBMS will automatically create an index for this column. However, this may not be the case for foreign keys, and you should take care to **always** clearly define indexes for foreign keys in tables, if not automatically done by the RDBMS.

General Implementation Considerations

Indexes speed up searches but slow down inserts, updates, and deletes. So, as a general rule of thumb, if a table will be updated more than it will be searched, keep the number of indexes to a minimum.

Computers work better with whole numbers than with characters, so indexes on numbers (integers) will always be faster than indexes on characters. As a rule of thumb, try to make the primary key, which is often implemented as a clustered index, an automatically generated sequential whole number – *auto-number* or *auto-increment*.

Indexes govern how the data in a table is accessed and searched, so as a database designer and developer you need to be aware of how the data will be accessed and searched by the client. You need to know what kind of queries will be conducted on the data in order to know what columns to create indexes on.

Here are some general guidelines to follow when creating indexes:

- Index *all* primary keys (if not already done by the RDBMS)
- Index *all* foreign keys
- Index columns used in JOINs. These will most likely be foreign keys.
- Index columns used in WHERE clauses of queries.
- Index columns used in ORDER BY clauses of queries.
- Index columns used in GROUP BY clauses of queries.

Example

Consider the following columns for Case Study 3:

Students	**Courses**	**ScheduledClasses**	**Semesters**
(PK) StudentId	**(PK) CourseCode**	**(PK) ScheduleId**	**(PK) SemesterId**
SSNumber	**ShortName**	**ScheduleCode**	SemesterNumber
LastName	LongName	Section	SemesterName
FirstName	CourseDescription	Day	SemesterYear
MiddleName		Time	StartDate
Gender		Location	EndDate
DOB			
Email			
Mobile			
HTel			
WTel			
AddressLine1			
AddressLine2			
City			
State			
PostCode			

Implementation Considerations

Based on the guidelines previously given it appears that there can be some improvement in the choices for indexes and primary keys:

- SSNumber - Not likely to be used in many searches. LastName and FirstName are more likely to be used.
- PostCode – Not likely to be used in many searches. DOB is more likely to be used.
- CourseCode – A text field for a primary key. Can slow down table access.

It is recommended that *SSNumber* and *PostCode* be removed as indexes, and that indexes be created for *DOB* and for the *LastName FirstName* composite. In addition, a new primary key should be created for the *Courses* table called *CourseId*, and *CourseCode* should be an ordinary index.

This example highlights how important it is to know a little about how the design will be implemented, and what queries will be run on the database. Subtle changes in the design and implementation of the database regarding indexes can make a dramatic difference in the performance of the database. It is a good idea to **always** have the Reference Manual of the RDBMS handy when implementing your design so you can make better decisions about indexes.

Data Types

Data in the database is stored in tables, and each table is made up of one or more columns. Each column holds one *unit* of data in the table and represents the smallest indivisible *unit* in which data can be stored in a database. Each of these columns (*units*) must specify the type of data that it holds, which is referred to as the *data type* of that column.

The type of data that a column holds is specified when the table is being created and can be done via a graphical tool or by using SQL commands. Different RDBMSs may offer slightly different options to the types of data that can be specified for columns, but they generally fall into four main categories:

- **Character data** – This type of data is used to represent fixed or variable length character strings. RDBMSs offer different sub-types of this data type that are based on the maximum number of characters in the string, including very long strings referred to as *text*.
- **Numeric data** – This type of data is used to represent numeric data, such as whole numbers, decimal numbers, real numbers, and even auto-numbers generated specifically for primary key (Id) columns. Most RDBMSs offer a rich variety of sub-types for this data type that are based on the size and type of number that the column will hold.
- **Date data** – This type of data is used to represent dates, times, and a combination of dates and times. The sub-types offered for this data type are usually very specific to the RDBMS and may store data in a format that is not portable from one RDBMS to another.
- **Binary data** – This type of data is used to represent binary data, such as simple *true/false* values or large binary objects, such as, images. RDBMSs will offer options as to how and where this binary data is stored, especially if the binary data is expected to be very large.

RDBMSs may offer data types that do not fall into one of the categories listed above. It is very important that you refer to the Reference Manual of the RDBMS to verify the types of data available, and to assess which of the available types is best suited for your column's data.

Example

Consider the following tables for Case Study 3, modified according to the recommendations given in the previous section:

Students	**Courses**	**ScheduledClasses**	**Semesters**
(PK) StudentId	**(PK) CourseId**	**(PK) ScheduleId**	**(PK) SemesterId**
SSNumber	**CourseCode**	**ScheduleCode**	SemesterNumber
LastName	**ShortName**	Section	SemesterName
FirstName	LongName	Day	SemesterYear
MiddleName	CourseDescription	Time	StartDate
Gender		Location	EndDate
DOB			
Email			
Mobile			
HTel			
WTel			
AddressLine1			
AddressLine2			
City			
State			
PostCode			

Implementation Considerations

When selecting the data types for columns it is important to first look closely at the type of data that will be stored in the column and the options available by the RDBMS for storing that type of data. RDBMSs offer many sub-types to the categories previously mentioned, and one of these sub-types will often be exactly what you need. **Always** refer to the Reference Manual of the RDBMS.

Another important consideration is for the *size* of the data, and this is something that you will have to finalize with the client. Some other things you may need to find out are the range and type of numbers that need to be represented, the maximum number of characters that may be needed for a column, and should a value be stored as a number or a character string.

The table below gives a breakdown of the possible data types along with the considerations that need to be taken into account when implementing the columns of tables for Case Study 3:

Table	**Column**	**Recommend**	**Comments**

		Data Type	
Students	**(PK) StudentId**	Number	An automatically generated sequential whole number.
	SSNumber	Character	Fixed length character string of numbers and dashes, with a length that's limited to the length of the Social Security number.
	LastName	Character	Variable length character string, with the maximum length set to a reasonable number (e.g. 50).
	FirstName		
	MiddleName		
	Gender	Character	Variable length character string, with the maximum length set to 6; or a fixed length character string of 1 character.
	DOB	Date	Date only.
	Email	Character	Variable length character string, with the maximum length set to a reasonable number for the 'Email Address' (e.g. 75).
	Mobile	Character	Fixed length character string of numbers and dashes with the maximum length set to the maximum length of the phone number you want to store (e.g. 12).
	HTel		
	WTel		
	AddressLine1	Character	Variable length character string, with the maximum length set to a reasonable number for the 'Address Line' (e.g. 75).
	AddressLine2		
	City	Character	Variable length character string, with the maximum length set to a reasonable number for the 'City' (e.g. 50).
	State	Character	Fixed or variable length character string depending on what will be stored for the 'State' (e.g. 'NY' or 'New York').
	PostCode	Character	Fixed length character string, with the maximum length set to the maximum length of the 'Post Code'.

Table	**Column**	**Recommend Data Type**	**Comments**
Courses	**(PK) CourseId**	Number	An automatically generated sequential whole number.
	CourseCode	Character	Fixed length character string, with the maximum length set to the maximum length of the ‘Course Code’.
	ShortName	Character	Variable length character string, with the maximum length set to a reasonable number for the ‘Short Name’ (e.g. 35).
	LongName	Character	Variable length character string, with the maximum length set to a reasonable number for the ‘Long Name’ (e.g. 150).
	CourseDescription	Character	Variable length character string, with the maximum length set to a reasonable number for the ‘Course Description’ (e.g. 5,000). Alternative data types may be available for holding large variable length character strings.
ScheduledClasses	**(PK) ScheduleId**	Number	An automatically generated sequential whole number.
	ScheduleCode	Character	Fixed length character string, with the maximum length set to the maximum length of the ‘Schedule Code’.
	Section	Character	Fixed length character string, with the maximum length set to the maximum length of the ‘Section’.
	Day	Character	Fixed or variable length character string depending on what will be stored for the ‘Day’ (e.g. ‘Th’ or ‘Thursday’).
	Time	Date	Time only.
	Location	Character	Variable length character string, with the maximum length set to a reasonable number for the ‘Location’ (e.g. 35).
Semesters	**(PK) SemesterId**	Number	An automatically generated sequential whole number.
	SemesterNumber	Number	A small positive whole number.
	SemesterName	Character	Variable length character string, with the maximum length set to a reasonable number for the ‘Semester Name’ (e.g. 6).
	SemesterYear	Date or Number	Year only as a *Date* data type, if possible, otherwise a positive whole number.
	StartDate	Date	Date only.
	EndDate		

Remember to **always** confirm with the client the data type and maximum width of columns, and to refer to the Reference Manual of the RDBMS for a list of available data types.

Implementation of Case Study Designs

The R-M diagrams created using the six-step process described in this book can be implemented on any RDBMS. This chapter outlines some of the many considerations that need to be made when moving from the design to the implementation. Remember to *always* have the Reference Manual of the RDBMS handy when implementing your design.

The SQL commands below, for the case studies, can be used to implement the designs (create the tables) in a MySQL database. Some modification *will* be necessary to execute these commands on MS SQL Server, Oracle, or any other RDBMS.

Case Study 1

The R-M diagram in figure 9.4 can be implemented on any RDBMS. The SQL commands on the subsequent pages can be used to implement the design (create the tables) in a MySQL database. Similar SQL commands can be used to create the tables on any RDBMS.

Entities, Attributes, and the R-M Diagram

The entities, attributes and R-M diagram for Case Study 1 are given below:

Employees	**Positions**	**Allowances**	**SalaryScales**	**Vehicles**
(PK) EmployeeId	**(PK) PositionId**	**(PK) AllowanceId**	**(PK) SalaryScaleCode**	**(PK) VehicleId**
SSNumber	**PositionName**	**AllowanceName**	**SalaryScaleName**	**VIN**
LastName	PositionDescription	AllowanceDescription	SalaryScaleDescription	**RegistrationNo**
FirstName	Details	Amount	MinimumSalary	Year
MiddleName			MaximumSalary	Make
Gender				Model
DOB				Color
Email				
Mobile				
HTel				
AddressLine1				
AddressLine2				
City				
State				
PostCode				

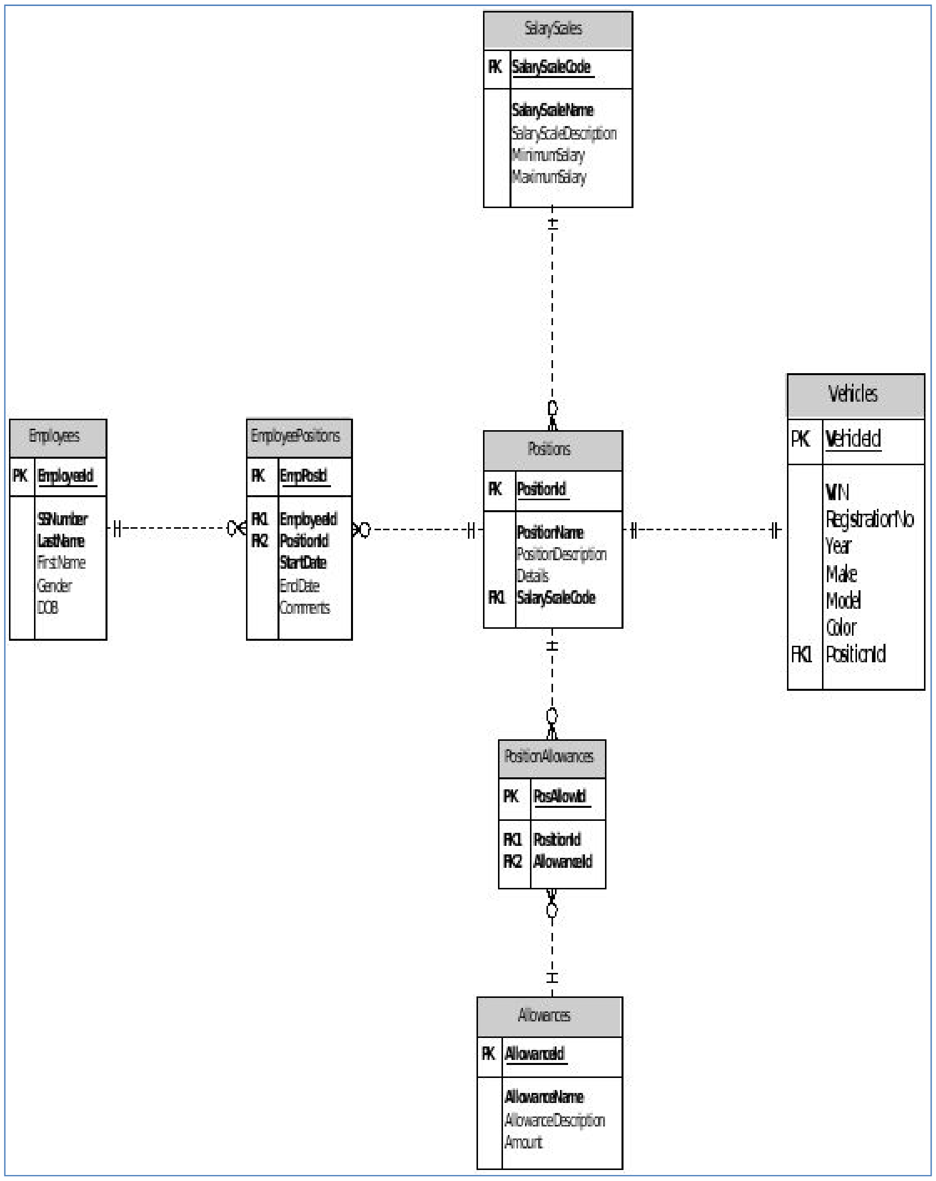

Figure 9.4 – Crow’s Foot R-M diagram for Case Study 1

Data Dictionary

The table below gives a breakdown of the possible data types along with the other important considerations that need to be taken into account when implementing the columns of the tables for Case Study 1:

Table	Column	Recommend Data Type	Length	Indexed	Required (Default)
Employees	**(PK) EmployeeId**	Number	NA	Yes, Primary Key	Yes
	SSNumber	Character	11	Yes	Yes
	LastName	Character	50	Yes, Composite	Yes
	FirstName				
	MiddleName			No	No
	Gender	Character	6	No	Yes (‘MALE’)

	DOB	Date	NA	Yes	Yes
	Email	Character	75	No	Yes
	Mobile	Character	14	No	No
	HTel				Yes
	AddressLine1	Character	75	No	Yes
	AddressLine2				No
	City	Character	50	No	Yes
	State				
	PostCode	Character	10	Yes	Yes
Allowances	**(PK) AllowanceId**	Number	NA	Yes, Primary Key	Yes
	AllowanceName	Character	50	Yes	Yes
	AllowanceDescription	Character	250	No	No
	Amount	Number	12	No	Yes (0.00)
SalaryScales	**(PK) SalaryScaleCode**	Number	4	Yes, Primary Key	Yes
	SalaryScaleName	Character	50	Yes	Yes
	SalaryScaleDescription	Character	250	No	No
	MinimumSalary	Number	12	No	Yes (0.00)
	MaximumSalary	Number	12	No	Yes (0.00)
Positions	**(PK) PositionId**	Number	NA	Yes, Primary Key	Yes
	PositionName	Character	75	Yes	Yes
	PositionDescription	Character	250	No	No
	Details	Character	5,000	No	No
	(FK) SalaryScaleCode	Number	4	Yes, Foreign Key	Yes

Table	**Column**	**Recommend Data Type**	**Length**	**Indexed**	**Required (Default)**
Vehicles	**(PK) VehicleId**	Number	NA	Yes, Primary Key	Yes
	VIN	Character	17	Yes	Yes
	RegistrationNo	Character	10	Yes	Yes
	Year	Date or Number	4	No	No
	Make	Character	25	No	No
	Model				
	Color				
	(FK) PositionId	Number	NA	Yes, Foreign Key	Yes
Position Allowances	**(PK) PosAllowId**	Number	NA	Yes, Primary Key	Yes
	(FK) PositionId	Number	NA	Yes, Foreign Key	Yes
	(FK) AllowanceId				
Employee Positions	**(PK) EmpPosId**	Number	NA	Yes, Primary Key	Yes
	(FK) EmployeeId	Number	NA	Yes, Foreign Key	Yes
	(FK) PositionId				
	StartDate	Date	NA	Yes	Yes
	EndDate			No	No
	Comments	Character	5,000	No	No

SQL Commands

The SQL commands to implement the design (create the tables) for Case Study 1 in a MySQL database are given below:

```
CREATE TABLE Employees (
EmployeeId int(11) NOT NULL AUTO_INCREMENT,
SSNumber varchar(11) NOT NULL,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MiddleName varchar(50) DEFAULT NULL,
Gender varchar(6) NOT NULL DEFAULT 'MALE',
DOB date NOT NULL,
Email varchar(75) NOT NULL,
Mobile varchar(14) DEFAULT NULL,
HTel varchar(14) NOT NULL,
AddressLine1 varchar(75) NOT NULL,
AddressLine2 varchar(75) DEFAULT NULL,
City varchar(50) NOT NULL,
State varchar(50) NOT NULL,
PostCode varchar(10) NOT NULL,
PRIMARY KEY (EmployeeId),
INDEX SSNumber (SSNumber),
INDEX DOB (DOB),
INDEX PostCode (PostCode),
INDEX FullName (LastName,FirstName)
);

CREATE TABLE Allowances (
AllowanceId int(11) NOT NULL AUTO_INCREMENT,
AllowanceName int(11) NOT NULL,
AllowanceDescription varchar(250) DEFAULT NULL,
Amount decimal(10,2) NOT NULL DEFAULT '0.00',
PRIMARY KEY (AllowanceId),
INDEX AllowanceName (AllowanceName)
);

CREATE TABLE SalaryScales (
SalaryScaleCode tinyint(4) NOT NULL,
SalaryScaleName varchar(50) NOT NULL,
SalaryScaleDescription varchar(250) DEFAULT NULL,
MinimumSalary decimal(10,2) NOT NULL DEFAULT '0.00',
MaximumSalary decimal(10,2) NOT NULL DEFAULT '0.00',
PRIMARY KEY (SalaryScaleCode),
INDEX SalaryScaleName (SalaryScaleName)
);

CREATE TABLE Positions (
PositionId int(11) NOT NULL AUTO_INCREMENT,
PositionName varchar(75) NOT NULL,
PositionDesctiption varchar(250) DEFAULT NULL,
Details text,
SalaryScaleCode tinyint(4) NOT NULL,
PRIMARY KEY (PositionId),
FOREIGN KEY SalaryScaleCode (SalaryScaleCode)
    REFERENCES SalaryScales (SalaryScaleCode)
    ON UPDATE CASCADE ON DELETE RESTRICT,
INDEX PositionName (PositionName)
);

CREATE TABLE Vehicles (
VehicleId int(11) NOT NULL AUTO_INCREMENT,
VIN varchar(17) NOT NULL,
RegistrationNo varchar(10) NOT NULL,
Year year(4) DEFAULT NULL,
Make varchar(25) DEFAULT NULL,
Model varchar(25) DEFAULT NULL,
Color varchar(25) DEFAULT NULL,
PositionId int(11) NOT NULL,
PRIMARY KEY (VehicleId),
FOREIGN KEY PositionId (PositionId)
    REFERENCES Positions (PositionId)
    ON UPDATE CASCADE ON DELETE RESTRICT,
INDEX VIN (VIN),
INDEX RegistrationNo (RegistrationNo)
);

CREATE TABLE PositionAllowances (
PosAllowId int(11) NOT NULL AUTO_INCREMENT,
AllowanceId int(11) NOT NULL,
PositionId int(11) NOT NULL,
PRIMARY KEY (PosAllowId),
FOREIGN KEY AllowanceId (AllowanceId)
    REFERENCES Allowances (AllowanceId)
    ON UPDATE CASCADE ON DELETE RESTRICT,
FOREIGN KEY PositionId (PositionId)
    REFERENCES Positions (PositionId)
    ON UPDATE CASCADE ON DELETE RESTRICT
);

CREATE TABLE EmployeePositions (
EmpPosId int(11) NOT NULL AUTO_INCREMENT,
EmployeeId int(11) NOT NULL,
PositionId int(11) NOT NULL,
StartDate date NOT NULL,
EndDate date DEFAULT NULL,
Comments text,
PRIMARY KEY (EmpPosId),
FOREIGN KEY EmployeeId (EmployeeId)
    REFERENCES Employees (EmployeeId)
    ON UPDATE CASCADE ON DELETE RESTRICT,
FOREIGN KEY PositionId (PositionId)
    REFERENCES Positions (PositionId)
    ON UPDATE CASCADE ON DELETE RESTRICT,
INDEX StartDate (StartDate)
);
```

Notice the order in which the tables were created. The tables with no foreign keys were created first and the many-to-many join tables were created last. Also, the table with the primary key for the referenced foreign key is always created before the table with the foreign key.

There was no need to create indexes for foreign keys because MySQL automatically does this for *InnoDB* tables, which is the default storage engine for MySQL 5.7.

Remember: Always have the Reference Manual of the RDBMS handy when implementing your design.

Case Study 2

The R-M diagram in figure 9.5 can be implemented on any RDBMS. The SQL commands on the subsequent pages can be used to implement the design (create the tables) in a MySQL database. Similar SQL commands can be used to create the tables on any RDBMS.

Entities, Attributes, and the R-M Diagram

The entities, attributes and R-M diagram for Case Study 2 are given below:

RepairJob	**Computers**	**Items**	**Repairmen**	**Customers**
(PK) JobNum	**(PK) ComputerId**	**(PK) ItemId**	**(PK) RepairmenId**	**(PK) CustomerId**
DateReceived	**SerialNum**	**PartNum**	**LastName**	**LastName**
DateToReturn	Make	**ShortName**	FirstName	FirstName
DateReturned	Model	ItemDescription	MI	MI
DateStarted	ComputerDescription	Cost	Email	Email
DateEnded		NumInStock	Mobile	Mobile
RepairDetails		ReorderLow	HTel	HTel
LaborDetails			Extension	AddressLine1
LaborCost				AddressLine2
TotalCost				City
PaidInFull				State
AdditionalComments				**PostCode**

Deposits	**Payments**
(PK) DepositNum	**(PK) PaymentNum**
DepositDate	PaymentDate
Amount	Amount

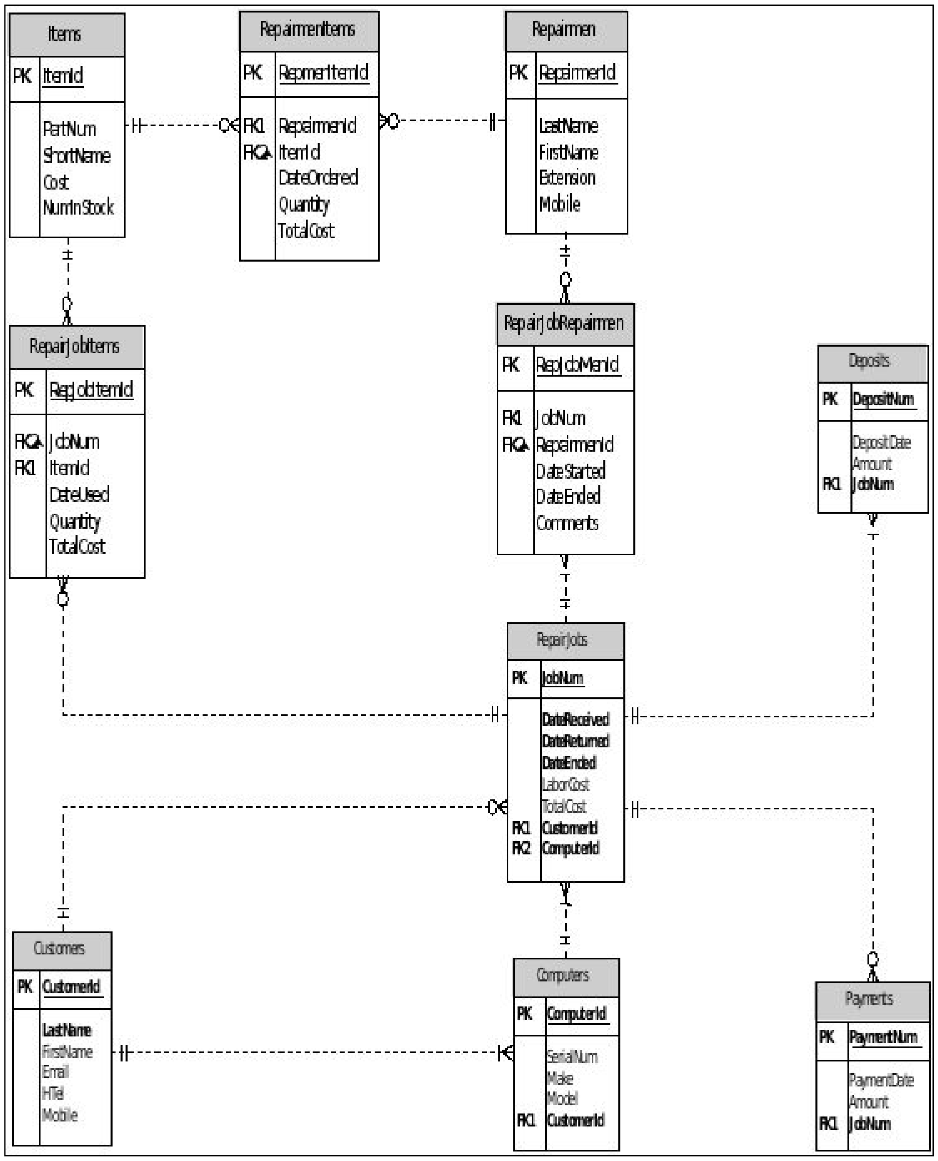

Figure 9.5 – Crow’s Foot R-M diagram for Case Study 2

Data Dictionary

The table below gives a breakdown of the possible data types along with the other important considerations that need to be taken into account when implementing the columns of the tables for Case Study 2:

Table	Column	Recommend Data Type	Length	Indexed	Required (Default)
Items	**(PK) ItemId**	Number	NA	Yes, Primary Key	Yes
	PartNum	Character	50	Yes	Yes
	ShortName	Character	75	Yes	Yes
	Cost	Number	12	No	Yes (0.00)
	NumInStock	Number	6	No	Yes (1)
Repairmen	**(PK) RepairmenId**	Number	NA	Yes, Primary Key	Yes
	LastName	Character	50	Yes, Composite	Yes

	FirstName		50		
	MI	Character	1	No	No
	Email	Character	75	No	Yes
	Mobile	Character	14	No	No
	HTel				Yes
	Extension	Character	5	No	No
Customers	**(PK) CustomerId**	Number	NA	Yes, Primary Key	Yes
	LastName	Character	50	Yes, Composite	Yes
	FirstName		50		
	MI	Character	1	No	No
	Email	Character	75	No	Yes
	Mobile	Character	14	No	No
	HTel				Yes
	AddressLine1	Character	75	No	Yes
	AddressLine2				No
	City	Character	50	No	Yes
	State				
	PostCode	Character	10	Yes	Yes
Computers	**(PK) ComputerId**	Number	NA	Yes, Primary Key	Yes
	SerialNum	Character	50	Yes	Yes
	Make	Character	50	No	No
	Model				
	ComputerDescription	Character	250	No	No
	(FK) CustomerId	Number	NA	Yes, Foreign Key	Yes

Table	Column	Recommend Data Type	Length	Indexed	Required (Default)
RepairJobs	**(PK) JobNum**	Number	NA	Yes, Primary Key	Yes
	DateReceived	Date	NA	Yes	Yes
	DateReturned				
	DateEnded			No	No
	LabourCost	Number	12	No	Yes (0.00)
	TotalCost				
	(FK) CustomerId	Number	NA	Yes, Foreign Key	Yes
	(FK) ComputerId				
Deposits	**(PK) DepositNum**	Number	NA	Yes, Primary Key	Yes
	DepositDate	Date	NA	No	Yes
	Amount	Number	12	No	Yes (0.00)
	(FK) JobNum	Number	NA	Yes, Foreign Key	Yes
Payments	**(PK) PaymentNum**	Number	NA	Yes, Primary Key	Yes
	DepositDate	Date	NA	No	Yes
	Amount	Number	12	No	Yes (0.00)
	(FK) JobNum	Number	NA	Yes, Foreign Key	Yes
RepairJob Repairmen	**(PK) RepJobMenId**	Number	NA	Yes, Primary Key	Yes
	(FK) JobNum	Number	NA	Yes, Foreign Key	Yes
	(FK) RepairmenId				
	DateStarted	Date	NA	No	No
	DateEnded				
	Comments	Character	5,000	No	No
RepairmenItems	**(PK) RepmenItemId**	Number	NA	Yes, Primary Key	Yes
	(FK) ItemId	Number	NA	Yes, Foreign Key	Yes
	(FK) RepairmenId				
	DateOrdered	Date	NA	No	No
	Quantity	Number	6	No	Yes (1)
	TotalCost	Number	12	No	Yes (0.00)
RepairJobItems	**(PK) RepJobItemId**	Number	NA	Yes, Primary Key	Yes
	(FK) JobNum	Number	NA	Yes, Foreign Key	Yes
	(FK) ItemId				
	DateUsed	Date	NA	No	No
	Quantity	Number	6	No	Yes (1)
	TotalCost	Number	12	No	Yes (0.00)

SQL Commands

The SQL commands to implement the design (create the tables) for Case Study 2 in a MySQL database are given overleaf:

```
CREATE TABLE Items (
ItemId int(11) NOT NULL AUTO_INCREMENT,
PartNum varchar(50) NOT NULL,
ShortName varchar(75) NOT NULL,
Cost decimal(10,2) NOT NULL DEFAULT '0.00',
NumInStock smallint(6) NOT NULL DEFAULT 1,
PRIMARY KEY (ItemId),
INDEX PartNum (PartNum)
);

CREATE TABLE Customers (
CustomerId int(11) NOT NULL AUTO_INCREMENT,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MI varchar(1) DEFAULT NULL,
Email varchar(75) NOT NULL,
Mobile varchar(14) DEFAULT NULL,
HTel varchar(14) NOT NULL,
AddressLine1 varchar(75) NOT NULL,
AddressLine2 varchar(75) DEFAULT NULL,
City varchar(50) NOT NULL,
State varchar(50) NOT NULL,
PostCode varchar(10) NOT NULL,
PRIMARY KEY (CustomerId),
INDEX PostCode (PostCode),
INDEX FullName (LastName,FirstName)
);

CREATE TABLE Repairmen (
RepairmenId int(11) NOT NULL AUTO_INCREMENT,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MI varchar(1) DEFAULT NULL,
Email varchar(75) NOT NULL,
Mobile varchar(14) NOT NULL,
HTel varchar(14) DEFAULT NULL,
PRIMARY KEY (RepairmenId),
INDEX FullName (LastName,FirstName)
);

CREATE TABLE Computers (
ComputerId int(11) NOT NULL AUTO_INCREMENT,
SerialNum varchar(50) NOT NULL,
Make varchar(50) DEFAULT NULL,
Model varchar(50) DEFAULT NULL,
ComputerDescription varchar(250) DEFAULT NULL,
CustomerId int(11) NOT NULL,
PRIMARY KEY (ComputerId),
FOREIGN KEY CustomerId (CustomerId)
    REFERENCES Customers (CustomerId)
    ON UPDATE CASCADE ON DELETE RESTRICT,
INDEX SerialNum (SerialNum)
);

CREATE TABLE Items (
ItemId int(11) NOT NULL AUTO_INCREMENT,
PartNum varchar(50) NOT NULL,
ShortName varchar(75) NOT NULL,
Cost decimal(10,2) NOT NULL DEFAULT '0.00',
NumInStock smallint(6) NOT NULL DEFAULT 1,
PRIMARY KEY (ItemId),
INDEX PartNum (PartNum)
```

```
);

CREATE TABLE Customers (
CustomerId int(11) NOT NULL AUTO_INCREMENT,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MI varchar(1) DEFAULT NULL,
Email varchar(75) NOT NULL,
Mobile varchar(14) DEFAULT NULL,
HTel varchar(14) NOT NULL,
AddressLine1 varchar(75) NOT NULL,
AddressLine2 varchar(75) DEFAULT NULL,
City varchar(50) NOT NULL,
State varchar(50) NOT NULL,
PostCode varchar(10) NOT NULL,
PRIMARY KEY (CustomerId),
INDEX PostCode (PostCode),
INDEX FullName (LastName,FirstName)
);

CREATE TABLE Repairmen (
RepairmenId int(11) NOT NULL AUTO_INCREMENT,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MI varchar(1) DEFAULT NULL,
Email varchar(75) NOT NULL,
Mobile varchar(14) NOT NULL,
HTel varchar(14) DEFAULT NULL,
PRIMARY KEY (RepairmenId),
INDEX FullName (LastName,FirstName)
);

CREATE TABLE Computers (
ComputerId int(11) NOT NULL AUTO_INCREMENT,
SerialNum varchar(50) NOT NULL,
Make varchar(50) DEFAULT NULL,
Model varchar(50) DEFAULT NULL,
ComputerDescription varchar(250) DEFAULT NULL,
CustomerId int(11) NOT NULL,
PRIMARY KEY (ComputerId),
FOREIGN KEY CustomerId (CustomerId) REFERENCES Customers (CustomerId) ON UPDATE
CASCADE ON DELETE RESTRICT,
INDEX SerialNum (SerialNum)
);

CREATE TABLE RepairJobs (
JobNum int(11) NOT NULL AUTO_INCREMENT,
DateReceived date NOT NULL,
DateReturned date NOT NULL,
DateEnded date DEFAULT NULL,
LabourCost decimal(10,2) NOT NULL DEFAULT '0.00',
TotalCost decimal(10,2) NOT NULL DEFAULT '0.00',
CustomerId int(11) NOT NULL,
ComputerId int(11) NOT NULL,
PRIMARY KEY (JobNum),
FOREIGN KEY CustomerId (CustomerId) REFERENCES Customers (CustomerId) ON UPDATE
CASCADE ON DELETE RESTRICT,
FOREIGN KEY ComputerId (ComputerId) REFERENCES Computers (ComputerId) ON UPDATE
CASCADE ON DELETE RESTRICT,
INDEX DateReceived (DateReceived),
INDEX DateReturned (DateReturned)
);

CREATE TABLE Deposits (
DepositNum int(11) NOT NULL AUTO_INCREMENT,
DepositDate date NOT NULL,
Amount decimal(10,2) NOT NULL DEFAULT '0.00',
JobNum int(11) NOT NULL,
PRIMARY KEY (DepositNum),
FOREIGN KEY JobNum (JobNum) REFERENCES RepairJobs (JobNum) ON UPDATE CASCADE ON
DELETE RESTRICT
);

CREATE TABLE Payments (
PaymentNum int(11) NOT NULL AUTO_INCREMENT,
DepositDate date NOT NULL,
Amount decimal(10,2) NOT NULL DEFAULT '0.00',
JobNum int(11) NOT NULL,
PRIMARY KEY (PaymentNum),
FOREIGN KEY JobNum (JobNum) REFERENCES RepairJobs (JobNum) ON UPDATE CASCADE ON
DELETE RESTRICT
);

CREATE TABLE RepairJobRepairmen (
RepJobMenId int(11) NOT NULL AUTO_INCREMENT,
DateStarted date DEFAULT NULL,
DateEnded date DEFAULT NULL,
Comments text,
JobNum int(11) NOT NULL,
RepairmenId int(11) NOT NULL,
PRIMARY KEY (RepJobMenId),
FOREIGN KEY JobNum (JobNum) REFERENCES RepairJobs (JobNum) ON UPDATE CASCADE ON
DELETE RESTRICT,
FOREIGN KEY RepairmenId (RepairmenId) REFERENCES Repairmen (RepairmenId) ON UPDATE
CASCADE ON DELETE RESTRICT
);

CREATE TABLE RepairmenItems (
RepmenItemId int(11) NOT NULL AUTO_INCREMENT,
DateOrdered date DEFAULT NULL,
Quantity smallint(6) NOT NULL DEFAULT 1,
TotalCost decimal(10,2) NOT NULL DEFAULT '0.00',
ItemId int(11) NOT NULL,
RepairmenId int(11) NOT NULL,
PRIMARY KEY (RepmenItemId),
FOREIGN KEY ItemId (ItemId) REFERENCES Items (ItemId) ON UPDATE CASCADE ON DELETE
RESTRICT,
FOREIGN KEY RepairmenId (RepairmenId) REFERENCES Repairmen (RepairmenId) ON UPDATE
CASCADE ON DELETE RESTRICT
);
```

```
CREATE TABLE RepairJobItems (
RepJobItemId int(11) NOT NULL AUTO_INCREMENT,
DateUsed date DEFAULT NULL,
Quantity smallint(6) NOT NULL DEFAULT 1,
TotalCost decimal(10,2) NOT NULL DEFAULT '0.00',
JobNum int(11) NOT NULL,
ItemId int(11) NOT NULL,
PRIMARY KEY (RepJobItemId),
FOREIGN KEY JobNum (JobNum) REFERENCES RepairJobs (JobNum) ON UPDATE CASCADE ON
DELETE RESTRICT,
FOREIGN KEY ItemId (ItemId) REFERENCES Items (ItemId) ON UPDATE CASCADE ON DELETE
RESTRICT
);
```

Notice the order in which the tables were created. The tables with no foreign keys were created first and the many-to-many join tables were created last. Also, the table with the primary key for the referenced foreign key is always created before the table with the foreign key.

There was no need to create indexes for foreign keys because MySQL automatically does this for *InnoDB* tables, which is the default storage engine for MySQL 5.7.

Remember: Always have the Reference Manual of the RDBMS handy when implementing your design.

Case Study 3

The R-M diagram in figure 9.6 can be implemented on any RDBMS. The SQL commands on the subsequent pages can be used to implement the design (create the tables) in a MySQL database. Similar SQL commands can be used to create the tables on any RDBMS.

Entities, Attributes, and the R-M Diagram

The entities, attributes and R-M diagram for Case Study 3 are given below:

Students	**Courses**	**ScheduledClasses**	**Semesters**
(PK) StudentId	**(PK) CourseId**	**(PK) ScheduleId**	**(PK) SemesterId**
SSNumber	**CourseCode**	**ScheduleCode**	SemesterNumber
LastName	**ShortName**	Section	SemesterName
FirstName	LongName	Day	SemesterYear
MiddleName	CourseDescription	Time	StartDate
Gender		Location	EndDate
DOB			
Email			
Mobile			
HTel			
WTel			
AddressLine1			
AddressLine2			
City			
State			
PostCode			

Lecturers	Users	AccessLevels	LogEntries
(PK) LecturerId	**(PK) UserId**	**(PK) AccessLevelId**	**(PK) LogEntryId**
SSNumber	**Login**	**AccessLevelCode**	LoggedOn
LastName	UserName	**ShortName**	LoggedOff
FirstName	Password	LongName	
MiddleName	Active	AccessLevelDescription	
Gender			
Email			
Mobile			
HTel			
WTel			
About			

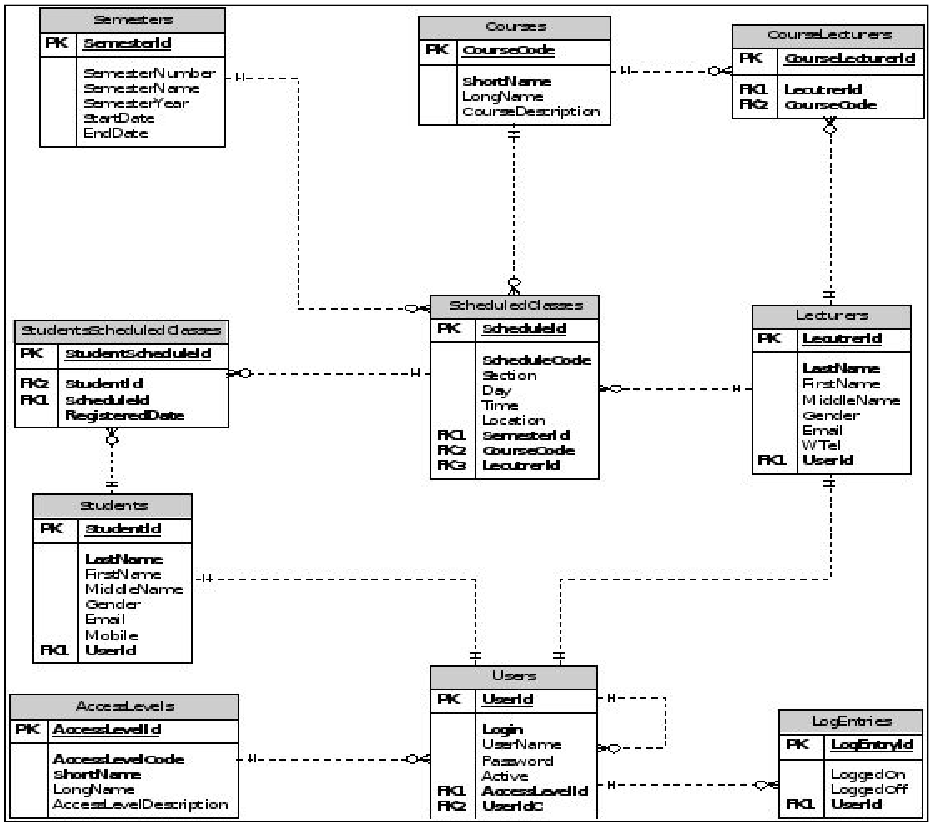

Figure 9.6 – Crow’s Foot R-M diagram for Case Study 3 (Part 1)

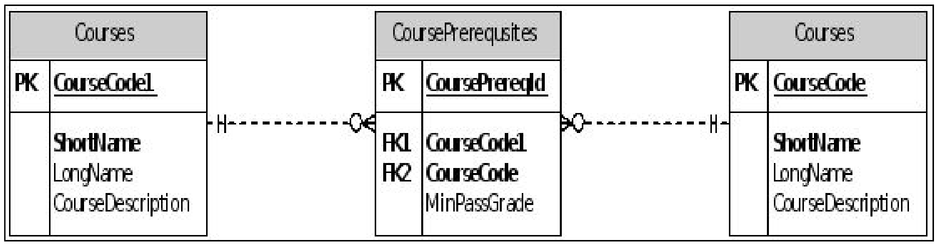

Figure 9.7 – Crow’s Foot R-M diagram for Case Study 3 (Part 2)

Data Dictionary

The table below gives a breakdown of the possible data types along with the other important considerations that need to be taken into account when implementing the columns of the tables for Case Study 3:

Table	**Column**	**Recommend Data Type**	**Length**	**Indexed**	**Required (Default)**
Courses	**(PK) CourseId**	Number	NA	Yes, Primary Key	Yes
	CourseCode	Character	6	Yes	Yes
	ShortName	Character	75	Yes	Yes
	LongName	Character	150	No	No
	CourseDescription	Character	5,000	No	No
Semesters	**(PK) SemesterId**	Number	NA	Yes, Primary Key	Yes
	SemesterNumber	Number	6	No	Yes (1)
	SemesterName	Character	12	No	Yes
	SemesterYear	Date or Number	4	No	Yes
	StartDate	Date	NA	No	Yes
	EndDate				
AccessLevels	**(PK) AccessLevelId**	Number	NA	Yes, Primary Key	Yes
	AccessLevelCode	Character	6	Yes	Yes
	ShortName	Character	35	Yes	Yes
	LongName	Character	75	No	No
	AccessLevel Description	Character	1,000	No	No
Users	**(PK) UserId**	Number	NA	Yes, Primary Key	Yes
	Login	Character	35	Yes	Yes
	UserName	Character	50	No	Yes
	Password	Character	50	No	Yes
	Active	Number	6	No	Yes (1)
	(FK) AccessLevelId	Number	NA	Yes, Foreign Key	Yes
	(FK) UserIdC				
Table	**Column**	**Recommend Data Type**	**Length**	**Indexed**	**Required (Default)**
Students	**(PK) StudentId**	Number	NA	Yes, Primary Key	Yes
	SSNumber	Character	11	Yes	Yes
	LastName	Character	50	Yes, Composite	Yes
	FirstName		50		
	MiddleName		50	No	No
	Gender	Character	6	No	Yes (‘MALE’)
	DOB	Date	NA	Yes	Yes
	Email	Character	75	No	Yes
	Mobile	Character	14	No	No

	HTel				Yes
	AddressLine1	Character	75	No	Yes
	AddressLine2				No
	City	Character	50	No	Yes
	State				
	PostCode	Character	10	Yes	Yes
	(FK) UserId	Number	NA	Yes, Foreign Key	Yes
Lecturers	**(PK) LecturerId**	Number	NA	Yes, Primary Key	Yes
	LastName	Character	50	Yes, Composite	Yes
	FirstName		50		
	MiddleName		50	No	No
	Gender	Character	6	No	Yes ('MALE')
	Email	Character	75	No	Yes
	WTel	Character	14	No	No
	(FK) UserId	Number	NA	Yes, Foreign Key	Yes
LogEntries	**(PK) LogEntryId**	Number	NA	Yes, Primary Key	Yes
	LoggedOn	Date and Time	NA	Yes	Yes
	LoggedOff			No	No
	(FK) UserId	Number	NA	Yes, Foreign Key	Yes
Scheduled Classes	**(PK) ScheduleId**	Number	NA	Yes, Primary Key	Yes
	ScheduleCode	Character	8	Yes	Yes
	Section	Character	1	No	Yes
	Day	Character	8	No	Yes
	Time	Date	NA	No	Yes
	Location	Character	75	No	No
	(FK) SemesterId	Number	NA	Yes, Foreign Key	Yes
	(FK) CourseId*				
	(FK) LecturerId				
Course Prerequsites	**(PK) CoursePrereqId**	Number	NA	Yes, Primary Key	
	(FK) CourseId*	Number	NA	Yes, Foreign Key	Yes
	(FK) CourseId1*				
	MinPassGrade	Character	2	No	No
Table	**Column**	**Recommend Data Type**	**Length**	**Indexed**	**Required (Default)**
CourseLecturers	**(PK) CourseLecturer Id**	Number	NA	Yes, Primary Key	Yes
	(FK) CourseId*	Number	NA	Yes, Foreign Key	Yes
	(FK) LecturerId				
StudentsScheduledClasses	**(PK) StudentScheduleId**	Number	NA	Yes, Primary Key	Yes
	(FK) StudentId	Number	NA	Yes, Foreign Key	Yes
	(FK) ScheduleId				
	RegisteredDate	Date	NA	Yes	Yes

**CourseCode* has been replaced by *CourseId* as per recommendation in the previous section on indexes.

SQL Commands

The SQL commands to implement the design (create the tables) for Case Study 3 in a MySQL database are given below:

```
CREATE TABLE Courses (
CourseId int(11) NOT NULL AUTO_INCREMENT,
CourseCode varchar(6) NOT NULL,
ShortName varchar(75) NOT NULL,
LongName varchar(150) DEFAULT NULL,
CourseDescription text,
PRIMARY KEY (CourseId),
INDEX CourseCode (CourseCode),
INDEX ShortName (ShortName)
);

CREATE TABLE Semesters (
SemesterId int(11) NOT NULL AUTO_INCREMENT,
SemesterNumber smallint(6) NOT NULL DEFAULT 1,
SemesterName varchar(12) NOT NULL,
SemesterYear year(4) NOT NULL,
StartDate date NOT NULL,
EndDate date NOT NULL,
PRIMARY KEY (SemesterId)
);

CREATE TABLE AccessLevels (
AccessLevelId int(11) NOT NULL AUTO_INCREMENT,
AccessLevelCode varchar(6) NOT NULL,
ShortName varchar(35) NOT NULL,
LongName varchar(75) DEFAULT NULL,
AccessLevelDescription varchar(1000) DEFAULT NULL,
PRIMARY KEY (AccessLevelId),
INDEX AccessLevelCode (AccessLevelCode),
INDEX ShortName (ShortName)
);

CREATE TABLE Users (
UserId int(11) NOT NULL AUTO_INCREMENT,
Login varchar(35) NOT NULL,
UserName varchar(50) NOT NULL,
Password varchar(50) NOT NULL,
Active smallint(6) NOT NULL DEFAULT '1',
AccessLevelId int(11) NOT NULL,
UserIdC int(11) NOT NULL,
PRIMARY KEY (UserId),
FOREIGN KEY AccessLevelId (AccessLevelId) REFERENCES AccessLevels (AccessLevelId) ON
UPDATE CASCADE ON DELETE RESTRICT,
FOREIGN KEY UserIdC (UserIdC) REFERENCES Users (UserId) ON UPDATE CASCADE ON DELETE
RESTRICT,
UNIQUE INDEX Login (Login)
);

CREATE TABLE Students (
StudentId int(11) NOT NULL AUTO_INCREMENT,
SSNumber varchar(11) NOT NULL,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MiddleName varchar(50) DEFAULT NULL,
Gender varchar(6) NOT NULL DEFAULT 'MALE',
DOB date NOT NULL,
Email varchar(75) NOT NULL,
Mobile varchar(14) DEFAULT NULL,
HTel varchar(14) NOT NULL,
AddressLine1 varchar(75) NOT NULL,
AddressLine2 varchar(75) DEFAULT NULL,
City varchar(50) NOT NULL,
State varchar(50) NOT NULL,
PostCode varchar(10) NOT NULL,
UserId int(11) NOT NULL,
PRIMARY KEY (StudentId),
FOREIGN KEY UserId (UserId) REFERENCES Users (UserId) ON UPDATE CASCADE ON DELETE
RESTRICT,
INDEX SSNumber (SSNumber),
INDEX FullName (LastName,FirstName),
INDEX DOB (DOB),
INDEX PostCode (PostCode)
);

CREATE TABLE Lecturers (
LecturerId int(11) NOT NULL AUTO_INCREMENT,
LastName varchar(50) NOT NULL,
FirstName varchar(50) NOT NULL,
MiddleName varchar(50) DEFAULT NULL,
Gender varchar(6) NOT NULL DEFAULT 'MALE',
Email varchar(75) NOT NULL,
WTel varchar(14) DEFAULT NULL,
UserId int(11) NOT NULL,
PRIMARY KEY (LecturerId),
FOREIGN KEY UserId (UserId) REFERENCES Users (UserId) ON UPDATE CASCADE ON DELETE
RESTRICT,
INDEX FullName (LastName,FirstName)
);

CREATE TABLE LogEntries (
LogEntryId bigint(20) NOT NULL AUTO_INCREMENT,
LoggedOn datetime NOT NULL,
LoggedOff datetime DEFAULT NULL,
UserId int(11) NOT NULL,
PRIMARY KEY (LogEntryId),
FOREIGN KEY UserId (UserId) REFERENCES Users (UserId) ON UPDATE CASCADE ON DELETE
RESTRICT
);

CREATE TABLE ScheduledClasses (
ScheduleId bigint(20) NOT NULL AUTO_INCREMENT,
ScheduleCode varchar(8) NOT NULL,
Section varchar(1) NOT NULL,
Day varchar(8) NOT NULL,
Time time NOT NULL,
Location varchar(75) DEFAULT NULL,
SemesterId int(11) NOT NULL,
CourseId int(11) NOT NULL,
LecturerId int(11) NOT NULL,
PRIMARY KEY (ScheduleId),
FOREIGN KEY SemesterId (SemesterId) REFERENCES Semesters (SemesterId) ON UPDATE
CASCADE ON DELETE RESTRICT,
FOREIGN KEY CourseId (CourseId) REFERENCES Courses (CourseId) ON UPDATE CASCADE ON
DELETE RESTRICT,
FOREIGN KEY LecturerId (LecturerId) REFERENCES Lecturers (LecturerId) ON UPDATE CASCADE
ON DELETE RESTRICT,
INDEX ScheduleCode (ScheduleCode)
);

CREATE TABLE CoursePrerequsites (
CoursePrereqId int(11) NOT NULL AUTO_INCREMENT,
MinPassGrade varchar(2) DEFAULT NULL,
CourseId int(11) NOT NULL,
CourseId1 int(11) NOT NULL,
PRIMARY KEY (CoursePrereqId),
FOREIGN KEY CourseId (CourseId) REFERENCES Courses (CourseId) ON UPDATE CASCADE ON
DELETE RESTRICT,
FOREIGN KEY CourseId1 (CourseId1) REFERENCES Courses (CourseId) ON UPDATE CASCADE ON
DELETE RESTRICT
);

CREATE TABLE CourseLecturers (
CourseLecturerId int(11) NOT NULL AUTO_INCREMENT,
CourseId int(11) NOT NULL,
LecturerId int(11) NOT NULL,
PRIMARY KEY (CourseLecturerId),
FOREIGN KEY CourseId (CourseId) REFERENCES Courses (CourseId) ON UPDATE CASCADE ON
DELETE RESTRICT,
FOREIGN KEY LecturerId (LecturerId) REFERENCES Lecturers (LecturerId) ON UPDATE CASCADE
ON DELETE RESTRICT
);

CREATE TABLE StudentsScheduledClasses (
StudentScheduleId int(11) NOT NULL AUTO_INCREMENT,
RegisteredDate datetime NOT NULL,
StudentId int(11) NOT NULL,
ScheduleId bigint(20) NOT NULL,
PRIMARY KEY (StudentScheduleId),
FOREIGN KEY StudentId (StudentId) REFERENCES Students (StudentId) ON UPDATE CASCADE ON
DELETE RESTRICT,
FOREIGN KEY ScheduleId (ScheduleId) REFERENCES ScheduledClasses (ScheduleId) ON UPDATE
CASCADE ON DELETE RESTRICT,
INDEX RegisteredDate (RegisteredDate)
);
```

Notice the order in which the tables were created. The tables with no foreign keys were created first and the many-to-many join tables were created last. Also, the table with the primary key for the referenced foreign key is always created before the table with the foreign key.

There was no need to create indexes for foreign keys because MySQL automatically does this for *InnoDB* tables, which is the default storage engine for MySQL 5.7.

Remember: Always have the Reference Manual of the RDBMS handy when implementing your design.

Summary

This chapter discusses the considerations that need to be taken into account when implementing a relational database.

RDBMSs

A relational database management system (RDBMS) is an administration and management system for relational databases. The role of the RDBMS is to control access to the files holding the tables' data, ensuring that they are accessed in a manner that preserves the integrity of the data in those files.

Transactions and ACID compliance

A database transaction is a logical unit of *database operations* that are executed as one, all together, or none at all. Each database transaction expects to find the database in a consistent and reliable state before it is processed and must leave the database in a consistent and reliable state for the next transaction after it has been processed.

Database transactions must be **A**tomic, **C**onsistent, **I**solated, and **D**urable (ACID). ACID compliance guarantees the integrity of the data in the tables of an RDBMS database each time a database transaction is processed:

Normalization

Normalization is the process of organizing the Relations (tables) in a database so that they reduce data redundancy and prevent inconsistent data dependencies. There are at least three normal forms associated with normalization: first normal form (1NF), second normal form (2NF), and third normal form (3NF).

Any Relation (table) that is in first normal form exhibits *domain integrity*, and domain integrity means that all the possible values of an attribute are legitimate ones. Any table that is in second normal form exhibits *entity integrity* and is also, by definition, in first normal form. Entity integrity ensures that each row in the table is a unique and genuine entity, and that every other column in that table depends solely on the primary key. Any table that is in third normal form exhibits *referential integrity* and is also, by definition, in first and second normal forms. Referential integrity ensures that all references to values in columns in other tables are authentic – authentic foreign key values.

Indexes

An index is a data structure that is created for a database table but is external to that table, and if well selected and implemented can result in faster searches for data in that table.

Primary keys and foreign keys play an important role in storing, organizing, and finding data stored in these databases. As a result, primary keys and foreign keys should *always* be indexed.

Data types

Each column in a table must specify the type of data that it holds, which is referred to as the *data type* of that column. The type of data that a column holds is specified when the table is being created and generally fall into four main categories: Character, numeric, date and binary data.

Remember: Always have the Reference Manual of the RDBMS handy when implementing your design.

Review Questions

1. What is an RDBMS and why is it important?

2. What is a transaction and why is it important?

3. With respect to transactions, explain what ACID compliance means.

4. What is normalization and why is it important?

5. What is an index and why is it important?

6. State six (6) general guidelines for implementing indexes.

7. What are the four (4) main categories of the type of data that is likely to be stored in a table’s column?

8. Discuss three (3) considerations that need to be taken into account when implementing a relational database.

9. Using appropriate examples, explain what it means for a database to be in first, second, and third normal forms.

10. Using an appropriate example, explain how cascade updates works.

11. Using an appropriate example, explain how cascade deletes works.

12. Using appropriate examples, explain why it is important to have the Reference Manual of the RDBMS handy when implementing your database’s design.

Exercises

1. Modify the SQL commands given in the chapter for the case studies so that they can be implemented on MS Access. You will need to research the equivalent data types used by MS Access.

2. Modify the SQL commands given in the chapter for the case studies so that they can be implemented on MS SQL Server. You will need to research the equivalent data types used by MS SQL Server.

3. Modify the SQL commands given in the chapter for the case studies so that they can be implemented on Oracle. You will need to research the equivalent data types used by Oracle.

References

Bailis, P. (2013) *When is "ACID" ACID? Rarely.* Retrieved April 2, 2013, from Peter Bailis :: Highly Available, Seldom Consistent: http://www.bailis.org/blog/when-is-acid-acid-rarely/

Buytaert, D. (2012). *The history of MySQL AB.* Retrieved January 9, 2012, from The history of MySQL AB: http://buytaert.net/the-history-of-mysql-ab

Chen, P. (1976, March). The Entity-Relationship Model-Toward a Unified View of Data. *Transactions on Database Systems, 1*(1), 9-36.

Codd, E. F. (1970, June). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM, 13*(6), 377-387.

CRC SOGEMA. (1998). *Data Modelling.*

Delaney, K. (2000). *History of SQL Server.* Retrieved January 8, 2012, from Kalen Delaney | SQL Server Resources: http://insidesqlserver.com/companion/History%20of%20SQL%20Server.pdf

Favero, W. (2008, January 27). *DB2 History 101: Version 1.1*. Retrieved January 8, 2012, from Toolbox.com: http://it.toolbox.com/blogs/db2zos/db2-history-101-version-11-22046

IBM. (2011). *IBM100 - Relational Database*. Retrieved January 8, 2011, from IBM100 - IBM at 100: http://www.ibm.com/ibm100/us/en/icons/reldb/

IBM. (n.d.) *Types of indexes.* Retrieved April 3, 2013, from DB2 Version 9.5 for Linux, UNIX, and Windows | Database Fundamentals: http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0020180.html

Kent, W. (1983, February). A Simple Guide to Five Normal Forms in Relational Database Theory. *Communications of the ACM*, 120-125.

Kroenke, D. M. (2006). *Database Processing - Fundamentals, Design, and Implementation.* Upper Saddle River, New Jersey, USA: Pearson Prentice Hall.

Litwin, P. (1994). *Fundamentals of Relational Database Design.* Retrieved November 01, 2011, from Deep Training:

http://www.deeptraining.com/litwin/dbdesign/FundamentalsOfRelationalDatabaseDesign.aspx

McGee, W. C. (1981, September). Database Technology. *Journal of Research and Development, 21*(5).

McJones, P. (n.d.). *Multics Relational Data Store (MRDS)*. Retrieved January 8, 2012, from Paul McJones: http://www.mcjones.org/System_R/mrds.html

Microsoft. (2013) *Description of the database normalization basics.* Retrieved April 3, 2013, from Microsoft: http://support.microsoft.com/kb/283878

MSDN. (2012) *Data Types (Transact-SQL).* Retrieved April 3, 2013, from MSDN: http://msdn.microsoft.com/en-us/library/ms187752.aspx

MSDN. (2012) *What is a Transaction?.* Retrieved April 3, 2013, from MSDN: http://msdn.microsoft.com/en-us/library/aa366402%28VS.85%29.aspx

Multics. (2011, September). *Multics History*. Retrieved January 8, 2012, from Multics: http://www.multicians.org/history.html

Network Dictionary. (2011). *Database Systems: Assertions*. Retrieved February 7, 2012, from www.NetworkDictionary.com: http://www.networkdictionary.com/Software/Assertions.php

Oracle. (2007, May). Profit Magazine. *The Executive's Guide to Oracle Applications*, pp. 26-33.

Oracle (n.d.) *Indexes and Index-Organized Tables.* Retrieved April 3, 2013, from Oracle: http://docs.oracle.com/cd/E11882_01/server.112/e10713/indexiot.htm

Index