UNIVERSITY
OF LONDON

Database systems: Volume 1

D. Lewis

CO2209

2016

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

D. Lewis, Department of Computing, Goldsmiths, University of London.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the author is unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

In this and other publications you may see references to the 'University of London International Programmes', which was the name of the University's flexible and distance learning arm until 2018. It is now known simply as the 'University of London', which better reflects the academic award our students are working towards. The change in name will be incorporated into our materials as they are revised.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

Contents

Chapter 1: Introduction to the subject guide

Introduction to Volumes 1 and 2 Database systems

Welcome to this course in **Database systems**, which is divided into two parts, Volume 1 and Volume 2. In this Introductory chapter we will look at the overall structure of the subject guide – in the form of a Route map – and introduce you to the subject, to the aims and learning outcomes, and to the learning resources available. We will also offer you some examination advice.

We hope you enjoy this subject and we wish you good luck with your studies.

1.1 Route map to the guide

Volume 1 of this subject guide deals with the relational model in theory and practice. There are five chapters in total in Volume 1; one Introductory chapter, and four main content chapters, which are described below.

Chapter 1, the Introductory chapter, introduces the subject and includes some general information about reading, learning resources, as well as some examination advice.

Chapter 2 introduces the basic concepts of database systems. It focuses on describing the **components** of a database environment, a prevalent **architecture** of a database environment and the concept of **data model**.

Chapter 3 presents the theory behind relational database systems – **the relational model**. It describes the relational data objects (**domains** and **relations**), relational operators (**relational algebra**), and issues about relational data integrity (**keys**). The theoretical description of each of the components of the relational model is accompanied by a description of the operational issues – the way relational concepts are implemented in a database management system (**DBMS**).

Chapter 4 of this subject guide is dedicated to the introduction of a relational database language, namely **ANSI SQL**. The chapter also details some of the differences between versions of the standard and their various implementations. This chapter is accompanied by activities and coursework aimed at developing **practical skills** in programming in SQL.

Chapter 5 describes the process of **database design**. It focuses on the **conceptual design** phase, by presenting the most popular conceptual model – the **entity-relationship (E/R) model** – and on the **logical design** phase, by presenting the **normal forms** associated with the relational model.

At the end of Volume 1, an Appendix contains some sample answers to the end of chapter Sample examination questions.

Volume 2 of this subject guide considers more advanced topics in database systems, in terms of both relational database management systems and alternative models.

An Introductory chapter appears as **Chapter 1** in Volume 2.

Chapter 2 is dedicated to the pragmatic considerations around data preservation, security and database optimisation in a relational DBMS. This chapter introduces the concepts of a **transaction** and **transaction processing** and then describes the way transaction processing can be employed in

database **recovery** and **concurrency control**. As another facet of data protection, the issue of **security** in a database environment is also presented. Each of these subtopics is supported by examples of ANSI SQL approaches to them. The second part of this chapter considers the concept of optimisation, particularly indexing strategies.

Chapter 3 of Volume 2 introduces **distributed architecture** models for database systems. It presents the objectives of distributed database systems development and the problems generated by this architecture.

Chapter 4 is dedicated to newer approaches to database systems development. The chapter starts with a consideration of some drawbacks of the relational model, and therefore of the motivations for exploring alternatives. The focus then shifts to newer approaches to database systems: firstly considering **deductive databases** and **object oriented (OO) databases***;* and then moving on to newer, web-scale approaches, such as **NoSQL** databases, and **triplestores** for **semantic web** data.

Volume 2 also contains appendices with answers to the end of chapter Sample examination questions. It also contains an appendix with a dataset.

The easiest way to understand database systems is by practical experiment on a live system. To help you to get started, we provide a dataset on the VLE that you can import into the database system of your choice. The appendix to Volume 2 lists the data so you also have the option of entering it manually. The structure and data of this dataset is also referred to in examples within this subject guide, in particular those in Chapter 4 of Volume 1.

1.1.1 Glossary of key terms

These two volumes of the subject guides introduce vocabulary, concepts and skills that you will need to pass the examination at the end of the course. At the end of each chapter, a **Learning outcomes** section lists what you should be able to do and **what key terms you should know** as a result of reading the chapter and engaging with the activities. You should use these lists to check your understanding of the chapter and during revision.

1.2 Introduction to the subject area

Every information system has a **database** at its core. That makes an understanding of **database systems** an essential skill for a computer scientist or information technologist. During the past 50 years – since its inception – the area of database systems has matured into a well-established, conventional subject. However, new ideas are continuously developed and new challenges and issues constantly appear. These need to be addressed from both a theoretical and practical standpoint. This subject can be seen as consisting of a 'classic' and a 'young' component. The former is based almost entirely on the relational model and at the time of writing still represents, de facto, the standard approach of most industrial applications. The latter proposes new models and architectures for database systems, and forms a growing part of internet-based uses.

1.3 Syllabus

Introduction to Database Systems (motivation for database systems, storage systems, architecture, facilities, applications). Database modelling (basic concepts, E-R modelling, Schema deviation). The relational model and algebra, SQL (definitions, manipulations, access centre, embedding). Physical

design (estimation of workload and access time, logical I/Os, distribution). Modern database systems (extended relational, object oriented). Advanced database systems (active, deductive, parallel, distributed, federated). DB functionality and services (files, structures and access methods, transactions and concurrency control, reliability, query processing).

1.4 Aims of this course

This course aims to provide you with an understanding of the main issues related to data storage and manipulation – the object of database systems. In particular, this course is aimed at the detailed presentation of the theory and practice of the relational model, on one side, and at the introduction of the emerging trends in database systems, on the other. You will also gain practical experience in a database language – ANSI SQL – and in developing relational database systems, through the activities and coursework assignments that you will undertake.

1.5 Learning objectives for the course

The learning objectives of this course are to:

- explain the need for **database systems**
- provide a general description of a **database environment**
- describe the **relational model** thoroughly, as the underlying framework of most industrial database management systems
- introduce a **relational database language**: ANSI SQL
- describe the issues pertaining to database design, focusing on **conceptual design** (through **E/R modelling**) and **logical design** (through **normalisation**)
- develop **practical skills** in designing and implementing a relational database
- present issues pertaining to **security**, **recovery** and **concurrency control**
- introduce **distributed architectures** for database systems
- describe **newer approaches** to database systems that differ significantly from the relational model
- describe links between database and web technologies.

1.6 Learning outcomes for students

By the end of this course, and having completed the Essential readings and activities, you should be able to:

- understand the main issues relating to database systems in general
- have detailed knowledge about the relational model
- analyse a specific problem, synthesise the requirements and accordingly perform a top down design for a corresponding (relational) database
- have the knowledge and practical skills to implement and maintain a relational database (in SQL)
- be aware of alternative models of and approaches to database systems, particularly web-scale systems, and also including object databases, deductive and knowledge-based systems, etc.
- have more in-depth knowledge in one or more of the previously mentioned trends.

1.7 Overview of learning resources

1.7.1 The subject guide

Each chapter in this guide starts with a list of **Essential reading** and possibly some **Further reading**.

The **Essential reading** section directs you to the textbook sections (or journals) that you have to read. For each chapter you are given an alternative choice between Date **and/or** Connolly and Begg. If you can, you are advised to read the recommended readings for **both** textbooks.

The **Further reading** section provides pointers towards relevant literature for the presented topic. This is not compulsory reading. A short descriptive note is associated with each pointer.

1.7.2 Essential reading

The following two books are recommended to support this course:

Date, C.J. *An introduction to database systems*. (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)].

and

Connolly, T. and C.E. Begg *Database systems: a practical approach to design implementation and management*. (Pearson Education, 2014) 6th edition, global edition [ISBN 9781292061184 (pbk)].

Both of these texts have been through multiple editions. Newer editions are preferable, since they will describe more recent developments, but they may be more expensive to buy. Those listed above are the latest at the time this subject guide was written. Any edition from 1999 or after will be adequate for this course – this includes the 7th edition or later of Date and the 2nd edition or later of Connolly and Begg. Earlier versions than this should be approached with caution. Since these books do not cover **exactly** the same topics, you should use both of them together if possible.

Detailed reading references in this subject guide refer to the editions of the set textbooks listed above. New editions of one or more of these textbooks may have been published by the time you study this course. You can use a more recent edition of any of the books; use the detailed chapter and section headings and the index to identify relevant readings. Also check the VLE regularly for updated guidance on readings.

1.7.3 Further reading

Please note that as long as you read the Essential reading you are then free to read around the subject area in any text, paper or online resource. You will need to support your learning by reading as widely as possible and by thinking about how these principles apply in the real world. To help you read extensively, you have free access to the virtual learning environment (VLE) and University of London Online Library (see below).

Other useful texts for this course include:

Books

Elmasri, R. and S.B. Navathe *Fundamentals of database systems*. (Pearson, 2016) 7th edition [ISBN 9781292097619 (pbk)].

Ramakrishnan, R. and J. Gehrke *Database management systems*. (McGraw-Hill, 2002) 3rd edition [ISBN 9780072465631 (hbk); 9780071231510 (pbk)].

Stevens, P. and R. Pooley *Using UML: software engineering with objects and components*. (Addison-Wesley, 2006) 2nd edition [ISBN 9780321269676].

Articles

Chen, P. 'The entity-relationship model – toward a unified view of data', *ACM Transactions on Database Systems*, 1/1 (1976), pp.9–36.

Zaniolo, C. 'A New Normal Form for the Design of Relational Database Schemata', *ACM Transactions on Database Systems*, Volume 7(3) 1982, pp.489–99.

Websites

You should not regard these readings as the only texts you should engage with – this is a subject that benefits from wide reading. You should try to consider multiple views on the topics contained here, and to consider them critically. Try to augment your reading with online resources. Although you should aim to read academic texts as well, there is much helpful discussion on the pages of database vendors and developers. Consultants often publish blogs that can promote interesting debate – for example, Database Debunkings (www.dbdebunk.com) posts lively and engaging short articles on current issues in database management and the relational model. Such resources can prove short-lived, and so we do not list them in this subject guide, but they are not hard to find.

You should expect to install and experiment with at least one SQL-based Database Management System during this course, and you should read documentation and commentary about your choice of system. Which software you choose is up to you. A comprehensive list of products and their features may be found on Wikipedia (https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems).

The ones you are most likely to find useful are:

- PostgreSQL, sometimes shortened to Postgres; this is probably the most powerful and standards-compliant free and open-source database available, and has also been extended by many add-ons. Organisations that use PostgreSQL include Instagram, TripAdvisor, Sony Online, MusicBrainz, Yahoo! and the International Space Station. Documentation, downloads and discussion can be found at: www.postgresql.org
- MySQL is probably the most popular free and open-source database, partly because it has historically been slightly faster than the competition. Its support for SQL is less comprehensive than PostgreSQL, and it is less powerful, but it has a reputation for being easy to run and easy to embed in a web database environment. Organisations using MySQL include Uber, GitHub, Pinterest, NASA and Walmart. Documentation, downloads and discussion can be found at: www.mysql.com/ When MySQL was bought by Oracle and became only partially open source, an alternative version, MariaDB, was created by some of its original developers. This is designed to be very close to MySQL itself. It is available from: https://mariadb.org
- SQLite is a very lightweight database management system designed to be embedded in other software rather than being used as a server in the usual sense. It has a reduced support for the SQL standard. It is embedded in most web browsers for internal storage, and is part of the Android, iOS and Windows 10 distributions. Documentation and downloads can be found at: www.sqlite.org

- Microsoft SQL Server is a commercial, proprietary database management system. Microsoft provide tools for 'upsizing' from Microsoft Access databases, although migrating data from Access to other SQL databases is not too difficult. Some information is available at: www.microsoft.com/SQLServer. Please note that Microsoft Access is not suitable for this course.
- Oracle Database is a commercial, proprietary database management system with a history of strong SQL support and high reliability. Oracle is the relational database software with the highest market share. More information is at: www.oracle.com/database

Unless otherwise stated, all websites in this subject guide were accessed in February 2016. We cannot guarantee, however, that they will stay current and you may need to perform an internet search to find the relevant pages.

1.7.4 Online Library and the VLE

In addition to the subject guide and the Essential reading, it is crucial that you take advantage of the study resources that are available online for this course, including the VLE and the Online Library.

You can access the VLE, the Online Library and your University of London email account via the Student Portal at: http://my.londoninternational.ac.uk

1.7.5 End of chapter Sample examination questions and Sample answers

To help you practise for the examination, we have included some end of chapter Sample examination questions with their scope limited to the content of that single chapter, but still of approximately the same scale, style and difficulty as the ones in the examination. Although different questions may emphasise one topic over another, it is unlikely that any question can be tackled just by knowing the contents of a single chapter from the subject guide. In practice, each question in your examination can require knowledge of the whole syllabus.

You should aim to spend no more than 45 minutes answering each of these Sample examination question sections. Remember that in the real examination, you will need time to read five questions, choose which ones to answer, write your answers and check them all within the allocated time of three hours.

Once you have attempted these Sample examination sections under timed conditions, you should check the Appendix at the end of the subject guide for the Sample answers.

1.8 Examination advice

Important: the information and advice given here are based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check both the current Regulations for relevant information about the examination, and the VLE where you should be advised of any forthcoming changes. You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

In the examination for this course, you will be required to answer four out of five questions in the paper within a 3-hour period. Each question is worth up to 25 marks. In practice, each question in your examination may require knowledge of the whole syllabus.

Although different questions may emphasise one topic over another, it is unlikely that any question can be tackled by knowing just one area, and you

should be expected to be tested on every topic no matter which questions you choose. Since one question from the examination may be left out, it is good practice to read each question before you start – if only one has an element that you doubt you will be able to answer, then skip it.

It is common for at least one question to provide a simple textual description of a real-world system and ask the candidate to provide a model for it, whether using a purely relational model (see Chapter 3 of Volume 1 of the subject guide), or an E/R model or a set of SQL tables (see Chapter 4 of Volume 1 of the subject guide). These questions present a good opportunity to get marks quickly, but take care in analysing the description, as it is important to show that you have understood it.

Be prepared to summarise definitions and key terms quickly and in a few words – some will certainly be needed. When asked for an explanation or a list of reasons, note the number of marks allocated to the question. This is likely to reflect both the amount of time the question is expected to take you and the number of elements in the answer – if you are asked to provide a list of advantages of a technology, firstly consider whether each answer is likely to be worth ½, 1 or 2 marks and then compare that with the marks for that question. This should tell you the number of elements you are expected to list.

Many aspects of the examination will recur, with changes, in different years. You can access previous papers and *Examiners' commentaries* on the VLE. Practising with past examination papers is probably the single best way to prepare in the weeks before the examination. You are strongly advised to practise them under timed examination conditions.

Remember, it is important to check the VLE for:

- up-to-date information on examination and assessment arrangements for this course
- where available, past examination papers and *Examiners' commentaries* for the course.

1.9 Overview of the chapter

In this chapter, we have introduced the course and this volume of the subject guide, listing the aims and objectives of the course and outlining some practical aspects of working through the subject guide, the reading and other resources, before offering some advice on examination preparation.

1.10 Test your knowledge and understanding

1.10.1 A reminder of your learning outcomes

By the end of this course, and having completed the Essential readings and activities, you should be able to:

- understand the main issues relating to database systems in general
- have detailed knowledge about the relational model
- analyse a specific problem, synthesise the requirements and accordingly perform a top down design for a corresponding (relational) database
- have the knowledge and practical skills to implement and maintain a relational database (in SQL)
- be aware of alternative models of and approaches to database systems, particularly web-scale systems, and also including object databases, deductive and knowledge-based systems, etc.
- have more in-depth knowledge in one or more of the previously mentioned trends.

Notes

Chapter 2: Databases – basic concepts

2.1 Introduction

This chapter considers some basic concepts of databases, what they are and their advantages and/or disadvantages before introducing data modelling and relational theory.

2.1.1 Aims of the chapter

The aims of this chapter are to give you an overview of databases, explain their components, introduce the architecture of a database system, consider what a database management system is, before explaining data modelling theory and defining its place within the context of database systems.

2.1.2 Learning outcomes

By the end of this chapter, and having completed the Essential readings and activities, you should be able to:

- discuss the limitations of the file-based approach
- describe the way the database approach overcomes the limitations of the file-based approach
- define and explain what is meant by a database system and a database management system (DBMS)
- describe the three-level ANSI/SPARC architecture of a database system
- discuss the schemas and mappings corresponding to the three level ANSI/SPARC architecture
- discuss the concept of data independence
- explain the role of each of the components of a database system – data, hardware, software and users
- present the most important features of a DBMS
- discuss the advantages and disadvantages of database systems
- discuss issues related to distributed database systems
- explain what a data modelling theory (or data model) is and define its place within the context of database systems.

2.1.3 Essential reading

- Date, C.J. *An introduction to database systems*. (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Chapter 1 and Chapter 2.

and/or

- Connolly, T. and C.E. Begg *Database systems: a practical approach to design implementation and management*. (Pearson Education, 2014) 6th edition, global edition [ISBN 9781292061184 (pbk)], Chapter 1 and Chapter 2.

2.1.4 Further reading

- Elmasri, R. and S.B. Navathe *Fundamentals of database systems*. (Pearson, 2016) 7th edition [ISBN 9781292097619 (pbk)], Chapter 1.

- Ramakrishnan, R. and J. Gehrke *Database management systems*. (McGraw-Hill, 2002) 3rd edition [ISBN 9780072465631 (hbk); 9780071231510 (pbk)], Chapter 1, sections 1.1–1.4.

2.2 What is a database?

The first – and most obvious – question to ask when you take up this subject is the simplest – 'What is a database?' Certainly, you will have dealt with them, indirectly, almost daily. Whether you are in a shop in person or whether you are exploring its catalogue on the internet, when you check whether a product is in stock, it is likely that a database will be used somewhere within the system. Amazon and Facebook, YouTube and iTunes all use databases to deliver products and services to their users.

The database and its structure may be quite obvious to the user for a library catalogue or an online retailer, but it may also be serving a less direct purpose, allowing the company to keep track of its employees and suppliers, or helping an advertiser track visitors to web pages across different sites, tailoring their adverts to match a browser's activity.

Activity

Before you read on, try to list some other examples of databases you have come across. What do they have in common? Based on your examples, write down what you think are the most important features of a database.

A database system is a system that stores data. To qualify as a database system, there are some features that it would have to offer:

- find (**retrieve**) data
- add (**insert**) new data
- **delete** unwanted data
- change (**update**) data.

This definition will be refined and formalised in the sections to come, but first, we can illustrate these features with an example.

Consider a shoe shop, specialising in trainers. The shop keeps information about the products it sells. This information could be organised in the form of a table, as shown in Figure 2.1 (prices are in U.K. pounds sterling, shown as £), and could be part of the shop's database.

Model	Brand	Size (EU)	Location	Price	Stock level
Air Max 90	Nike	35	K1S4	£40.00	3
Air Max 90	Nike	37	K1S4	£40.00	12
Mesh	Kaplan	48	MFash1	£65.00	18
Ferris	Vans	41	WPlim2	£10.00	1
…	…	…	…	…	…

Figure 2.1. A part of a shoe shop's database of trainers.

The 'Vans Ferris' range is about to be discontinued, and the shop will no longer have them for sale once the pair in stock is sold. At that point, the **record** for that shoe – the row in the table – will be **deleted**.

'Kaplan Mesh' shoes have not been selling well, and the large number in stock is taking up space, so the price will be reduced to £40. The corresponding **field** for the price of that product – a cell in this table – will be **updated**.

A new fashion trainer, the 'Asics Gel Lyte VI' has just been released, so a new record – a new row in the table – must be **inserted**. In this example, we would expect several new records – for different sizes of shoe – to be inserted, but we will only show one, to save space.

Figure 2.2 shows the table after these operations – deletion, updating and insertion – have been carried out.

Model	Brand	Size (EU)	Location	Price	Stock level
Air Max 90	Nike	35	K1S4	£40.00	3
Air Max 90	Nike	37	K1S4	£40.00	12
Mesh	Kaplan	48	MFash1	£40.00	18
Gel Lyte VI	Asics	46	MFash3	£100.00	14
…	…	…	…	…	…

Figure 2.2. Part of the shoe shop's database after some changes. Altered fields are underlined.

A database has a structure and content. The structure is represented in this example by the table headings; the content by the body of the table. The content changes in time – it is dynamic in nature. The structure can change, but it is far less changeable than the content. For instance, you could add a new column to this table – the type of trainer or the activity it might be associated with – but you would not expect to make such changes that often. The structure of the database is called its **intension** and the content is called its **extension** (we will return to this in more detail later in this chapter).

Although it may not be obvious from this example, a database is capable of storing a large amount of data.

So far, a database system is, for us, nothing more than a system that manages data. But is any system that manages data a database system? Is there anything that all database systems have in common, that distinguishes them from other software systems? The answer is obviously yes. In order to understand the '**database approach**', we shall first have a brief look at file-based systems. In appearance (behaviour) they are similar to database systems, but they are conceptually (qualitatively) different. We shall identify the drawbacks of the file-based approach to data management and then introduce the database approach as a solution to most of these drawbacks.

Activity

Consider the database examples you listed in the previous activity. For each one, think about it as a table like the ones in Figures 2.1 and 2.2 above. What columns would the table have? Would all the information fit in a single table, or would there be several?

2.2.1 File-based systems

We shall start with a definition of file-based systems.

Definition: A file-based system is a collection of application programs, each managing its own data.

In a file-based system, permanent data is stored in various files of ad-hoc structures. Each application program defines and handles its own data files independently of the others. This approach is called the **de-centralised** approach. Each application program works with its data at the physical level, manipulating records as they are organised in persistent memory. Sharing of data between applications is likely to be limited.

The concept of a **physical** level for data is one to which we will return later. The structure we describe is not purely physical, but we use the term to indicate that it is to some extent platform dependent, because access to files is made through the primitives (built-in functionality) of the operating system.

Take, for example, an estate agent's office, for which we shall consider the Sales and the Contracts department. Each department maintains its own data in its own data files, as depicted in Figure 2.3.

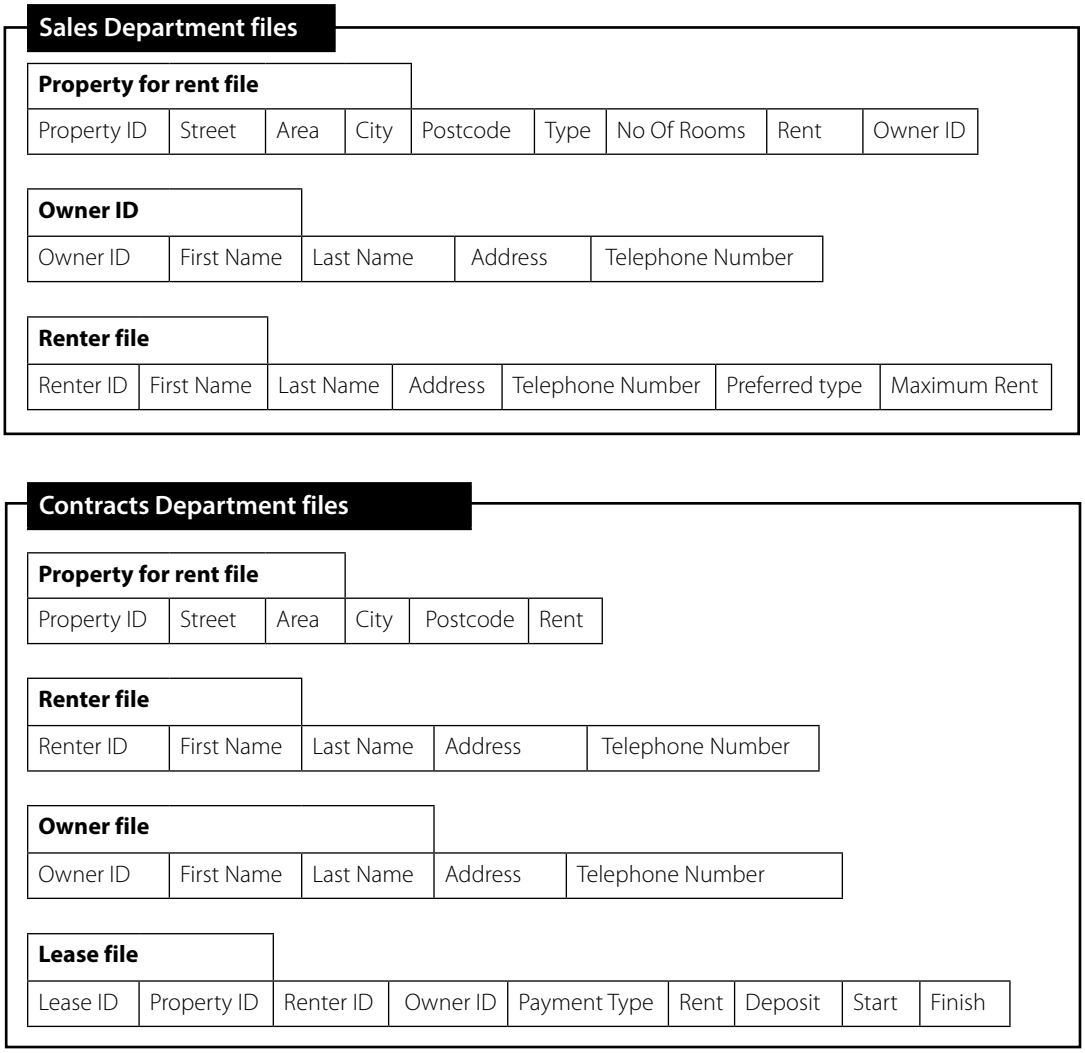


Figure 2.3. A file-based system for an estate agent's company.

The Sales Department needs:

- detailed information about the properties for rent, so staff can give good advice to customers (such as Type and No Of Rooms from the Property for rent file);
- detailed information about customers, so that their needs can be appropriately matched to what is available (such as Preferred Type and Max Rent in the Renter file);
- 'identification' information – such as name, address and telephone number – about customers, the properties on offer and their owners.

The Contracts Department needs:

- detailed information about the renting contracts (in the Lease file);

- 'identification' information – such as name, address, telephone number – about customers, the contracted properties and their owners.

Some drawbacks of this solution are obvious. These are the limitations of the file-based approach in general. The most important are enumerated below.

Duplication. Different applications might have to make use of the same information. Because each application has its own files, data is duplicated (e.g. the 'identification' information in our example). This aspect has at least two negative consequences. Firstly, duplication is wasteful.[1] Secondly, data can become inconsistent – it can have different values in different files (belonging to different applications), even though it is supposed to give the same piece of information. For example, the address of an owner, Mr. J. Morris, might be updated in the Owner file belonging to the Sales Department, while the Contracts Department might still have Mr. Morris's old address.

[1] Wasting disk space is unlikely to be a significant concern in the example we have given – storage is cheap – but in situations where the number of applications and the scale of duplication is greater, this can become more important.

Separation and isolation. Data is scattered among different files, each file belonging to a certain department. A department has access to its own files, but no access to the files of the other departments. Files belonging to different departments cannot be used together in order to create more complex data or analysis. Often, because they are based on different infrastructures (platforms, development software, etc.) files belonging to different departments cannot be transferred (copied) across.

Program-data dependence. Each file belongs to a certain application program. The (physical) structure of data is defined inside the application program. This could easily – and usually does – lead to incompatible file formats between applications, meaning that it becomes impossible to share data between them. Another aspect is that data definition is embedded in the application program. That means that if the physical structure of data is to be changed – for instance, if instead of representing a year with two digits, it is to be represented with four[2] – then the application program itself must be changed. Not only that, but the methods of access and manipulation of data are also embedded in the application program (for instance, in previously-defined queries); to change them, the application program must be modified.

[2] This was a problem in the late 1990s with the 'millennium bug'. Systems that represented, for example, 1998 as '98' were in danger of getting calculations wrong, or not functioning at all, when called upon to represent 2000. Many businesses had to pay for the rewriting of their file-based systems.

In the file-based approach, the emphasis is placed on functionality – provided by the application program. Data modelling takes a lower priority. This approach leads to the drawbacks we have listed. If the approach is inverted and we consider data as central, then these problems can be removed. Informally, this represents the database approach.

2.2.2 Databases and database management systems

We shall start with the definition provided by **Connolly and Begg**:

Database. 'A database is a (shared) collection of logically-related persistent data (including its description) as part of the information system of an organisation.'

Let us now explain this definition. A database is a **large** repository of data, in which data is **defined once** and **stored once**. Data that was scattered in different files – with different formats and owners – in the file-based approach, is now **integrated** with minimum redundancy (duplication), as a single resource. Different application programs will **share** this common resource, usually concurrently (at the same time).

There are times when redundant data is necessary, some of which will be explored in this guide. You might choose to store intermediate results that are called for often (using **snapshots**) or to ensure the atomicity of a set of operations (**transactions**). In these cases, the redundant data is intentionally included to achieve something extra, and even so requires special treatment.

The diagrams in Figure 2.4 and Figure 2.5 illustrate the differences between the database approach and the file-based approach.

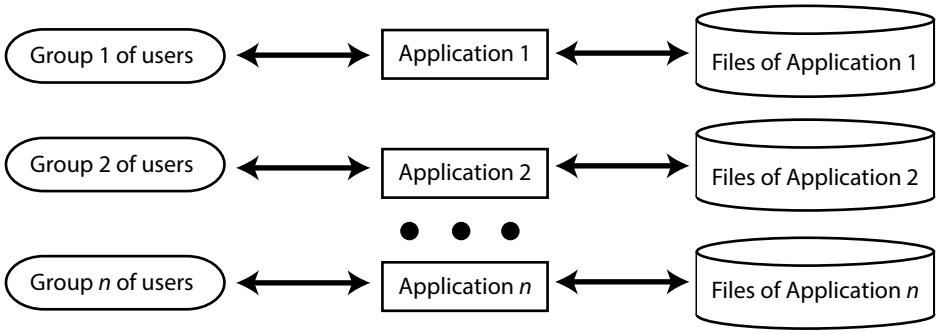


Figure 2.4. The file-based approach.

In the database approach (see Figure 2.5), the raw data is integrated in a common database for all applications. The data is managed by a **database management system (DBMS)**, which provides shared access to it, for all the applications in the system.

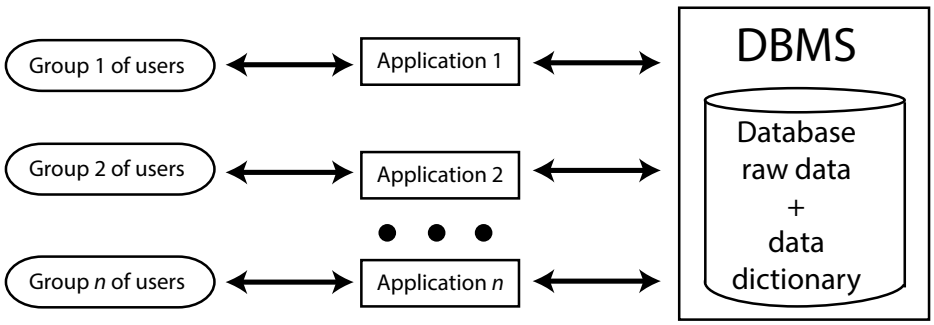


Figure 2.5. The database approach.

A database management system is a software system that provides a set of primitives (built-in functionality) for defining, accessing and maintaining a database.

A database stores both the raw data and its description. We say that the information[3] stored in a database is self-describing. The description of the raw data is known as the **system dictionary, data dictionary or metadata**.

[3] *For the purposes of this course, 'data' and 'information' will be used synonymously.*

The consequence of this approach is **program-data independence**. This means that the structure of data may change without affecting the application programs that use it. This basic definition is going to be refined and better explained later in this chapter.

This approach – separating the data definition from application programs – is similar to **data abstraction** in programming, where the internal definition of an object is kept separate from its external definition. An outside system can only see the exterior of the object. As far as the external definition remains unchanged, any changes in the object's internal definition do not affect the outside system.

A database management system automatically performs a lot of the housekeeping tasks that would otherwise be the responsibility of the

application programmer. As a result, the user – i.e. the person who defines and uses the database – is presented with a **clean** and **powerful** set of **tools** for database development and exploitation. A more detailed description of both database systems and database management systems is provided in the following sections. At this stage, it is important that you broadly understand why database systems are needed and what their main benefits are.

This definition, with its distinction between file-based and database approaches is quite high-level and functional. From the discussion above, many organisations that use **database software** would still be defined as having a file-based system, if different departments use different database implementations for storing similar data.

Activity

Before you read on, try to think of an organisation you know – perhaps one you've worked for or studied at – that has multiple systems similar to what is described above. What problems can occur when you have data duplication like this? Does it matter whether the separate systems store their information in database software or spreadsheets? Do you think this is a useful definition of database systems?

2.3 The three-level ANSI/SPARC architecture of a database environment

Program-data independence is one of the most important advantages offered by the database approach. This independence can be achieved if the system is **abstracted** into two or more levels. A low-level abstraction deals with how data is organised on the physical support.[4] Meanwhile, a high-level abstraction describes the logical structure of data, **irrespective of its physical representation**. This separation allows the separation of the design of a database from the details of its implementation.

[4] This is a generalised term for the permanent memory or disk storage of the system.

This idea can be further refined.

- Users and application programs should be freed from considering the aspects of the system related to the physical representation of data, such as storage and accessing details. Instead, they should be able to take into consideration only the logical structure of data. Rather than having to deal with such aspects in each application program it would be much better if these problems were to become the responsibility of the system (DBMS) that manages data.
- It should be possible to change the physical representation of data without affecting users, as long as its logical structure is preserved.
- As we have seen, the database integrates **all** the information required within an organisation. Individual users will often only need (or be allowed) access to certain parts of this 'pool of information'. Each user, then, needs to have a customised view of the database and it should be possible to change that view without affecting other users.

These aims were formalised in the early 1970s and codified and adopted as a standard in 1975 as the **ANSI/SPARC three-level architecture**. The architecture forms a basis for most modern DBMS.

The ANSI/SPARC architecture consists of three levels of abstraction (see Figure 2.6). The **external level** represents the way data is viewed by individual users. The **conceptual level** represents the way the organisational data (i.e. all data that is relevant for the organisation) is structured. The **internal level**

represents the way data is physically stored, although the very lowest-level aspects of that are likely to be handled by the operating system itself.

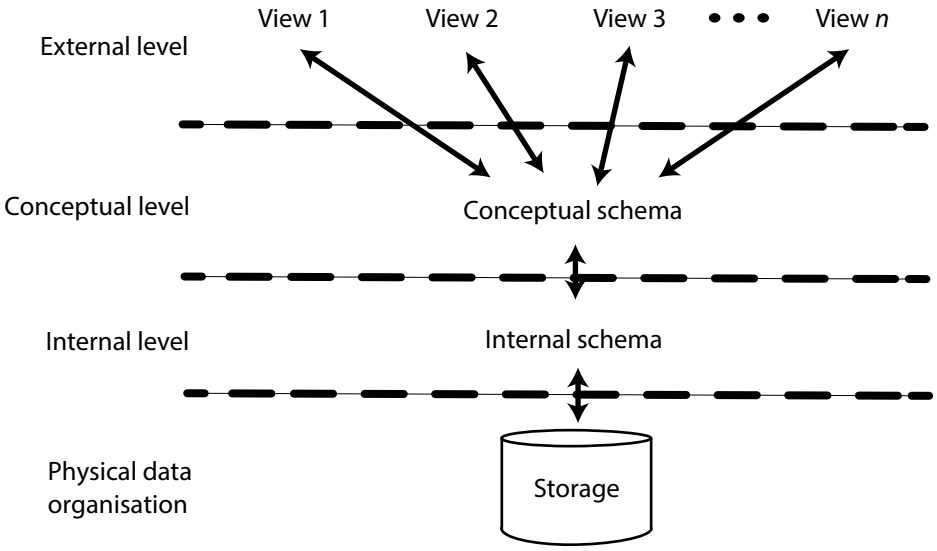


Figure 2.6. The ANSI/SPARC three-level architecture.

The external level. This incorporates each user's external view of the database. A user's view consists only of the data needed by that user. Other data may exist in the database, but the user does not need to be aware of it.

For instance, suppose that the database of a software company includes information about its employees. The Personnel Department's view of the employees – the data that is relevant to them – might consist of: name, address, sex, date of birth, qualifications, department for which the employee works, current salary, job contract details, and details about previous jobs. The Personnel Department needs to be able to access this data about any employee in the company.

On the other hand, the Development Department's view of the employees might consist of: departmental ID, name, telephone number, timetable, the projects in which the employee is involved including the employee's role in each project, the objectives and their deadlines. Only the data about its own employees is relevant for the Development Department.

The whole information about the company's employees is stored in the database. However, the two departments need access to different **projections** of data. The data relevant to each department represents the department's external view of the database (Figure 2.7).

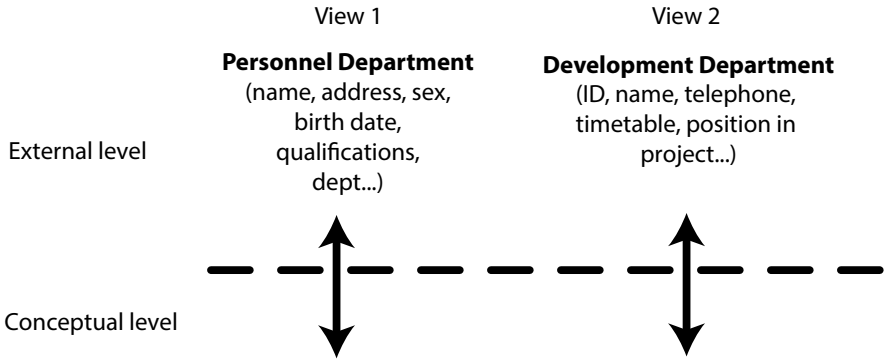


Figure 2.7. Two external views of the same database, one accessible to Personnel Department users, and the other to Development Department users.

Different users may want to see the same data in different formats, for instance, name might be stored in multiple fields (for first and last name), but will be required as a single name. The external level might also include derived data – calculations based on stored data – for instance, a user might need an employee's age, which could easily be calculated from the date of birth stored in the database.

The conceptual level. This represents the logical structure of the database (of all the data required by the organisation). It can be seen as the union of the whole set of views at the external level. Conversely, any view should be derivable from the conceptual level. The conceptual level represents the information stored in the database about the real life system's entities (objects) and the relationships between them. The representation of data at this level is still independent from any physical considerations – it specifies what is stored, rather than how it is stored.

The internal level. This describes the physical representation of data. The internal level specifies **how data is stored**. It is at this level where the physical data structure and file organisation are defined. The internal level is situated at the interface between the DBMS and the Operating System (OS). It is quite common that the internal level of the DBMS uses the file management primitives of the OS. However, there is no clearly defined boundary between the OS and the internal level. Because of this, there often exists another level below the internal level, namely the actual physical support (cylinders, blocks, clusters, etc.) (this extra layer is shown in Figure 2.6).

2.4 Schemas and mappings

Before we look into how the three abstraction levels are defined within a DBMS, the distinction between the description of the database and the content of the database must be made clear.

Database schema. The description of the database is called the **database schema** or the database **intension**. This is specified at the creation of the database. It is not expected to change very often.

Database instance. The raw data that populates a database at a particular point in time is called a **database instance** or the **extension** of the database.

Consider, for example, a database that maintains information about the employees of a company. The required information for each employee is: name, date of birth, address, job and pay scale. Suppose that the database is organised in the form of a table. Then, the specification of the table's heading can be considered as the database schema (or database intension), whereas the content of the table at a specific moment in time can be considered as the database instances (or database extension) (see Figure 2.8).

Name	**Date of birth**	**Address**	**Job**	**Pay scale**
String	*Date*	*String*	*String*	*Integer*
Name	**Date of birth**	**Address**	**Job**	**Pay scale**
A. Johnson	02-02-1995	London	Programmer	10
S. Lee	09-07-1996	Leeds	Analyst	14
J. Singh	05-05-1989	Edinburgh	Programmer	12

Figure 2.8. Tables showing a database schema (above) and extension (below) of a database table.

The database schema consists of three types of schemas, one for each level of abstraction:

- **External schemas** describe the external level. There is one schema for each view.
- The **conceptual schema** (one only) describes the conceptual level. All the definitions should only take the logical structure of the data into consideration. Implementation aspects, and user views, should be disregarded.
- The **internal schema** describes the internal level. It defines the physical records, methods of representation, index implementation, etc.

As a result of this separation of concerns, mappings are needed to allow navigation between the schemas. Since there are three types of schema, there are two types of mapping:

- **External/conceptual** defines the correspondence between an external schema and the conceptual schema.
- **Conceptual/internal** defines the correspondence between the conceptual and the internal level.

Figure 2.9 illustrates these ideas with a simple database maintaining information about a company's employees. Two external views of the database are considered in this example; one for the Finance Department and one for the switchboard. The information needed by the Finance Department uses the full name, age and salary of each employee and is defined by its corresponding external schema. Each employee must be uniquely identified. Since two employees might have the same name, a unique identifier, ID, is used.

The information needed by the switchboard, defined by the respective schema, only refers to the name, job and telephone number of each employee. The switchboard needs employees' first and last names to be separate, so they can be sorted easily. For the switchboard, unique identification of each employee is less critical, and the job title combined with the employee's name is used. The assumption that two employees having the same name will not have the same job title as well is considered safe enough.

Other external views might exist too, and other data might be stored in the database but we are dealing with a heavily simplified example for the sake of clarity.

The **conceptual schema**, that defines the conceptual level, unites the data required to support the two views. It specifies the identifier, first and last name, date of birth, job title, employment date, salary scale and the telephone number of each employee.

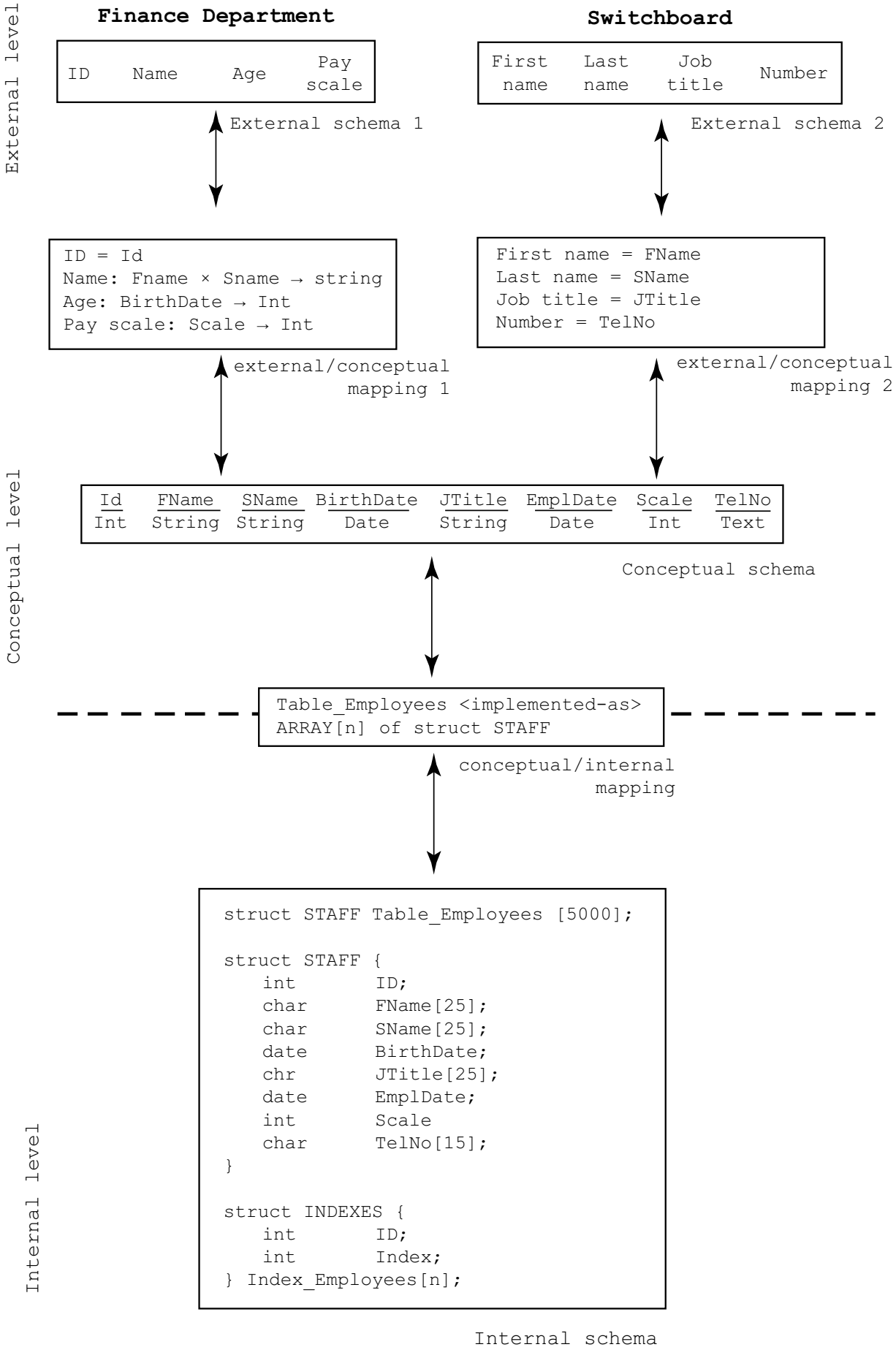


Figure 2.9. Schemas and mappings for an employee database.

The link between the external schemas and the conceptual schema is made by the **external/conceptual mappings**. The mappings corresponding to the Finance Department's view is defined as:

Finance Department schema	Conceptual schema
ID=Id	Id
Name=concatenate(Fname, ' ', Sname)	Fname, Sname
Age=current_year() – yearOf(BirthDate)	BirthDate
Pay Scale = Scale	Scale

The other mapping – between the Switchboard's view and the conceptual schema – is self-explanatory.

The **internal schema** consists of the data structures that are used to represent (implement) the conceptual schema. For the above example, this is struct `STAFF`, in a C-like hypothetical language. It also can include other structures (i.e. not derived from the logical level), used for pragmatic reasons (e.g. efficiency). In the above example, an index was defined, `INDEXES`, in order to make the retrieval operations (from `Table_Employees`) faster.

The **conceptual/internal mapping** links the definition of data at the conceptual level with the way it is actually represented – it links **what** data is represented with **how** it is represented. The table `Employees` is implemented as an array of records of type struct `STAFF`. Note that the index, `Index_Employees`, is used purely at the internal level (i.e. it is not mapped to the conceptual level). This is because the index does not describe the data as such – it is a way of making access to the data faster or more efficient.

As a final point, the issue of data independence can now be reconsidered. One of the main advantages provided by the three-level architecture of a database system is the provision of data independence. There are two types of data independence:

Physical data independence is the immunity of application programs to changes at the internal/physical level.

Logical data independence is the immunity of application programs to changes at the conceptual level.

For instance, in the example above, the `Employees` table can be implemented using a linked list. If the **conceptual/internal mapping** is modified appropriately, and as long as the conceptual schema stays the same, the application programs (situated above) remain unaffected.

Logical data independence may be more difficult to achieve since application programs typically rely heavily on the logical structure of data. However, suppose another view is needed, for the Personnel department, which requires information about the address and the family status – such as marital status, dependants and next of kin – for each employee. The conceptual schema can be extended as necessary, without affecting the other two views.

2.5 The components of a database system

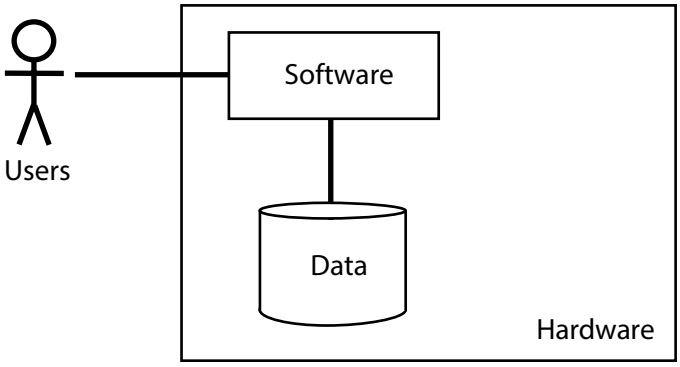


Figure 2.10. The components of a database system.

At the highest, most general level, a database environment consists of:

- **data**, representing the information needed for an organisation;
- **software**, serving two purposes:
  - the management of the stored data, and
  - further processing of the data to the users' needs;
- **hardware**, supporting both the stored data and the software components;
- **users**, broadly divided into two categories:
  - developers of the database system, and
  - users of the system.

2.5.1 Data

Data can be classified into two categories, namely:

1. Primary data – the fundamental information necessary to provide the database service, stored on permanent support, such as hard disks.
2. Derived data – information that can be inferred or calculated from primary data (and may be recalculated at any time).

Derived data may be the output of the application programs – the result of processing the primary data – in a form suitable for the users' needs, but it can also be the input from users that will then be processed by the application to be stored as primary data.

The focus of a database system is on primary data. This has to be appropriately identified, described and implemented. The primary data has three important characteristics. It is:

- **integrated**, rather than existing in separate systems – it has been gathered together into a single system[5]
- **shared**, with all the applications belonging to the information system having common access to (at least parts of) it
- **extensive**, in that database systems are usually developed for data intensive applications, where their benefits are more clearly felt.

Stored data, as we have already seen, does not include only the raw data, but also its description – the metadata, system dictionary or catalogue.

[5] This 'single system' may still be distributed across multiple servers, the implications of which will be discussed in Volume 2 of this subject guide.

2.5.2 Software

The software component can be seen as consisting of three layers (Figure 2.11):

- **the operating system (OS)**, positioned at the base, provides the necessary routines for accessing the hardware resources (such as file handling or memory management routines);
- **the database management system (DBMS),** placed above the OS – and using the routines that the OS makes available – provides all the necessary primitives for data management, including languages for defining schemas, manipulating and reading data and so on;
- **application programs**, above the DBMS – and using the routines made available by the DBMS – provide data formats and computations beyond the capabilities of the DBMS.

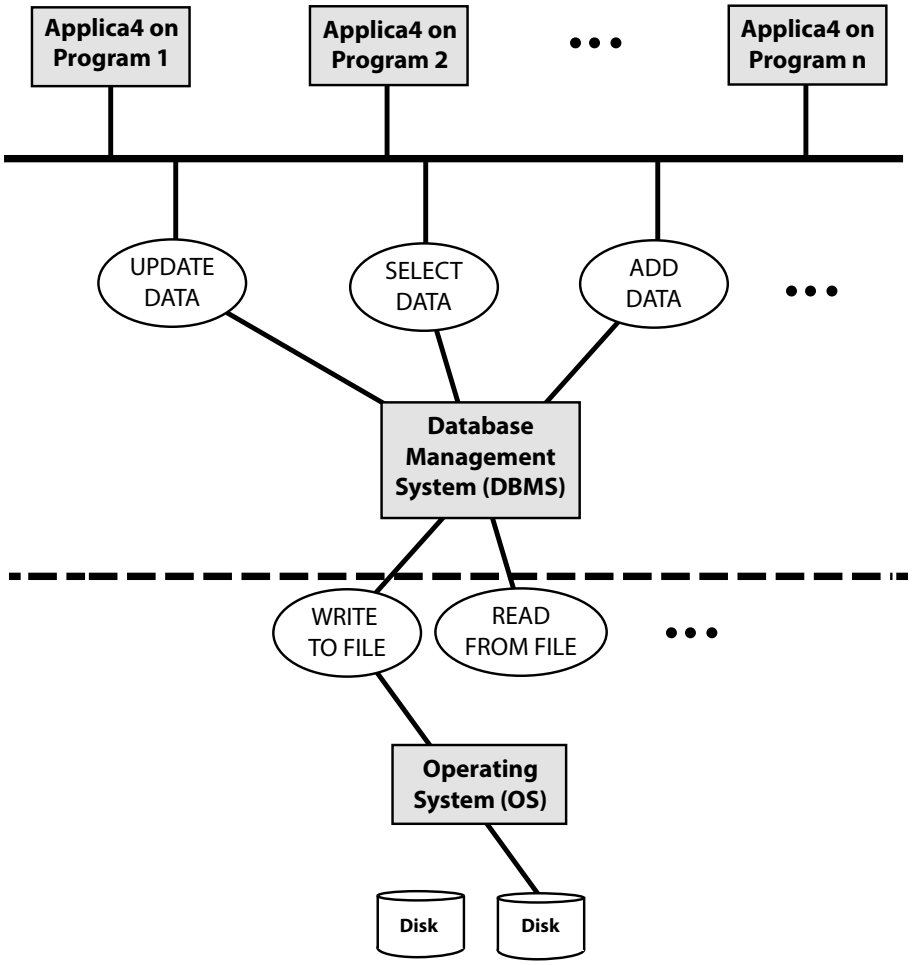


Figure 2.11. The layered structure of the software component of a database system. Anything below the dashed grey line is platform-dependent, and so will not be discussed here in any detail.

The hardware and the OS are often grouped together and called the **platform**. There is considerable variation between platforms, which is one reason for having the DBMS software handle this variation and present a more abstracted interface to higher-level components. This provides a platform independence that shields the application programs from unnecessary physical details, and means that we need not concern ourselves with details of hardware or OS for the remainder of these subject guides. Instead, we focus on the features provided for the application programs by the DBMS.

The features of the DBMS will be considered in detail over the course of this chapter. Briefly, the DBMS provides support for schema definition, data manipulation, data security and data integrity.

The application programs can be of two kinds:

1. user developed;
2. provided together with the DBMS by its developer.

The former class of applications will generally be written in a high-level programming language, such as *C*, *Java* or *Python*. Support for database access in such languages is provided by means of a data sub-language, embedded within the **host language**. Statements written in the **embedded sub-language** are processed and passed on to the DBMS using the appropriate routines.

Programs provided by the DBMS developer allow the rapid development of user applications, without the user writing any conventional code. Programming tools abstract away or remove so much functionality in order to allow often application-specific software to be constructed quickly; these are known generically as **fourth-generation tools**. Home or small business database systems – such as Microsoft Access or OpenOffice Base – provide graphical fourth-generation tools for this purpose.

The DBMS can also be referred to as **server** or **backend** (server), whereas the application programs are referred to as **clients**, or **front-ends**. Clients use the services provided by a server for data management. The division between client and server makes it possible for the server and client to run on different machines, giving rise to the idea of distributed processing, an issue discussed in the 'Database architectures' section and elsewhere in these subject guides.

2.5.3 Hardware

As we have seen, the DBMS allows both the developer of a database and the database users to operate without knowing the details of the hardware being used. This does not remove from the system administrator the need to select hardware and operating systems that, firstly, are capable of running the chosen software; and secondly that can cope with the demands that will be placed upon it by the database and associated systems. The system administrator should be satisfied that:

1. There is enough permanent storage space, for instance disk space, to store the data and any indexes and cached derived data.
2. There is enough temporary storage space, for instance RAM, to hold intermediate results and computations.
3. There is enough computational power to manipulate the data at the rate that will be required.
4. There is fast enough communication between components of the system for moving the data between them. This is only usually an issue for particularly data-heavy applications or systems with a very large user base.

Although DBMS vendors will provide recommendations for minimal configurations required for different sizes of application, individual use cases will have a large impact on the system requirements.

2.5.4 Users

Users, as a component of a database environment, can be classified in four categories, according to the role they play.[6]

Data administrator. The data administrator (DA) is a user who properly understands the data requirements of the organisation and is in charge of administering the organisation's data. This user:

- decides which data is relevant and which is not;
- is in charge of applying the organisation's policy and standards;
- decides on the security policy, and so on.

The DA does not need to be a technical expert or a manager. Rather, the DA is somewhere in between, liaising with the management on one hand, and with the technical team, on the other.

Database administrator. The database administrator (DBA) is the technical user in charge of the database system. More specifically the DBA is responsible for the database's design, implementation and maintenance, and deals with both the correctness of the implementation and the efficiency of the database system. The DBA must have good technical knowledge and is in charge of the definition of the DB schemas, integrity and security rules, access procedures, backup and recovery procedures, performance of the system, etc.

Application programmer. The application programmer writes programs that perform more complex processing of data (either computations or formatting). For this, they use either a third-generation language, embedded with a database language, or a fourth-generation tool. The resulting programs are for use by **end users**.

End user. The end users are the 'beneficiaries' of the database system. They may range from technically **naïve** to extremely sophisticated. A technically naïve user, for example a bank employee, may interact with the system using application programs developed for specific tasks. A naïve user does not have to be aware of the functionality of the DBMS. All they need is reliable and easy to use programs that they can use with minimal fuss. A **sophisticated** user, on the other hand, will know how to access the database directly, through the database language supported by the DBMS. Sometimes a sophisticated user might even develop applications, and so become an **application programmer**.

*[6] The term **user** is often used in software engineering to refer to one or more people playing a particular role in interacting with software. That means that a 'user' here can mean several people, and one person can be several different 'users' in different contexts, depending on the work that she or he is doing at the time.*

Activity

At the beginning of this chapter, you were asked to list databases you had encountered in real life. For each one, consider which group of user takes which of the above roles. Is the separation always clear?

2.5.5 DBMSs and database languages

The database management system is the software through which all access to the database is made. This is a concise but limited definition. In reality the DBMS is responsible for much more. Some of its important features are presented below.

Data definition. The DBMS must provide support for defining or modifying the database schema. Schema definition includes specifying data types, structures, constraints and security restrictions. This is achieved by means of a **data definition language (DDL).** The statements made in DDL for a specific database system represent the system's catalogue (or data dictionary[7]). In theory, there should be a DDL at each level of abstraction (i.e. external, conceptual and internal), but in practice there usually exists a single DDL that allows definitions at any level.

[7] *Some authors consider the term data dictionary to be more general than system's catalogue (Connolly).*

Data manipulation. The DBMS must provide support for data manipulation. In particular it has to support:

- **retrieval** of existing data, contained in the database
- **deletion** of old data from the database
- **insertion** of new data
- **modification** of out-of-date data.

This is achieved by means of a **data manipulation language (DML**). There can be a DML at each level of abstraction. At the external and conceptual level, the DML is concise, comprehensive and easy to use; in other words, the emphasis is on its expressive power – on these levels, efficiency is a secondary goal. On the other hand, at the internal level, the emphasis is placed on the DML's efficiency. This means that its statements are complex – and probably not that straightforwardly expressible – but quite efficient.

These languages (DDLs and DMLs) are called data **sub-languages** because they do not include constructs for the control of flow – they are computationally incomplete (meaning they cannot be used as general purpose programming languages).

Users can use them directly in order to define and access the database. However, for applications that require more complex data processing (and formatting) they are usually embedded into a full high-level programming language.

Some authors prefer to further divide DMLs into two categories; namely, procedural and non-procedural (declarative). Within a procedural language one must specify how the result to be obtained is computed; whereas using a declarative language one only has to specify what result must be obtained – what it looks like – the system being responsible for its computation. Since there are neither pure declarative or pure procedural DMLs – they range between the two – any classifications of this kind are rather ad-hoc in nature. For example in certain situations SQL can be considered declarative while in others it can be considered procedural.

An important requirement for DMLs is to allow **unplanned** or ad-hoc queries; namely, requests that were not foreseen at the time of design. A problem that may result from this is how to gain reasonable efficiency for such unpredicted use.

Other features that the DBMS must provide include:

- support for **data integrity** – the system must ensure that there are no 'contradictions' between the data values in the database; this is achieved based on a set of integrity constraints (part of the data dictionary)
- support for **security control** – the system must ensure that data is not accessed by unauthorised users or applications; this is achieved based on a set of security rules (part of the data dictionary)

- **recovery services** – the ability to restore the database to a previously correct state in the case of a crash or error
- **concurrency facilities** – allowing the database to be accessed by more than one user at a time
- support for data **communication**
- user-accessible **data dictionary**.

2.6 Advantages and disadvantages of database systems

The main advantages and disadvantages associated with the database approach are not described in detail here. You are advised to keep this issue in your mind throughout the whole course, and to continuously review it.

2.6.1 Advantages

Reduced redundancy. In a file-based system each application has its own private files. This often leads to data being duplicated in different files, wasting storage space. In a database approach, all data is integrated, reducing or removing unwanted redundancy. There are various reasons why eliminating redundancy completely is often not possible or desirable in a DBMS – and we shall return to these in later chapters. However, where the file-based system forces redundancy in an ad-hoc way, a DBMS should provide mechanisms for specifying redundant data and for controlling it (to maintain the consistency of the database).

Avoiding inconsistency. This is largely as a result of the reduced redundancy. A database is in an inconsistent state if the same item of information is stored in at least two places in the database, but with different values. The database approach dramatically reduces that sort of repetition, making the risk of inconsistent data smaller. Even where redundant information is stored, the repetition can be made known to the DBMS, so that the system automatically enforces consistency, so that whenever some changes are made to one set of data, the same changes are propagated to the same version that is duplicated elsewhere. The support provided by most current DBMSs for preventing inconsistencies is limited to a relatively small number of categories, but the mechanism is present.

Improved data sharing. Since all data is centralised, the restrictions on which applications and users can see it are ones of security constraints rather than those of system and network architecture. In contrast to having a set of separate file-based systems, here all data is integrated, meaning that more information can be derived from the same amount of data. Both aspects considerably improve the accessibility of data.

Data independence. As we have seen in earlier sections, a database approach provides protection for applications from changes in both the physical and – at least to some extent – the logical structure of the data (physical and logical data independence).

Some other benefits of the database approach are:

- the **maintenance** of the overall information system can be improved due to **data independence**
- **integrity** can be maintained – any DBMS should allow the specification of **integrity constraints** on data

- more detailed and coherent **security restrictions** can be applied – a DBMS should allow the specification of **security rules** on data and on users
- **standards** can be enforced
- **concurrent access** can be more easily achieved
- better **recovery** mechanisms can be devised
- large-scale requirement conflicts for the information system of an organisation can be balanced and resolved.

2.6.2 Disadvantages

Complexity. In the database approach the information needed by an organisation is modelled and implemented as a whole. Where the file-based approach can often be achieved piece by piece as individual departments develop a need and budget, the process of developing a database system is by its nature a single, unifying and more complex process, which will include:

- data acquisition
- data modelling and design
- database implementation
- database maintenance.

The greater complexity of this process may mean that errors in implementation, design and data acquisition may occur, and be harder, within the organisation, to get fixed.

Depending on the organisation and its data need, the database approach may require extra hardware and IT infrastructure, along with new maintenance contracts. Depending on the system being replaced, this can make a database approach more expensive, in terms of either initial or ongoing costs. The DBMS software itself may cost no money, since there are many free and open source options, but the system built around it will require developer time and may also incorporate other, paid-for software. In some cases, the integration of several systems may represent a reduction in costs, as separate contracts and IT structures are rationalised and unified.

Higher impact of failure. The database system is at the core of the information system of an organisation. All data is stored centrally, in the database. As a result, most applications rely on this data. If the DBMS fails, the whole organisation is paralysed, unlike a decentralised system, where a failure in one system will only directly affect the department that uses it.

Performance. DBMS software is heavily optimised for its core functionality, but it is still a generic piece of software. A database application may be slower for an individual user than a bespoke, perhaps local, file-based solution.

2.7 Architectures of database systems

We have seen that when the data of an organisation is integrated in a single database, it can be shared between many applications. Accordingly, a 'natural' organisation of a database system is the **client-server architecture** (Figure 2.12). The DBMS is the **server** and the application programs are **clients**. A server can also be referred to as back-end and a client as front-end.

In the client-server architecture, the DBMS (including the database) runs on a dedicated machine – the server machine. The server machine is tailored to support the DBMS, both in terms of storage space and computational power. In high-demand situations, it has to provide:

- extensive and fast external (persistent) memory
- powerful processing capabilities (fast processors), combined with sufficient internal memory.

The main requirement for the server machine is to provide the resources that the DBMS needs to respond efficiently to the requests received from the clients (i.e. to provide what they need at an appropriate speed).

Although we speak of the server as a separate machine, it is becoming increasingly common for virtual servers to be used by organisations on a subscription basis. These run on remote servers and may offer advantages, such as easier upgrades and less in-house maintenance, and disadvantages, including questions about data security and privacy.

The applications would normally run on different client machines, with each client machine specified to meet the needs of its application or applications. For instance, if an application program performs complex graphical processing, then a more powerful graphical workstation might be required, whereas if an application only performs simple data entry, then a cheaper, less powerful machine might be enough. If these needs change, it is only the client machine that has to be 'modified', making thus the client-server architecture quite flexible.

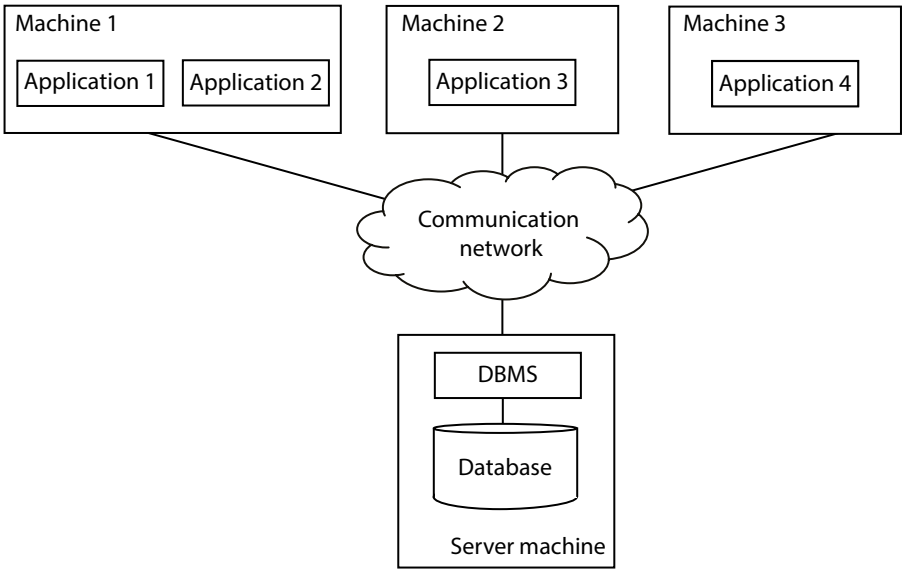


Figure 2.12. The client-server architecture.

The communication between an application and the DBMS is accomplished through the link between the client machine and the server machine, via a communication network.

Distributed database systems can be developed using various different architectures (a chapter in Volume 2 of this subject guide considers them in more detail). Distributed systems may follow the client-server architecture. Another possible way of organising this is to have the database itself distributed on several machines (Figure 2.13).

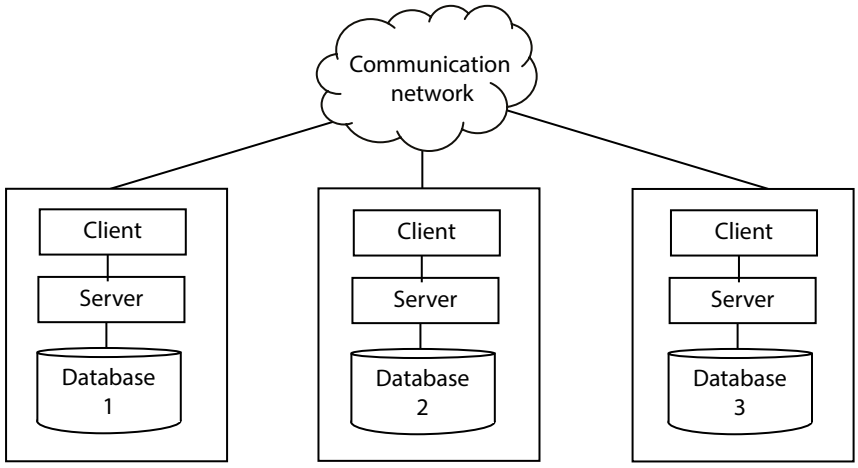


Figure 2.13. A distributed database system architecture.

In this architecture, each machine supports a part of the organisation's database and can become a client of the other servers in the network. The organisation's database is thus constituted from the union of the individual databases. For this approach to confer its main advantages, though, the individual databases should be exploitable independently.

2.8 Data models

Physical data independence is one of the main advantages of database systems. This is achieved based on the conceptual level. Users work with data – both defining and manipulating it at the conceptual level, while the DBMS takes care of the physical details. We have also mentioned that at the conceptual level, data is described purely in terms of its intrinsic characteristics – its logical structure.

The definition and manipulation of data happens at the conceptual level by reference to a modelling theory. The theory we will primarily be referring to in this course is called **relational theory** or the **relational model**. This remains the most common data modelling theory for database systems, although there is increasing competition from other models. The relational theory consists of:

- **concepts**, relational data objects by means of which data is modelled
- **operators**, which support the manipulation of the objects in the model
- **rules**, specifying how the concepts and operators are allowed to be combined.

Relational theory provides us with the components that we need to model information and the relationships between parts of the information. It allows us to define the types of information – such as numbers and text – that will be stored and to define constraints on it – for instance to indicate that a date in a particular context must be a past rather than a future date. These are all concepts that will be considered in more detail in the next chapter.

Once a suitable data model has been defined, it must be implemented before it can be used. A DBMS that implements relational theory to the extent that it supports the implementation of models defined using relational theory is called a **relational DBMS**. The result of implementing an abstract data model using a DBMS is a **database system** (see Figure 2.14). In practice, DBMSs do not fully implement a formal theory, and a restricted subset only will be available. This means that some data models need to be adjusted before they can be implemented as database systems.

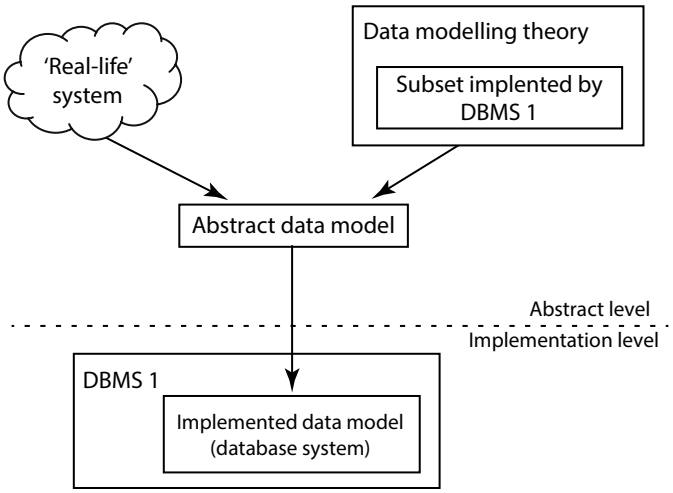


Figure 2.14. Data modelling and data development.

The relational model is by no means the only data modelling theory. Others that are relevant to DBMS implementation will be considered in Volume 2 of this subject guide.

2.9 Overview of the chapter

In this chapter, we explained the advantages of a database over less sophisticated ways of organising data, which we characterised as the file-based approach. We then defined database systems and database management systems and described the architecture and components of a standard database system. The advantages and disadvantages of such a system were considered in more detail. Finally, we introduced the idea of data modelling.

2.10 Reminder of learning outcomes – concepts

Having completed this chapter, and the Essential readings and activities, you should be able to:

- discuss the limitations of the file-based approach
- describe the way the database approach overcomes the limitations of the file based approach
- define and explain what is it meant by a database system and a database management system (DBMS)
- describe the three-level ANSI/SPARC architecture of a database system
- discuss the schemas and mappings corresponding to the three level ANSI/SPARC architecture
- discuss the concept of data independence
- explain the role of each of the components of a database system – data, hardware, software and users
- present the most important features of a DBMS
- discuss the advantages and disadvantages of database systems
- discuss issues related to distributed database systems
- explain what a data modelling theory (or data model) is and define its place within the context of database systems.

2.11 Reminder of learning outcomes – key terms

Having completed this chapter, and the Essential readings and activities, you should be able to understand the following terms:

- Data administrator (DA), Database administrator, Application programmer, End user
- Database
- Data Definition Language (DDL)
- Data dictionary (system dictionary, metadata)
- Database instance
- Database management system (DBMS)
- Data Manipulation Language (DML)
- External, conceptual and internal schema
- Intension (structure) and Extension (content)
- Logical data independence
- Physical data independence
- Program-data independence and program-data dependence
- Retrieve, Insert, Delete, Update.

2.12 Test your knowledge and understanding

2.12.1 Sample examination questions

a. Regarding the three-level ANSI/SPARC architecture:
   i. What is the conceptual level? [2]
   ii. What is the external level, and how does it relate to the conceptual level? [3]
   iii. How many mappings are defined between levels? [1]

b. What is the difference between physical and logical data independence? Which do you think is harder to achieve and why? [5]

c. A friend tells you that her company uses a file-based approach for organising, storing and sharing their data.
   i. What does she mean? [2]
   ii. Would you recommend moving to a database system? If so, why? If not, why not? [5]
   iii. How would you explain to her what a database is? [2]

d. 'Physical data independence is achieved by the creation of data model.'
   i. What does this statement mean? Is it true? [4]
   ii. What modelling theory would you expect to use to generate a model for a database system? [1]

Notes

Chapter 3: The relational model and relational DBMSs

3.1 Introduction

This chapter describes the relational model in database systems, introducing basic terminology and concepts, before considering data structures, how they are defined, and how data is added, manipulated and retrieved. The concept of data integrity is then considered.

3.1.1 Aims of the chapter

The aims of this chapter are to give you an overview of the relational model in database systems, discuss relational data objects, relational operators, data manipulation and the optimiser, before concluding with the integrity constraint definition and the foreign key rules.

3.1.2 Learning outcomes

By the end of this chapter, and having completed the Essential readings and activities, you should be able to:

- describe how a real-life system can be modelled within a data model
- describe the relational model and the way it is used in a relational DBMS; be familiar with the terminology of the relational model
- describe the concept of domains
- describe the concept of relations and discuss the properties of relations
- discuss, in general terms, how the relational data objects are operationalised (used in a relational DBMS)
- describe each of the operators of relational algebra
- be able to express natural language statements, representing information to be inferred from relations, as relational algebra expressions
- explain the way relational algebra is used in the context of DBMSs (including the optimiser)
- present different types of inconsistencies that can exist within a relational model
- define and classify integrity constraints
- define the concepts of candidate, primary, alternate and foreign key
- discuss the issue of null values
- describe how the definition of generic integrity constraints is operationalised, including the foreign key rules.

3.1.3 Essential reading

- Date, C.J. *An introduction to database systems.* (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Chapters 3, 5 and 6 (1999 edition: Chapters 3 and 5).

and/or

- Connolly, T. and C.E. Begg *Database systems: a practical approach to design, implementation and management*. (Pearson Education, 2014) 6th edition, global edition [ISBN 9781292061184 (pbk)], Chapter 4 (1999 edition: Chapter 3).

3.1.4 Further reading

- Date, Chapter 7 (1999 edition: Chapter 6).
- Elmasri, R. and S.B. Navathe *Fundamentals of database systems*. (Pearson, 2016) 7th edition [ISBN 9781292097619 (pbk)], Chapter 5.
- Ramakrishnan, R. and J. Gehrke *Database management systems*. (McGraw-Hill, 2002) 3rd edition [ISBN 9780072465631 (hbk); 9780071231510 (pbk)], Chapter 3.

3.1.5 References cited

- Codd, E.F. 'Relational Completeness of DataBase Sublanguages', in *Data Base Systems*, Courant Computer Science Symposia Series 6 (Englewood Cliffs, NJ: Prentice Hall, 1972) pp.65–98.
- Codd, E.F. 'A relational Model of Data for Large Shared Data Banks', *Communications of the ACM* 13 (6) 1970, pp.377–87.

3.2 The relational model: a general introduction

The relational model is a theory in which **all** data is modelled as **relations**; there are no records, no pointers or data structures – only relations. The concept of a relation is formally introduced later in this chapter. Until then, since tables are well suited to graphically representing relations, we shall treat 'relation' and 'table' as synonyms.

For example, information about departments and employees in an organisation might be modelled in two relations – Employees and Departments – as illustrated in Figure 3.1.

Employees			
Emp_ID	**Emp_Name**	**Salary**	**Department**
4182	A. Singh	36,003.12	Development
5542	M. Lee	39,818.15	Development
3351	T. Esterhazy	31,919.18	Development
6164	A. Barraclough	38,002.05	Marketing
9095	N. Prabakar	36,003.12	Research

Departments	
Department	**Budget**
Development	500,000
Marketing	150,000
Research	100,000

Figure 3.1. Two relations illustrated as tables.

Two conditions are assumed by the relational model that restrict the definition of a relation:

1. All data values are **atomic** (**scalar**). This means that its structure cannot be broken down into values of a more basic type. Figure 3.2 shows an incorrect relation and a corrected version. In the incorrect version, the 'Children' data value is a set of three values, and so can be further broken down; the decomposed set is used in the corrected version.

Incorrect relation		Corrected relation	
Parent	**Children**	**Parent**	**Children**
Leda	Helen	Leda	Helen
	Clytemnestra	Leda	Clytemnestra
	Castor	Leda	Castor
	Pollux	Leda	Pollux
Anakin Skywalker	Luke Skywalker	Anakin Skywalker	Luke Skywalker
	Leia Organa	Anakin Skywalker	Leia Organa
Shmi Skywalker	Anakin Skywalker	Shmi Skywalker	Anakin Skywalker

Figure 3.2. The table on the left does not represent a legal relation because the data values for Children each represent a set of data. The table on the right is altered so that all values are atomic.

2. All information must be represented as **explicit** data values. More formally, all base relations[1] are defined **extensionally**.

   Suppose, given the Parent-Child relation given in corrected form in Figure 3.2, we wish to define a new base relation called 'Ancestor'. The Ancestor relation would link fathers, grandfathers, great-grandfathers and so on to their children and grandchildren.

   Such a relationship is deterministic and easily computed based on the Parent-Child relation. A non-relational model might allow us to specify a set of logical rules to define the new base relation, but within the relational model, the relation must be explicit, as in Figure 3.3.

[1] Base relations will be defined later in this chapter in section 3.4.

Ancestor	Descendant
Shmi Skywalker	Anakin Skywalker
Shmi Skywalker	Luke Skywalker
Shmi Skywalker	Leia Organa
Anakin Skywalker	Luke Skywalker
Anakin Skywalker	Leia Organa
Leda	Helen
Leda	Clytemnestra
Leda	Castor
Leda	Pollux

Figure 3.3. The ancestor relation must include all information explicitly if it is to be used as a base relation. Relations like this can easily become very large as the family tree expands.

Clearly such a restriction could give rise to issues in practice, but there do exist other mechanisms – such as **views** – that we shall consider in later chapters which do make this restriction less problematic.

If relations are tables, then each column draws its data values from a **domain**. A relation shown as a table with three columns is defined over a set of three domains; formally, we would write this as R: D1 × D2 × D3. In general, any given relation will be defined over a set of n domains as R: D1 × D2 × … × Dn. There are only two data objects needed by the relational model: **relations** and **domains**. We will return to relational data objects later in this chapter.

Data objects are of little use unless you can do things with them. The relational model provides a set of operators for data manipulation. Two main approaches exist with respect to relational operators:

- declarative, represented by **relational calculus**; and
- procedural, represented by **relational algebra**.

In this chapter, we look only at relational algebra operators, since the most used database language, **SQL**, implements relational algebra operators.

Relational algebra operators are global (more formally, **set at a time**). This means that a single operator is considered to be applied to entire relations **at once**, with the results being returned in the form of relations.

One important relational algebra operator is the **restrict** operator, of which we will see more later. If we wanted to take the relation containing employees from Figure 3.1 and show only the higher earners, then we would use the operator as follows:

```
RESTRICT Employees SUCH THAT Salary > 37000;
```
[2]

[2] The notation used here is chosen to illustrate the operations clearly. It is based on relational algebra, but is not a standard notation. Formal relational algebra will be introduced later.

The result of this expression is, of course, a relation. It is shown in Figure 3.4.

Emp_ID	Emp_Name	Salary	Department
2	M. Lee	39,818.15	Development
4	A. Barraclough	38,002.05	Marketing

Figure 3.4. Relation resulting from a restriction operator.

The information relevant to a real-life system is modelled, within the relational model, by means of explicitly-defined relations. Since there is no such thing in the relational model as a pointer, relations that are in some way related to each other must be linked in some other way. This is achieved using **corresponding fields**, implemented using **keys**.

If you look back at the two relations shown in Figure 3.1, you will see that the Department column in both tables contains the same data values. In each relation, the Marketing department, for example, means the same thing. So the links are implemented purely in terms of data values.

Usually, a field will be constrained by restrictions on the real-world concepts they model, limiting it to a set of **correct** or **valid** values. For example, the `Salary` column in the `Employee` relation would not make sense if it contained a negative value, nor is `Emp_Name` likely to be valid with a value of "☻12✈□". This means that when devising a relational model, we do not define only the structure of the data. Limitations that constrain data to correct values must also be defined.

These defined limitations are called **integrity constraints**. In this chapter, we shall introduce two generally applicable integrity constraints: entity and referential integrity constraints.

Summarising what we have seen so far, the relational model can be defined as a way of talking about and handling data, under three categories:

- **data representation** (relational data objects)
- **data manipulation** (relational operators)
- **integrity constraints representation** (relational data integrity).

The rest of the chapter will cover these in more detail.

3.2.1 Relational DBMSs

A DBMS that implements relational theory is called, unsurprisingly a **Relational Database Management System**, or **RDBMS**. RDBMSs make up a large proportion of the DBMSs currently in use.

As we have seen in Chapter 2 of this subject guide, an RDBMS uses the relational model to allow operations to be carried out at the **conceptual level**, on relations. This abstraction protects the user from having to engage with the **internal level** and the specifics of a platform, while at the same time making migration to a new platform – and even between different DBMS implementations – easier. For a reminder about these concepts, look at Figures 2.6 and 2.9 of Chapter 2 of the subject guide.

In general, RDBMSs are so much the dominant approach that most writers simply use the term DBMS and assume that it is relational. If a different model is used it will usually be specified explicitly. From here onwards, this subject guide will follow that practice.

Despite the prevalence of relational DBMSs, it is rarely the case that any system implements the full relational theory. This must be considered when modelling and implementing databases for a specific DBMS.

3.3 Relational data objects – domains and relations

3.3.1 Terminology

- Until its formal definition, a **relation** is understood as a table.
- Each **attribute** of a relation is represented as a column in its corresponding table. **Field** is another common word for attribute.
- Each **tuple** of the relation is represented as a row in the table. In the context of relational databases, tuples are often called **records**.
- The number of attributes (i.e. columns) represents the **degree** of the relation.
- The number of tuples (i.e. rows) represents the **cardinality** of the relation.

These are core formal terms for the topic and we recommend that you familiarise yourself with them. They are illustrated below in Figure 3.5.

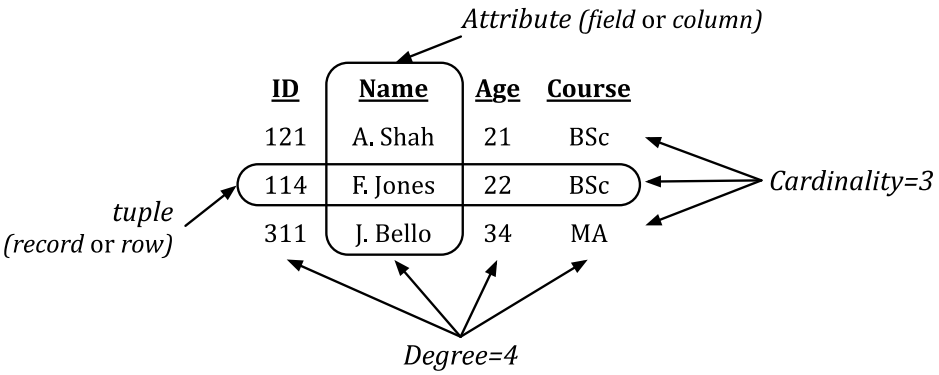


Figure 3.5. Core terminology applied to the `Students` relation. The whole table represents a *relation*, with rows as tuples and columns as attributes.

3.3.2 Domains

Date defines a domain informally as a 'pool of values' from which a given attribute draws its actual values. Every attribute of a specific relation must be defined on exactly one domain – it must draw its values from one pool, and only one pool. A domain is a set of values of the same **type**.

For example, the Students relation shown in Figure 3.5 has four attributes, each defined on a different domain. Values for the age attribute will be drawn from the positive integers {…17, 18, 19, 20, 21, … 30, 31, …40…}. Although the upper and lower limits for this domain could be debated, certainly they can be defined and there are clearly values that would not be correct.

The attribute Name will also be defined over a domain; in this case, textual strings with some criteria for legal characters and length.

One big advantage offered by domains is that they prevent certain meaningless operations from being performed.

Activity

Suppose a relation that comprises information about some rectangular products, has the following attributes: `Name`, `Weight`, `Length`, `Width`.

Consider the following questions. Which would make sense in this context and which are meaningless?

Which of these products is more than a metre long?

Which of these products are square (as long as they are wide)?

Which of these products has an area greater than 1 square metre?

Which of these products weighs more than its length?

Which products are heavier than the average product weight?

Which product is wider than it is long?

Which product has a longer name than weight?

Domains represent a means to distinguish between reasonable and meaningless queries. This can be achieved within the relational model by restricting the applicability of both scalar and relational operators. We call such restricted activities domain-constrained operations. A scalar operator (such as +) may only be applied to values of the domain it is defined on. Loosely speaking, a relational operator can be applied to two relations only if the attribute operations involved are not applied to attributes defined on different domains (these operations include scalar operators).

As an example, consider the JOIN operator, which we shall meet in more detail in a later section. This operator allows us to combine two relations based on the attributes of their respective tuples. The scalar operator associated with JOIN – and on which it relies – is **equality**. Suppose we wish to join two relations, R1 and R2. If R1 has an attribute A1 with the same name and defined on the same domain as an attribute in R2, A1 is said to be a **common attribute** and the tables can be joined. If R1 and R2 have no common attributes – even if some attributes have the same name, but are defined over different domains – they cannot be joined. The domains here constrain the applicability of the JOIN operator.

If the DBMS supports domains, then the system will prevent the user from making mistakes such as meaningless comparisons. For the example from earlier, consider the existence of two domains, Dimensions and Weight. By defining the attributes Length and Weight on the domain Dimensions and the attribute Weight on the domain Weight, we inform the system about the meaning of these attributes, and so the system should be able to enforce domain-constrained operations.

The meaningless questions in the activity above make no sense because they mix domains in an undefined way: 'Which of these products weighs more than its length?' and 'Which product has a longer name than weight?' If we use domains, we should expect that questions like these will be rejected automatically.

Before we provide a more formal definition for domains, we must first introduce the concept of atomic or scalar values.

Atomic/scalar data item. An atomic or scalar data item is the smallest semantic unit of data – it is a data item with no internal structure as far as the model or the DBMS is concerned.

The data values in the Students relation in Figure 3.5 are all scalar, although that only means that there is no internal structure **as far as the model or DBMS is concerned**. For example, in other contexts, the name 'J. Bello' might be broken up into surname and initial, while many programming languages would regard it as an ordered collection of eight separate characters: <J><.><><B><e><l><l><o>. For our purposes, however, since our model never decomposes the name, and the DBMS can represent it as an atomic structure, the fact that it **could** be broken down is of no concern to us.

As far as the **theoretical model** is concerned, data of any complexity, if viewed with no internal structure, can be considered as atomic. It is our understanding, view or assumption for a particular application that defines whether a data item is atomic. If a data type T was considered to be scalar, then that means that there is no way of accessing the constituents of data values of type T – no operator should provide access to them. For instance, if a name of a person is considered scalar, James T. Kirk, then there is no way of accessing its constituents (e.g. surname, Kirk and first names, James T.).

Domain. A domain is a named set of scalar values of the same type.

Domains are similar to data types in typed-programming languages. In a DBMS, domains are not explicitly stored within the system – they are specified as part of the database definition, in the system catalogue. **Built-in**, or **system-defined data types** – such as integer, real and string – are available in all relational DBMSs, while some provide full or partial support for **user-defined data types**, or **domains**.

The ability to specify new domains makes a DBMS more amenable to a wider variety of applications. A domain is not just a named set of values, but it also comprises a set of **operators** that are applicable to these values. For instance, integer values can be added or subtracted, while strings can be joined together or shortened.

If a domain is absent from a DBMS, then data values of that kind cannot be manipulated by the system. For example, if the system has no built-in spatial domain, and there is no way provided for defining on, then queries such as 'find shops within 4 kilometres from this point' cannot easily be carried out without first exporting all the data from the relational system and inspecting it with other software. Such a query requires the ability to store the location of points on a map and paths between them, a ≤ operator that can compare a path and a distance. A spatial database of this type would also benefit from the ability to index the points and paths in a way that ensures that retrieval is fast.

We can see that domains offer a great deal of representational power by directly increasing the modelling capabilities of the database. Even where user-defined types are available, having a rich set of built-in scalar types is helpful. We shall see that some of the perceived drawbacks of the relational model are based on the poor implementation of mechanisms such as these.

To summarise, before we move on to **relations**, the primary benefits of domains are:

- domain-constrained operations
- increased modelling power.

3.3.3 Relations

The relation is the only data structure used for modelling data in the relational model. So far, we have treated a relation as being equivalent to a table. A more formal definition is now necessary.

Relation. A **relation**, R, on a set of domains $D_1, \ldots, D_n$ consists of two parts, a **heading** and a **body**.

1. **Heading**
   a. The **heading** consists of a fixed, unordered set of **attributes**.
   b. Each **attribute** is described by <attribute-name : domain-name> pairs, $\{<A_1 : D_1>, <A_2 : D_2>, \ldots, <A_n : D_n>\}$.
   c. Each **attribute** A_i corresponds to exactly one of the underlying **domains** D_i.
   d. Attribute names are distinct – they may not be repeated.
   e. The domains $D_1, \ldots, D_n$ need not be distinct – they may be reused.
2. **Body**
   a. The **body** consists of a set of (unordered) **tuples**.
   b. Each **tuple** is a unique set of <attribute-name : attribute-value> pairs.
   c. In each tuple i, there is exactly one $<A_j : V_{ij}>$ pair for each attribute A_j in the heading, so that the i^{th} tuple of a relation with attributes $A_1 \ldots A_n$ can be represented as $\{<A_1 : v_{i1}>, <A_2 : v_{i2}>, \ldots, <A_n : v_{in}>\}$.
   d. For any given $<A_j : v_{ij}>$ pair, v_{ij} is a value from the domain D_j that is associated with the attribute A_j.

This definition can seem complicated, but much of that comes from its precision rather than the ideas being hard to understand. Two examples should help to make this clear; firstly, a correct example of a relation. Consider the table Students in Figure 3.6.

ID	Name	Age	Degree
AS191	A. Turing	21	CS
BC24Z	A. Lovelace	28	CS
AX007	F. Allen	22	CIS
NN02M	M. Ibuka	21	IT

Figure 3.6. The `Students` table.

The relation this table illustrates can also be represented as in Figure 3.7. The attributes of the relation are `ID`, `Name`, `Age` and `Degree`. `ID`, an identifier that is unique for each student, is necessary to ensure that the tuples are distinct, even if two students share the same name, age and degree. The heading of the relation is represented here as a set of pairs <`name : type`>. We shall meet the types used in the next chapter, but for now, it is enough to note that `CHAR` and `VAR-CHAR` are string types and `INT` is an integer.

Students relation

Heading

{<Id: CHAR(5)>, <Name: VAR-CHAR>, <Age: INT>, <Degree: CHAR(3)>}

Body

{

{<Id: "AS191">, <Name: "A. Turing">, <Age: 21>, <Degree: "CS">},

{<Id: "BC24Z">, <Name: "A. Lovelace">, <Age: 28>, <Degree: "CS">},

{<Id: "AX007">, <Name: "F. Allen">, <Age: 22>, <Degree: "CIS">},

{<Id: "NN02M">, <Name: "M. Ibuka">, <Age: 21>, <Degree: "IT">}

}

Figure 3.7. The `Students` relation – a different, but still correct, representation.

Neither the order of the pairs in each set nor the order of the tuples in the body matters, so the representation of Students in Figure 3.8 is also correct.

Students relation

Heading

{<Id: CHAR(5)>, <Name: VAR-CHAR>, <Degree: CHAR(3)>, <Age: INT>}

Body

{

{<Id: "AS191">, <Name: "A. Turing">, <Age: 21>, <Degree: "CS">},

{<Degree: "CS">, <Id: "BC24Z">, <Name: "A. Lovelace">, <Age: 28>},

{<Age: 22>, <Id: "AX007">, <Name: "F. Allen">, <Degree: "CIS">},

{<Name: "M. Ibuka">, <Id: "NN02M">, <Age: 21>, <Degree: "IT">}

}

Figure 3.8. The `Students` relation – different ordering, but still the same, correct, representation.

Relations have some important properties arising from the definition:

- **A relation cannot contain duplicate tuples**. Each tuple is unique within a given relation. This also means that each tuple is uniquely identifiable. The smallest set of attributes that can be used to uniquely identify all the tuples in a relation is called a **candidate key**.
- **Tuples are unordered**. Statements like 'the fifth tuple' make no sense (this is also a reason why the uniqueness criterion is vital).
- **Attributes are unordered**.
- **All attribute values are atomic**. This property dictates the **first normal form**. We shall meet the normal forms again later in the next chapter, but by definition, all relations are in the first normal form.

From the first three of these, we can see the difference between tables and relations and why taking them as equivalent was an approximation. The table in Figure 3.9 lists these differences.

Table	Relation
Rows are ordered	Tuples are unordered
Columns are ordered	Attributes are unordered
Duplicate rows are permitted	Duplicate tuples are not permitted
Duplicate column names are permitted	Duplicate attribute names are not permitted

Figure 3.9. Differences between tables and relations.

The requirement for attribute values to be atomic can be explained with an example. Consider two ways of representing tutors and their tutees. Figures 3.10 and 3.11 show an unnormalised and a normalised representation (only the latter is a relation).

Tutor	**Tutees**	
M. Taylor	**Name**	**Age**
	P. James	22
	S. Saldin	21
	J. Bentham	22
A. Rai	**Name**	**Age**
	P. Philips	19
	K. Khan	22
	S. Pereira	24

Figure 3.10. A table (but not a relation) showing tutors and their tutees.

Tutor	**Tutee Name**	**Tutee Age**
M. Taylor	P. James	22
M. Taylor	S. Saldin	21
M. Taylor	J. Bentham	22
A. Rai	P. Philips	19
A. Rai	K. Khan	22
A. Rai	S. Pereira	24

Figure 3.11. A table showing a normalised (first normal form) relation of tutors and tutees.

Now, suppose that two tasks were to be performed on the data:

- Add that a new tutor, P. Rosin, was assigned a tutee, P. Black, aged 26.
- Add that M. Taylor has been assigned a new tutee, H. Higgins, aged 23.

Each task can be solved in the normalised version in Figure 3.11 by adding a new tuple, so the two tasks are logically similar in this representation. For the table in Figure 3.10, though, the two tasks are quite different. The first only involves the insertion of a new row (or, by extension, a tuple), but the second requires the retrieval of the row for M. Taylor, and the insertion of the row `{<Name : H. Higgins>,<age : 23>}` into the current value for `Tutees`.

So, if the relation is not normalised, it is difficult to guarantee that operations will be simple, and they can require multiple, nested procedures to be carried out. In the relational model, a relation is constrained to be atomic, so that any more complex structures must be made explicit through the use of multiple relations.

At this point, it is important to be clear about the distinction between a relation **value** and a relation **variable**. This is similar to the distinction between a value and a variable in a programming language. In the example of Figure 3.12, `example_var` is declared as a **variable** of type integer, which means that it can hold any **value** as long as it is an integer (and as long as it is between the maximum and minimum values allowed by the language).

```
/* this is a fragment of a C program
   illustrating values and variables */
int example_var;
example_var = 10; /* example_var is assigned the integer value 10 */
example_var = example_var * 2; /* value of example_var is doubled */
```

Figure 3.12. Values and variables in a programming language.

When we talk about the attributes of a relation, we can see a similar process. A relation is defined on creation, and then its heading, declaring the relation's set of attributes and their domains, is somewhat like the variable declaration. A particular state of that relation, with a set of tuples, is a **value** assigned to that **variable**, just as 10 is the first value assigned to the variable `example_var` in Figure 3.12. Figures 3.6–8 show a **relational variable** – `Students` – and its first assigned relational value – the contents of the relation. Any insertion, deletion or alteration to the data in `Students` will change the relational **value**, but the definition of the variable itself stays the same, just as `example_var` is still `example_var`, and still declared as an integer, even when its value changes.

It might appear that the relations in a relational database are independent, in that there is no way to relate one to another – they can only contain explicit data values rather than language constructs like pointers or variables. That appearance is deceptive – relations in a database can be logically connected through matching attributes, called **keys**. If several relations are to be linked by an attribute or a set of attributes that are common to all the relations, in one of the relations the key is called the **candidate key** and in the other relations that link to it, the key is called the **foreign key**. In Figure 3.1, the relation `Employees` is linked to the relation `Departments` via the common attribute, `Department`. If one wants to find out the details of the department in which a specific employee works then one will identify the Department of that person from `Employees` and then look it up in the relation `Departments`. For instance, `T. Esterhazy` works in the department identified by `Development` (from the Employees relation). The department identified by `Development` has a budget of £500,000 (from the `Departments` relation). `Department` in `Employees` is a **foreign key** whereas in `Departments`, it is a **candidate key**.

3.4 Data definition in a relational DBMS

In his 1970 article that laid out the basic concepts of relational theory, Codd defined a relational database as '**a database that is perceived by the user as a collection of normalised relations of assorted degrees**.'[3] Such a relational database must provide a **Data Definition Language**, or DDL, that supports the definition of domains and relations (that is, relation **variables**) as a minimum requirement. One such language is SQL, the subject of the next chapter. SQL is a relational database language that provides a DDL, a small subset of which is illustrated below.

[3] *Codd, E.F. 'A relational Model of Data for Large Shared Data Banks,'* Communications of the ACM *13(6) 1970, pp.377–87.*

In defining the syntax of the language, we shall use the following notation conventions:

- everything that is not within < and > signs (which will always be either numerals, symbols or capital letters) is a terminal or a keyword symbol, which means it has to be unchanged
- symbols between < and > are non-terminals and have to be replaced with a corresponding value (for instance, <domain name> should be replaced with a user-defined name for a domain).

A simplified version of the syntax for SQL's data definition language is:

```
CREATE DOMAIN <domain name> AS <definition>;
CREATE TABLE <table name> {
    <attribute name>    <domain name>,
    <attribute name>    <domain name>
};
```

Activity

Note how SQL implements relations as tables. What effect might this have on the implementation? What differences would you expect to see?

Using this syntax, the relation `Students` (see Figures 3.6–8) could be defined as follows:

```
CREATE DOMAIN IdDomain      AS CHAR(5);
CREATE DOMAIN NamesDomain   AS VARCHAR;
CREATE DOMAIN AgeDomain     AS INT;
CREATE DOMAIN DegreesDomain AS CHAR(3);
CREATE TABLE Students {
   ID      IdDomain,
   Name    NamesDomain,
   Age     AgeDomain,
   Degree  DegreesDomain
};
```

A relation, once defined, would have to be populated with data, so the DBMS must provide at least two further statements – this time as part of the **Data Manipulation Language**, the DML. These statements are for adding and deleting tuples.

A simplified version of the syntax for SQL's INSERT and DELETE statements is:

```
INSERT INTO <table> VALUES <row>;
DELETE FROM <relation> WHERE <condition for identifying
tuples>;
```

And some examples of their use in the `Students` relation:

```
INSERT INTO    Students
   VALUES    ("AS191", "A. Turing", 21, "CIS");
INSERT INTO    Students
   VALUES    ("BC24Z". "A. Lovelace", 28, "CIS");
DELETE FROM    Students
   WHERE     ID = "AS191";
```

Note that SQL's table implementation means that attribute values can be specified in order.

It is possible that certain relations or domains might not be needed any longer in the database, and so there needs to be a mechanism for removing them. In SQL, the (simplified) syntax is as follows:

```
DROP DOMAIN <domain name>;
DROP TABLE  <relation name>;
```

An important characteristic of a relational DBMS is that both the external and the conceptual levels are based on the same data model; that is, **data is perceived at both levels as relations**.

Relations can be classified by the source of their values, whether or not they can be identified by name, and whether they necessarily persist in memory. Relations that must exist in a relational system are:

- **Named relations**: relations that have been defined and named within a database system.
- **Base relations**: named relations defined purely in terms of their extension – that is, for which all data values are explicitly provided.

- **Derived relations**: relations defined in terms of other relations – whether base or derived themselves – by using a relational expression.
- **Views**: named, derived relations whose extension (values) are not stored in the system. A view can be considered a **virtual** relation.
- **Snapshot**: named, derived relation, but whose extension is stored in the system after it is computed.
- **Query result**: unnamed, derived relation with no persistent existence within the system.

3.4.1 The data dictionary

As described in the previous chapter, the database includes the description of its raw data in what is called the **data dictionary** or catalogue (sometimes using the US spelling, 'catalog').

The data dictionary contains all kinds of information describing the database: schemas, mappings, integrity rules, security rules, etc. The data dictionary is itself a part of the database, and so it is also represented by means of relations (usually tables). They are called system relations or tables, in order to differentiate them from userdefined relations or tables. The information stored in the data dictionary is useful to some of the modules of a relational DBMS, but can be used in the same way as any other part of the database.

`Relations`

Relation name	Degree	Cardinality	…
Students	4	1587	…
…	…	…	…
Relations	7	39	…
Attributes	7	243	…

`Attributes`

Relation name	Attribute name	…
Students	ID	…
Students	Name	…
Students	Age	…
Students	Degree	…
…	…	…
Relations	Relation name	…
Relations	Degree	…
Relations	Cardinality	…
Attributes	Relation name	…
…	…	…

Figure 3.13. The Relations and Attributes relations of a data dictionary, showing how they describe not only the user-defined tables, but themselves also.

For example, every DBMS data dictionary usually contains two relations describing all the named relations in the database – Relations (or Tables) and Attributes (or Columns). This is illustrated in Figure 3.13, which also shows how they are self-describing – they contain information about themselves and each other.

3.5 Relational operators

We have seen so far how the structure of data can be defined. Now, we present **relational operators**, which allow data to be manipulated. There are two main approaches to relational operators:

- **Procedural operators**, using these we can prescribe **how** a result is going to be obtained
- **Declarative operators**, using which we can prescribe **what** result is to be obtained.

In the procedural approach, we have to state **which operations** are to be performed on data, and **the order** in which they are to be performed – we describe how the result is to be computed.

In the declarative approach, we do not specify explicitly how the result should be computed. Instead, we describe **what the result looks like**. We do this by stating the **conditions** that the result should satisfy.

So, in a procedural system, we tell the system what to do and in a declarative system, we tell it what result we want. In the latter case, it is for the system to work out what operations must be performed to compute the result.

The relational model includes these approaches as **relational algebra**, corresponding to the procedural approach, and **relational calculus**, which uses the declarative approach. The two approaches are equivalent to one another, in that any expression in the relational algebra has an equivalent in relational calculus, and vice versa. The two formalisms are different only in style of expression and philosophy of approach.

Since SQL is based on relational algebra, the rest of this section will be dedicated to this, and so to procedural operators.[4]

Relational algebra, as described by Codd (1972),[5] consists of eight basic operators.

- Four based on set theory: **union**, **intersection**, **difference** and the **cartesian product**.
- Four relation-specific operators: **restrict**, **project**, **join** and **divide**.

We will return to each of these operators in more detail soon, but first, two important properties of relational algebra must be considered.

- **Set at a time operators**: this property refers to the fact that relational algebra operators act **globally** on relations – relations **as a whole** constitute their operands and not individual tuples. This is related to our definition earlier of all the data in a relation as a single **relational value**.
- **Relational closure**: refers to the fact that the result of the application of any relational operator is a relation, too. This means that relational operators can be used in devising complex expressions, since the result of the application of one operator can become the input for another operator.

A comparable situation arises in real number algebra. Consider three arithmetic operators: +, - and *. They are applied to real numbers and result in real numbers. This means, for example, that given real-number variables `x`, `y`, `z`, `u` and `v`, we can first construct complex expressions such as:

x*(2*x + 2*y) + z

and

2*u*v – (u+v).

[4] Date devotes a chapter to relational calculus, should you wish to learn more (Chapter 7; or Chapter 6 in the 1999 edition).

[5] Codd, E.F. 'Relational Completeness of Data Base Sublanguages', in Data Base Systems*, Courant Computer Science Symposia Series 6 (Englewood Cliffs, NJ: Prentice Hall, 1972), pp.65–98.*

Because of the closure property, these two expressions can be used in other expressions – they can become the input of other operators. Similarly, the first expression can be used as the input `u` in the second expression, giving:

2*(x*(2*x + 2*y) + z)*v – ((x*(2*x + 2*y) + z) + v).

These equations are meaningless out of context, but the power that real number algebra gains from the closure property is something that you will have experience with from mathematics. Relational algebra expressions are constructed in a similar way. A relational algebra expression denotes a relation, and so it can constitute, as a whole, the operand of any relational algebra operator.

These two properties of relational algebra provide the theory with a very powerful formalism.

3.5.1 Relational algebra operators based on set theory

The set specific relational operators are almost identical to the set operators from Set theory. Relations are a special kind of set, though, and the operators have to be restricted a little to suit them, and particularly to ensure relational closure.

- Except for the Cartesian product, they assume **type compatibility**.
- They are accompanied by a mechanism for inheriting attribute names.
- They are accompanied by a mechanism for inheriting candidate keys.

In simple terms, type compatible relations are ones that are directly comparable. More formally:

Type compatibility. Two (or more) relations are type compatible if their headings are functionally identical, having:

1. The same set of attribute names.
2. The same domains applying to attributes with the same name.

An example of two type compatible relations is given in Figure 3.14. This example will be used to illustrate the set-specific operators as we introduce them.

`Relation1`

ID	Name	Age	City
S1	A. Braun	22	London
S2	T. Elliot	21	London
S3	Y. Dhillon	22	Singapore

`Relation2`

ID	Name	Age	City
S1	A. Braun	22	London
S4	F. Williams	19	Port of Spain

Figure 3.14. Two relations which are type compatible.

`UNION`

`Relation1` ∪ `Relation2`

ID	Name	Age	City
S1	A. Braun	22	London
S2	T. Elliot	21	London
S3	Y. Dhillon	22	Singapore
S4	F. Williams	19	Port of Spain

Figure 3.15. Applying `UNION` to the relations returns all the tuples present in either `Relation1` or `Relation2` or both.

The union, intersection and difference operators can only be applied to type-compatible relations, and the result of applying them is a relation that is also type compatible with the operands. Figures 3.15, 3.16 and 3.17 illustrate the application of union, intersection and difference respectively on the relations in Figure 3.14.

`INTERSECTION`

`Relation1` ∩ `Relation2`

ID	Name	Age	City
S1	A. Braun	22	London

Figure 3.16. Applying `INTERSECTION` to the relations returns all the tuples present in both `Relation1` and `Relation2`.

`DIFFERENCE`

`Relation1 – Relation2`

ID	Name	Age	City
S2	T. Elliot	21	London
S3	Y. Dhillon	22	Singapore

Figure 3.17. The `Difference` of `Relation1` and `Relation2` is a relation that contains only tuples of `Relation1` that are absent from `Relation2`. Note that the operator is not symmetrical, and `Relation2` – `Relation1` will yield a different result.

For the Cartesian product, the type compatibility restriction is not necessary. The Cartesian product may be applied to any two relations. An example of the Cartesian product operator is provided in Figure 3.18, using a new relation and `Relation2` from above. Since the Cartesian product has a cardinality (number of tuples) that is the product of the cardinality of the two operands and a degree (number of attributes) that is the sum of the degrees of the operands, this can produce large relations very quickly.

`Relation3`

Letter ID	Letter description
L1	Registration
L2	Reward

CARTESIAN PRODUCT

Relation2 × Relation3

ID	Name	Age	City	Letter ID	Letter description
S1	A. Braun	22	London	L1	Registration
S1	A. Braun	22	London	L2	Reward
S4	F. Williams	19	Port of Spain	L1	Registration
S4	F. Williams	19	Port of Spain	L2	Reward

Figure 3.18. The Cartesian product of Relation2 and Relation3.

Although type compatibility does not apply for the Cartesian product, attribute name inheritance does, and can be seen in the example above – the resulting relation inherits the names of all the attributes from both Relation2 and Relation3.

3.5.2 Relation-specific relational algebra operators

The other four basic relational algebra operators are: **restriction**, **projection**, **join** and **division**. These operators will be illustrated using the Product-Details relation shown in Figure 3.19.

Product-Details

ProductID	ProductType	Cost	InStock	Supplier
PID23	Washing machine	289	2	E-A inc.
XX24A	Dishwasher	399	0	H200
00012	Power extension cord	14.99	15	E-A inc.
MM25y	Television	395	0	S-TV
MM45x	Television	555	0	S-TV

Figure 3.19. The Product-Details relation.

Restriction

Restriction. A restriction operator[6] is applied to a single relation R – it is a **unary operator** – and produces another relation R′ according to a condition C, so that:

1. R′ is type compatible with R.
2. All the tuples of R′ are from R.
3. All the tuples of R′ satisfy C.
4. There are no tuples in R that satisfy C and are not in R′.

[6] *You may see this operator called the **selection operator**; however, restriction is Codd's original term, and SQL uses* SELECT *to mean something quite different.*

The condition C is expressed on one or more of the attributes of R by means of some scalar comparison operators. For instance, the relation Product-Details can be restricted to only those products that are currently available, as indicated by an inStock value greater than zero. The result is the relation shown in Figure 3.20.

ProductID	ProductType	Cost	InStock	Supplier
PID23	Washing machine	289	2	E-A inc.
00012	Power extension cord	14.99	15	E-A inc.

Figure 3.20. The relation resulting from a restriction of Product-Details to available products.

The syntax for the restriction operator is:

```
<relation name> WHERE <condition>
```

where `<condition>` is a conditional expression – one which returns a truth value – on the attributes of the relation `<relation name>`. For instance, the 'currently available' restriction described above for `Product-Details` can be defined as:

```
Product-Details WHERE InStock > 0
```

The condition may be of any complexity. A slightly more complicated example might be to restrict the results to items where the value of the items in stock is above 500. This would be expressed as:

```
Product-Details WHERE Cost*InStock > 500
```

The conditional expression in this case involves a multiplication between a currency-type value and an integer value (the result being of currency type) and then a comparison with a currency-type value.

As you may have realised, the conditional expressions you can use are determined by the attribute domains which, in turn, determine the available scalar operators. For instance, the integer type includes the following operators: +, -, *, / (integer division), and comparison operators such as >, < and =. The string type can provide operators such as: **concatenation** (of two strings), **division** (of one string into two substrings) and comparison operators such as **inclusion** (of a string in another) and **equality** (of two substrings).

A scalar operator can only be applied to values corresponding to the domain it is defined on, so, for instance, the multiplication operator can be applied to values of type integer or real, but cannot be applied to string values or dates.

According to the definition above, the restriction operator can only use one conditional expression, which we have defined as using one single scalar comparison operator. If we wanted to combine several of these 'atomic' conditions, you might expect to have to write:

```
(Relation WHERE Condition1) WHERE Condition2
```

This would take advantage of the **relational closure** property of the operators to allow them to be used as operands for each other. This is a little ugly, though, and could quickly get quite long-winded. Worse, it is not very expressive. Instead, the syntax of relational algebra allows **non-atomic conditions** to be used in conjunction with the restriction operator. These non-atomic conditions combine conditional expressions by using logical operators (AND, OR, NOT). So, for instance, the above nested expression can be expressed as:

```
Relation WHERE Condition1 AND Condition2
```

To return to the `Product-Details` example, the following expression restricts the relation to those products that have the string "MM" in their ID or are supplied by S-TV, and whose current stock is greater in value than 500.

```
Product-Details WHERE ("MM" SUBSTRING_OF ProductID
   OR Supplier = "S-TV")
   AND Cost*InStock > 500
```

The use of brackets in the expression above is vital for getting the subclauses to be evaluated in the correct order. Bracketed expressions are always evaluated first. Without brackets to show the order, `AND` expressions are evaluated before `OR` expressions, which would give the wrong answer in this case.

Projection

Projection. Given a relation R having attributes {A, B, …, L, M, …, X, Y, Z}, a projection of R on A, B, …, L is a relation R' having attributes {A, B, …, L} and the tuple {A:ai, B:bi, …, L:li} for each tuple in R.

Usually, but **not always**:

5. R and R' have the same **cardinality**.
6. The degree of R' is smaller than the degree of R.

In visual terms, treating a relation as a table, where a restriction is a **horizontal** sub-relation, returning a relation with fewer rows than its operand, projection is a **vertical** one, returning a relation with a reduced number of columns. The benefit here is to leave aside the attributes of a relation that are not relevant for the current purpose. For instance, if you wanted to select only product types, stock levels and suppliers, you could **project** the Product-Details relation onto the ProductType, InStock and Supplier attributes, yielding the relation in Figure 3.21.

ProductType	InStock	Supplier
Washing machine	2	E-A inc.
Dishwasher	0	H200
Power extension cord	15	E-A inc.
Television	0	S-TV

Figure 3.21. A relation resulting from a projection of `Product-Details`**.**

The syntax for the projection operator is:

```
<relation name> [<attr name 1>, <attr name 2>, ..., <attr name n>]
```

where <attr name 1> (and so on) should all be attributes of the relation denoted by <relation name>. For instance, the projection above is described by the following expression:

```
Product-Details [ProductType, InStock, Supplier]
```

It is important to note that, since the result of a projection is a relation, every tuple must be unique. Formal relational algebra requires that any duplicates are omitted, as is the case in Figure 3.21 where the two tuples describing television sets are no longer unique and only one is presented. This means that cardinality is not always preserved.

Projections may use all the attributes of the operand, in which case the relation is duplicated (and so the degree of R' is **not** smaller than the degree of R). A projection on no attributes produces an empty, or **null** relation.

Join

The join operator, as its name suggests joins two relations together. It is similar to the Cartesian product, but permits constraints that may limit the order and cardinality of the resulting relation.

There are two types of join operators:

- The **Natural-join** operation combines relations based on common attributes, including only tuples where the values of those attributes are the same in the two tables. Figure 3.22 shows a Suppliers relation for the items in Product-Details. Figure 3.23 shows the natural join of the two relations. Note that the **degree** of the result is the sum of the degrees of the joined tables minus the number of common attributes, while the **cardinality** remains the same as it is in Product-Details.

Supplier	City	Email
E-A inc.	Pittsburgh	orders@e-a-inc.com
E&E GMBH.	München	kontakt@e-und-e.de
H200	London	p.parker@h200.co.uk
S-TV	Tokyo	s-tv@stv.co.jp

Figure 3.22. The `Suppliers` relation.

ProductID	ProductType	Cost	InStock	Supplier	City	Email
PID23	Washing machine	289	2	E-A inc.	Pittsburgh	orders@e-a-inc.com
XX24A	Dishwasher	399	0	H200	London	p.parker@h200.co.uk
00012	Power extension cord	14.99	15	E-A inc.	Pittsburgh	orders@e-a-inc.com
MM25y	Television	395	0	S-TV	Tokyo	s-tv@stv.co.jp
MM45x	Television	555	0	S-TV	Tokyo	s-tv@stv.co.jp

Figure 3.23. The natural join of `Product-Details` and `Suppliers`.

- The **Θ-join (theta-join)** operation[7] is a Cartesian product with a condition that restricts the resulting relation.

[7] *Θ is a capital letter from the Greek alphabet, called theta. The easiest way to write it is to draw an O and put a small horizontal line inside.*

The natural join is the simpler and more common operator, when you see reference to 'join' without a qualifier to indicate whether it is a natural or theta join, it is likely that a natural join is meant. A more formal definition is as follows:

Natural join. Let relations `R1` and `R2` have the headings {`X1, X2, X3, …, Xm, Y1, Y2, Y3, …, Yn`} and {`Y1, Y2, Y3, …, Yn, Z1, Z2, Z3, …, Zp`} respectively (i.e. they have some attributes – `Y1`, `Y2`, etc. – in common, and some different). The natural join

R1 JOIN R2

is a relation having the heading {`X1, X2, X3, …, Xm, Y1, Y2, Y3, …, Yn, Z1, Z2, Z3, …, Zp`} and body consisting of the set of all tuples {`X1:x1, X2:x2, X3:x3, …, Xm:xm, Y1:y1, Y2:y2, Y3:y3, …, Yn:yn, Z1:z1, Z2:z2, Z3:z3, …, Zp:zp`} where the tuple {`X1:x1, X2:x2, X3:x3, …, Xm:xm, Y1:y1, Y2:y2, Y3:y3, …, Yn:yn`} is present in `R1` and {`Y1:y1, Y2:y2, Y3:y3, …, Yn:yn, Z1:z1, Z2:z2, Z3:z3, …, Zp:zp`} is present in `R2`.

Two properties of the join operator that can be derived from this definition are:

- R1 JOIN R2 = R2 JOIN R1 (i.e. JOIN is commutative).
- (R1 JOIN R2) JOIN R3 = R1 JOIN (R2 JOIN R3) (i.e. JOIN is associative).

Θ-**join**. Let `R1` and `R2` be relations such that attribute `X` belongs to `R1` and `Y` to `R2`. If Θ is an operator such that x Θ y is a conditional expression for all values of X and Y, then the Θ-join of `R1` and `R2` is defined as:

(R1 TIMES R2) WHERE X Θ V

For example, Figure 3.24 shows two relations, `Deliveries` and `Vehicles`. If we wish to see all the vehicles capable of carrying each delivery, we need to check the size of the delivery and the capacity of the vehicle. This can be achieved with a Θ-join operator expressed as follows:

```
(Deliveries TIMES Vehicles) WHERE Volume < Capacity
```

The results of this operation are shown in Figure 3.25.

DeliveryID	Volume	Destination	VehicleID	Type	Capacity
D1	200	London	V1	Motorbike	50
D2	5130	Glasgow	V2	Car	1100
D3	1050	Cowes	V3	Van	20000

Figure 3.24. `Deliveries` **(left) and** `Vehicles` **(right) relations.**

DeliveryID	Volume	Destination	VehicleID	Type	Capacity
D1	200	London	V2	Car	1100
D1	200	London	V3	Van	20000
D2	5130	Glasgow	V3	Van	20000
D3	1050	Cowes	V2	Car	1100
D3	1050	Cowes	V3	Van	20000

Figure 3.25. The relation resulting from a Θ-join on `Volume` **<** `Capacity`**.**

Note that a Θ-join can become a natural join if Θ is the = operator and a projection is applied to remove duplicate attributes.

Division

The division operator is a little harder to describe than the others that we have met so far, and is best illustrated with an example. Consider the students whom we met first in Figure 3.6. Figure 3.26 shows three relations: `Passed-Courses`, which records courses that students have completed successfully; `Degree-Requirements`, which lists the passes required for a Bachelor's degree; and `Honours-Requirements`, listing the courses that should be passed for a student to qualify for a degree with honours.

Passed-Courses		Degree-Requirements	Honours-Requirements
Student	**Course**	**Course**	**Course**
AS191	Databases2	Programming1	Programming1
AS191	Programming1	Databases2	Databases2
AS191	Music3	Robotics3	Robotics3
BC24Z	Programming1		Programming3
BC24Z	Databases2		
BC24Z	Robotics3		
AX007	Programming1		
AX007	Robotics3		
AX007	Databases2		
AX007	Programming3		
NN02M	Programming1		

Figure 3.26. Three relations about music and computing courses (see courses).

The division operator takes two relations and returns a relation with only the attributes of the first relation that are not in the second. The body of the result contains only entries that have tuples in the first operand that match **all** the tuples in the second operand.

Using the division operator, we can see which students have passed all the courses required for a Bachelor's or Honours degree, since the operator will return only student IDs for students for whom tuples exist in `Passed-Courses` for every course listed in the requirements tables. Figure 3.27 shows the result of the two division operations, `Passed-Courses` ÷ `Degree-Requirements` and `Passed-Courses` ÷ `Honours-Requirements`.

`Passed-Courses` ÷ `Degree-Requirements`	`Passed-Courses` ÷ `Honours-Requirements`
Student	**Student**
BC24Z	AX007
AX007	

Figure 3.27. The division operation returns only entries from `Passed-Courses` having all the entries of the requirements table in their tuples. So the results are a list of students meeting the relevant degree criteria, having passed all the necessary courses.

More formally, we can define division as follows:

Division. Let relations `R1` and `R2` have the headings {`X1, …, Xm, Y1, …, Yn`} and {`Y1, …, Yn`} respectively, so that `Y1…Yn` are common attributes of `R1` and `R2` – they have the same name and domains in both. `R2` necessarily has no attribute that is not in `R1`. Let *k* be the cardinality of `R2`. The result of the division of `R1` by `R2`, written as:

R1 DIVBY R2

or

R1 ÷ R2

is a relation `R3`, having the heading {`X1, …, Xm`} and the tuples {`X1:x1, …, Xn:xm`} such that the tuples {`X1:x1, .., Xn:xm, Y1:y11, ..., Yn:Yn1`}, {`X1:x1, .., Xn:xm, Y1:y12, ..., Yn:Yn2`}, …, {`X1:x1, .., Xn:xm, Y1:y1k, ..., Yn:Ynk`} appear in `R1` for all tuples {`Y1:y11, ..., Yn:Yn1`}, {`Y1:y12, ..., Yn2`}, …, {`Y1:y1k, ..., Ynk`} or `R2`.

Relational operators: conclusions

These eight operations are a set of operators that are useful and make relational theory powerful in practice. They are not a minimal set of primitives – some of these operators can be defined in terms of the others. Where they are not primitive, they have usually been included to simplify common expressions. For example, both the theta- and the natural-join operators can be expressed in terms of the Cartesian product, restriction and projection, but joins are so common and important, that expressions would become over-complicated quickly without them. In fact, the minimal set of operations contains only five operators – **restriction**, **projection**, **Cartesian product**, **union** and **difference**.

The power of relational algebra comes from its ability to manipulate relations as a whole. The **relational closure property** allows for an unlimited set of statements to be expressed by combining the eight basic operators. A grammar for relational algebra expressions can be found in **Date**, Chapter 'Relational Algebra', section 'Syntax'.

To illustrate the power of this formalism, we shall consider some further examples.

3.5.3 Examples

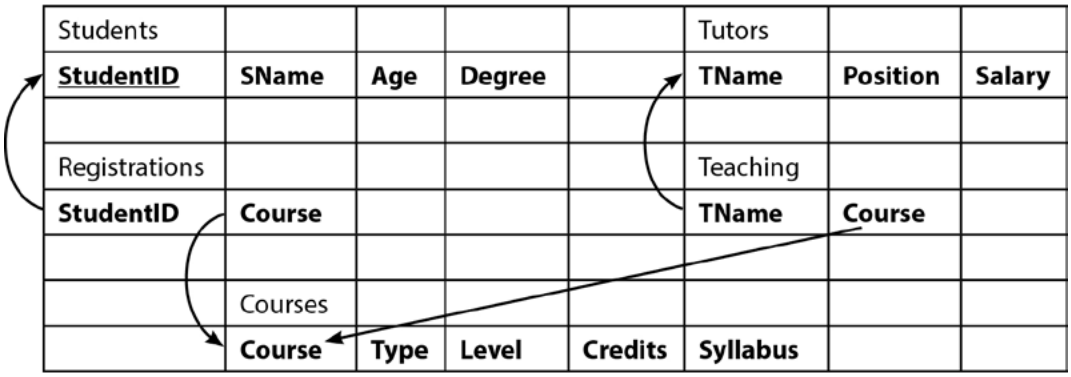


Figure 3.28. The set of relations for the examples in this section.

For these examples, we return to the Students relation and add some further relations (see Figure 3.28). In the expressions that follow, the meaning of the results should be clear and so no values are provided, only attributes. If you have difficulty following any of these examples, try making up some sample values and working them through, as an exercise.

Most of these attributes should be self-explanatory. The `Type` attribute of `Modules` takes the value of either compulsory or optional.

Here are some example tasks using these relations.

1. Get the name of the tutors who teach at least one module.

   ```
   Teaching[TName]
   ```

2. Get the name of tutors who do not teach any module.

   ```
   Tutors[TName] – Teaching[TName]
   ```

3. Get the name, position and salary of the tutors who do not teach any module.

   ```
   (Tutors[TName] – Teaching[TName]) JOIN Tutors
   ```

4. Get all the modules that are taught by Professors or give 1 course credit.

   ```
   ((Tutors WHERE Position="Professor") JOIN Teaching)[Course]
   ∪ (Courses WHERE Credits=1)[Course]
   ```

5. Get the name and the age of all students who take level 1 courses.

   ```
   ((Courses WHERE Level=1)
   JOIN Registration JOIN Students)[SName, Age]
   ```

6. The same result could also be obtained using the following:

   ```
   ((Courses WHERE Level=1)[Course] JOIN Registration)[SName]
   JOIN Students[SName, Address]
   ```

7. Get the name of all students who take all the optional courses.

   ```
   (Registration ÷ ((Courses WHERE Type="Optional")[Course]))[SName]
   ```

8. Get the names and the syllabuses for all the courses taken by the student "A. Lovelace" together with the name and position of the tutor who teaches each course.

   ```
   ((Registrations WHERE SName="A. Lovelace")
     JOIN Courses)[Course, Syllabus]
   JOIN
   (Tutors[TName, Position] JOIN Teaching)
   ```

   Another possible solution is as follows:

   ```
   ((Registrations WHERE SName="A. Lovelace")
   ```

```
    JOIN Courses JOIN Teaching JOIN Tutors)
[Course, Syllabus, TName, Position]
```

Activity

Use the examples above to help formulate your own practise queries. First write a statement in natural language clearly stating what information you want to retrieve and then see how many ways you can satisfy that query in relational algebra. The more complicated the query, the more chance there is that there will be more than one way of solving it.

3.6 Data manipulation and the optimiser

We have seen that each relational DBMS should provide a language by means of which to define the domains and relations of a relational model (DDL). In order to retrieve and update data, a Data Manipulation Language (DML) should also be provided. Very often, both commercial (industrial) DBMSs and their open-source alternatives provide a DML which implements relational algebra – the DML allows the statement of relational algebra expressions as a means of retrieving and updating data. The standard relational DB language is SQL. SQL implements a subset of relational algebra, and is the subject of Chapter 4 of the subject guide.

Relational algebra is used as a measure of the expressive power of a relational language. A database language is said to be **relationally complete** if it is at least as powerful as relational algebra,[8] that is, if any expression from relational algebra can be implemented in the language. Since there are elements of most database languages that cannot be expressed by Codd's relational algebra, it is important not to mistake relational completeness with expressive completeness, nor to regard it as either a minimum or maximum requirement for a satisfactory DBMS.

[8] *For more about this definition, see Codd (1972), which is listed in the 'References cited' section at the head of Chapter 3 of the subject guide.*

As we have seen in the examples above, it may be possible to express a single query in several different ways in relational algebra. From the point of view of a theoretical model such as relational theory, these statements are equivalent, since they will always give the same results from the same input. The situation is different for a database language – such as SQL – that **implements** relational algebra. The data manipulation statements of such a language must be evaluated by the DBMS, and although the end-result should be the same for different, relationally equivalent expressions, they may take different amounts of time to process. Some queries will be faster or more **efficient** than others, depending on the way they are specified and how the DBMS interprets them and optimises its processing of them.

Relational algebra has two characteristics that are relevant in this context:

- relational algebra expressions specify the operations, but not the **order** in which they are to be performed (although there are precedence rules, such as the use of brackets)
- relational algebra operators are **set-at-a-time** – the way they are performed on individual tuples is not specified.

The order in which operations are performed when evaluating an expression and the mechanism for executing operators on individual tuples are implementation issues. They are taken care of by a module of the DBMS called the **optimiser**. The optimiser selects the best evaluation strategy for a given expression.

A consequence of the existence of the optimiser is that users do not have to worry about how to best state the queries, as long as the expressions they devise are correct.

Consider the following example. From the relations used in our examples above, suppose that the university using these relations has 3,000 students in the `Students` relation, while each student takes around four courses, so the `Registrations` relation contains around 12,000 tuples.

Now suppose the course called AI has around 100 students, and that the lecturer needs to know which degree programmes they are all on. The natural language query for that would be "Get the degree programme of all students who take the AI course", which can be expressed in relational algebra as:

```
((Students JOIN Registrations) WHERE Course="AI")[SName]
```

Several evaluation strategies can be taken here, of which two are:

- Perform the join (12,000 tuples to be joined to 3,000, resulting in a new relation with 12,000 tuples and four attributes), then the restriction (searching through 12,000 tuples to find 100 student records) and then the projection.
- Perform the restriction first on `Registration` (find the 100 student records in 12,000 tuples), then the join (100 tuples to be joined to 3,000 resulting in 100 records) and then the projection.

It should be clear that the second of these is far superior. The restriction operation is approximately the same in each case, but the join is about a hundred times smaller and also returns a smaller relation. We shall return to optimisation strategies in Volume 2 of this subject guide, but more discussion is also given in the 'Optimization' Chapter in **Date**.

3.7 Relational data integrity

Databases are systems that implement data models of real-life systems.[9] If data were to be modelled only by specifying the structure of such a system and the operations that can be performed upon it, then an important aspect would be lost. In real-life systems, all kinds of constraints can exist between data values. The data incorporated within a database has to comply with these constraints to be correct; namely, be an accurate representation of reality.

For example, the two relations illustrated in Figure 3.29, Persons and Departments, illustrate some of the possible inaccuracies in data representation. The ID attribute, of the Persons relation was intended to be a unique identifier for a person but, because this constraint (the uniqueness property) was not expressed in any way, it was possible to have a duplicated value in table rows 2 and 3. An invalid name and an invalid date of birth were given in row 2, while row 3 contains an invalid (negative) value for income.

According to Persons, there are two people working in the MMO1 department, but according to Departments, there are three, and while HR has no people associated with it, a non-existent department – HP – has 1. All these examples illustrate the fact that certain configurations of data cannot be valid models of reality and such situations must be somehow described and subsequently avoided.

[9] It is convenient to talk about them in terms of real-life systems, but of course a database can just as easily represent information from fictional sources – such as novels or films – or model speculative or theoretical realities. Even in such cases, we would expect to be modelling something external to the database, and that such a thing, whether real or imaginary, would obey some sort of internal logic.

Persons				
ID	**Name**	**DoB**	**Income**	**Department**
1	M. Jackson	29/8/1958	34,000	MM01
3	Robert') DROP TABLE Persons;[10]	01/04/2105	29,000	MM01
3	F. Mercury	05/09/1946	-45,000	HP

[10] This name is a reference to this cautionary webcomic: http://xkcd.com/327/

Departments		
Department	**Name**	**No_of_employees**
MM01	Manufacturing Management	3
HR	Human Resources	1

Figure 3.29. `Persons` **and** `Departments` **relations, containing nonsensical and incorrect values.**

An informal description of some of the **integrity constraints** for the situation described above could be:

1. No two tuples in `Persons` can have the same value for the ID attribute.
2. Any date of birth (`DoB`) in `Persons` has to be a valid date, be in the past, and date from between 16 and 80 years prior to the date of entry.
3. The values for `Income` should be positive and contain no more than two decimal places (there is also likely to be an upper limit).
4. For each tuple in `Persons`, there must exist exactly one tuple in `Departments` with the same value for the `Department` attribute.
5. The number of people in Persons with a given value for `Department` must be the same as the corresponding value of the `No_of_employees` attribute in `Departments`.

Data integrity denotes the accuracy or correctness of data. In order to devise a correct model of a real-life system, the set of constraints existing between the data values must be identified and specified.

In the context of the relational model, two types of integrity constraints can be identified, namely:

- application or database specific, in that they are applicable only to the application at hand; and
- generic or general, being relevant to the integrity of any model.

The requirement of a positive value for `Income`, the acceptable value range for dates and the correspondence between `no_of_employees` and the `Persons` relation are all database-specific integrity constraints. The relational model does not specifically cater for them, although some can be expressed and enforced using domains. Most database languages do provide some degree of support for expressing these constraints, often using relational algebra expressions. The next chapter will show how this is achieved in SQL.

The fact that the attribute chosen for tuple identification has to be unique (constraint 1 above) and the 'corresponding values' constraint, used for linking relations (constraint 4, above) are aspects of two integrity constraints, relevant to any relational model.

The two general integrity constraints stem from the following rationales.

- An addressing mechanism must exist that provides the unique identification of each tuple within a relation.
- No tuple in a relation that is required to make a reference to another tuple (either in the same or in another relation) should refer to one that does not exist.

They are modelled with two concepts from the relational model, **candidate key** and **foreign key**.

3.7.1 Candidate keys

Candidate key. Given a relation `R`, a subset `CK` of the attributes of `R` represents a candidate key for `R` if, for any extension, it has:

1. The **uniqueness property** – no distinct tuples have the same value for `PK`; and
2. The **irreducibility property** – no proper subset of `PK` has the uniqueness property.

One of the generic integrity constraints can now be defined:

Entity integrity. The entity integrity constraint specifies that each relation must have at least one candidate key.

This constraint is always satisfied within the relational model. Every relation has at least one candidate key – since a relation cannot contain duplicate tuples, the complete set of attributes of a relation is always a possible candidate key. If it has the irreducibility property, then it is the (only) candidate key for the relation. If it is reducible, then it must have a proper subset that is irreducible which, in turn, will be a candidate key.

Consider the relation `StorageBoxes` shown in Figure 3.30. Each model of box has a unique `Model-No`, so this is a candidate key. It is also the case that each model has a distinct size, so the subset {`Height, Width, Depth`} is also a candidate key for `StorageBoxes`.

StorageBoxes				
Model-No	**Height**	**Width**	**Depth**	**Price**

Figure 3.30. The `StorageBoxes` relation has two candidate keys: `Model-No` and `{Height, Width, Depth}`.

If a candidate key consists of only one attribute, it is said to be **simple**. If it contains more than one, it is called **composite**.

Since a given relation can have more than one candidate key, there is a choice of which should be used for tuple addressing. The chosen key is called the **primary key** of the relation, with the others called **alternate keys**.

For a database system, specifying candidate keys allows the uniqueness property of the corresponding attributes to be enforced – the system ensures that no operation that could result in a candidate key having duplicate values is permitted.

3.7.2 Foreign keys

Foreign key. Given two relations, `R1` and `R2`, a subset of the attributes of `R2`, `FK` is a foreign key of `R2`, referencing `R1`, if:

1. `R1` has a candidate key, `CK`, defined on the same domains as `FK`; and
2. Each value in `FK` is equal to the value of `CK` in some tuple in `R1` **at all times**.

Note that the reverse of the second requirement above is not necessary – there may well exist values for the candidate key that are not matched by any value of foreign key.

Students				Registrations	
Name	**Age**	**Degree**	**ID**	**ID**	**Course**
A. Turing	21	CS	AS191	AS191	Prog1
A. Lovelace	28	CS	BC24Z	AS191	CS
F. Allen	22	CIS	AX007	AS191	AI
M. Ibuka	21	IT	NN02M	BC24Z	MusTech
				BC24Z	AI
				AX007	Prog1
				AX007	CS
				AX007	DB
				AX007	AI

Figure 3.31. An example of foreign and candidate keys. Here, `ID` is a candidate key in `Students` and a foreign key in `Registrations`. The only candidate key in `Registrations` is `{ID, Course}` – all the attributes of the relation.

In the example in Figure 3.31, each tuple in `Registrations` has a corresponding tuple – having the same `ID` – in `Students`. For instance:

- {`AS191, CS`} corresponds to {`A. Turing, 21, CS, AS191`}
- {`AS191, AI`} also corresponds to {`A. Turing, 21, CS, AS191`}
- {`AX007, DB`} corresponds to {`F. Allen, 22, CIS, AX007`}

There is no tuple in Registrations that corresponds to {M. Ibuka, 21, IT, NN02M}, presumably because that student is yet to register for a course. There is no requirement in the definition of foreign keys that would make this a problem.

The `Students` relation has ID as its primary key, while {`ID, Course`} is the primary key of `Registrations`. ID is a foreign key in Registrations, **referencing** `Students` – `Students` is the **target**, or the **referenced relation**, whereas `Registrations` is the **referencing relation**. {`AS191, CS`} is a **referencing tuple** in Registrations, its **target**, {`A. Turing, 21, CS, AS191`} is a **referenced tuple** in `Students`.

The fact that a relation `R2` has a foreign key that references a relation `R1` can be represented diagrammatically in the form of a **referential diagram**, as in Figure 3.32.

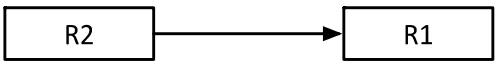


Figure 3.32. A simple referential diagram. R2 references R1.

A referential diagram is actually constructed either for the whole database (the whole set of relations modelling a real-life system) or for a substantial part of it; for instance, for the relations in the figure, we can construct the diagram shown in Figure 3.33.

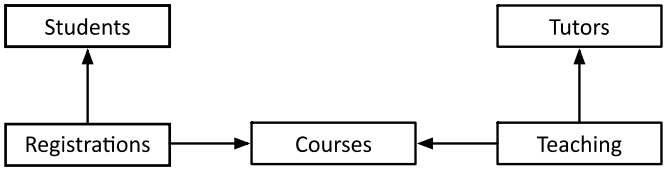


Figure 3.33. A more complex referential diagram.

The other generic integrity constraint, **referential integrity**, can also now be defined.

Referential integrity. The referential integrity constraint specifies that the database – the whole set of relations – must not contain any unmatched foreign key values.

In other words, there always must exist a target tuple for any existing referencing tuple. For a database system, the specification of the foreign keys enforces the referential integrity constraint. That is, the system ensures that the result of any update operation cannot result in unmatched foreign keys.

3.7.3 Nulls

When a relational model is built, it is usually assumed that all the analysed information is available. However, there are situations when some information is unavailable or missing. For instance:

- The address of the customers of a certain chain of shops is confidential; therefore, there may be situations when this information is not provided.
- New students have registered with the university but have not yet decided which modules to take, so this information is not yet available.
- Some antique coins have been found on an architectural site, but the year is unknown.

These instances of missing or unavailable information are handles using **nulls**.

Null. A null is a way of indicating missing information.

Note that a null is not a value; do not confuse it with zero or blank. A null is a marker and means unknown value. A simple example is shown below.

ID	Name	Age	Degree
IZ00B	M. Methuselah	**NULL**	CIS

Some definitions previously given must be revisited in the light of the existence of nulls.

Entity integrity. The entity integrity constraint specifies that each relation must have at least one candidate key. **A candidate key may not accept NULL values.**

Foreign key. Given two relations, `R1` and `R2`, a subset of the attributes of `R2`, `FK` is a foreign key of `R2`, referencing `R1`, if:

1. `R1` has a candidate key, `CK`, defined on the same domains as `FK`; and
2. Each value in `FK` is either null or is equal to the value of `CK` in some tuple in R1 **at all times**.

Referential integrity. The **referential integrity** constraint specifies that the database – the whole set of relations – must not contain unmatched, **non-null** foreign keys.

3.7.4 Domains and normal forms

There are two other mechanisms within the relational model that can be used for the expression of certain kind of constraints.

The first mechanism is through **domains**. Domains can be used to impose limitations on the admissible values of a certain attribute and also on the

admissible operations that can be performed upon it. For instance, in order to express a date constraint for a date of birth, so that DoB has to be of the form dd/mm/yyyy and should be between 01/01/1960 and 01/01/2001, we can define a domain, say DatesOfBirth, and then define the attribute DoB of this type.

As a language for illustration we shall use the simplified SQL language we have used before. For illustration purposes we allow for an interval of integer values and for a record to be specified by:

```
INTERVAL OF INTEGER [<min> .. <max>]
```

and

```
RECORD (<type>/<type>/<type>)
```

respectively, even though these data types are not supported by SQL.

```
CREATE DOMAIN Days    AS INTEGER [1 .. 31]
CREATE DOMAIN Months  AS INTERVAL OF INTEGER [1 .. 12]
CREATE DOMAIN Years   AS INTERVAL OF INTEGER [1960 .. 2001]
CREATE DOMAIN DatesOfBirth AS RECORD (Days/Months/Years)
CREATE BASE RELATION   Persons (
   ID    INTEGER,
   Name  VARCHAR,
   DoB   DatesOfBirth,
   --etc.
)
```

Of course, this definition is not very sophisticated, for example, permitting impossible dates such as 31/2/1960, but the definition could be extended; many DBMSs will have built-in date domains making this easier.

The second mechanism uses the **normal forms**. Three particular kinds of constraint – functional dependencies, multiple dependencies and join dependencies – are expressed by means of normal forms. They will be introduced only briefly here – they are treated in detail in Chapter 5 of the subject guide.

Normalising a relation generally involves decomposing it into a set of relations of smaller degree, in order to eliminate avoidable and undesirable dependencies between its attributes. Such dependencies create redundant data in the original relation, which, in turn, may lead to problems when updating it. Removing the dependencies reduces the risk of those problems.

Consider, for example, a relation `Employees` (Figure 3.35), which contains information about workers and their salaries.

Employees				
ID	**Name**	**YearsInService**	**Role**	**Salary**
D13	W. Ganim	2	Programmer	25,000
D14	A. Fry	3	Programmer	27,000
D25	O. Lai	3	Programmer	27,000
D23	V. Kotlář	5	Programmer	34,000
D04	A. Randall	5	Programmer	34,000
D03	R. Fry	5	Programmer	34,000
D02	M. Singh	8	Programmer	41,000
D01	M. Singh	8	Analyst	43,000

Figure 3.35. The relation `Employees`, showing experience, role and salary.

The attribute `ID` is the unique candidate key (and the primary key). If, in this organisation, the salary of an employee depends entirely on their role and number of years in service, we should be able to predict it from them. Even if we do not have the formula used for that calculation, if more than one staff member has the same role and experience level, there will be repetition of information in the table. For instance, in the table above, three staff members have five years of service and are programmers, and so have the same salary level. This repeated information is called **redundant data**.

In order to express the dependency between `YearsInService`, `Role` and `Salary`, we decompose `Employees` into two relations – `Employees` and `Salaries` – that, combined, are equivalent to the old relation.

Employees			
ID	**Name**	**YearsInService**	**Role**
D13	W. Ganim	2	Programmer
D14	A. Fry	3	Programmer
D25	O. Lai	3	Programmer
D23	V. Kotlář	5	Programmer
D04	A. Randall	5	Programmer
D03	R. Fry	5	Programmer
D02	M. Singh	8	Programmer
D01	M. Singh	8	Analyst

Salaries		
YearsInService	**Role**	**Salary**
2	Programmer	25,000
3	Programmer	27,000
5	Programmer	34,000
8	Programmer	41,000
8	Analyst	43,000

Figure 3.36. A normalisation of the `Employees` relation from Figure 3.35. The candidate key for `Employees` is unchanged, while the candidate key for the `Salaries` relation is `{YearsInService, Role}`.

3.8 Integrity constraint definition and foreign key rules

It is not sufficient for a data definition language (DDL) to support only the definition of the structure of data. It also has to support the definition of integrity constraints on data. Since the two generic integrity constraints are represented by means of keys, the DDL has to support their definition.

SQL supports the definition of keys, so we shall continue to use it for illustrations. Three more notational conventions are needed:

- @ in front of a construct means a list of those elements, separated by commas.
- ::= means 'is by definition'.
- | (vertical bar) signifies exclusive selection (a choice of either…or)

The syntax for data definition in (simplified) SQL, allowing the definition of keys is:

```
CREATE TABLE <relation name> (
   @<attribute definition>,
   <primary key definition>,
   @<candidate key definition>,
   @<foreign key definition>
);
<primary key definition>   ::=  PRIMARY KEY (<set of attributes>)
<candidate key definition> ::=  CANDIDATE KEY (<set of attributes>)
<foreign key definition>   ::=  FOREIGN KEY (<set of attributes>)
           REFERENCES <relation name>
```

For instance, the normalised relations of Figure 3.36 can be defined as follows:

```
CREATE TABLE Employees (
   ID              CHAR(3),
   Name            VARCHAR,
   YearsInService  INTEGER,
   Role            VARCHAR,
   PRIMARY KEY     (ID),
   FOREIGN KEY     (YearsInService, Role)
)
CREATE TABLE Salaries (
   YearsInService  INTEGER,
   Role            VARCHAR,
   Salary          INTEGER,
   PRIMARY KEY     (YearsInService, Role)
)
```

Since relations are linked to one another through keys – in other words, through the values of some of their attributes – a question arises of what to do when one of the linked values changes. How should the link be maintained? More precisely, what happens when the value of the primary key attribute of a target tuple changes – what should happen to the referencing tuples? To answer this, we require **foreign key rules**.

There are two situations to consider:

- the target tuple is going to be deleted; and
- a value in the primary key attribute of the target tuple is going to be modified.

If no other action is taken, then, in both cases, the referencing tuples will be left referring to a tuple that does not exist anymore. Such a situation is not acceptable, and so some extra operations must be performed by the DBMS in order to maintain the consistency of the database.

SQL provides two types of actions that a DBMS can perform automatically in case a target tuple is modified, to **restrict** or to **cascade**. To illustrate the required behaviour of the DBMS, we shall consider four examples.

Consider two relations describing models of a company's products and the components they require.

```
CREATE TABLE Models (
   ModelID      CHAR(8),
   Name         VARCHAR,
   -- other attributes…
   PRIMARY KEY  ModelID
);
CREATE TABLE Components(
   ModelID      CHAR(8),
   ComponentID  CHAR(8),
   -- other attributes…
   PRIMARY KEY  (ModelID, ComponentID),
   FOREIGN KEY  (ID) REFERENCES Employees
);
```

If a model ceases to be produced, the respective tuple from `Models` is deleted. Assuming that this model has components, what should be done with their records? Since the model no longer exists, the component entries are now meaningless and should be deleted from the database. The delete operation on the model has to be **cascaded** onto the referencing tuples.

A slightly different situation might occur for a university that holds information on students in one relation, and on the books currently borrowed from the library in another.

```
CREATE TABLE Students (
   StudentID      CHAR(6),
   Name           VARCHAR,
   -- other attributes
   PRIMARY KEY    (StudentID)
);
CREATE TABLE Borrowings (
   BookID         CHAR(10),
   StudentID      CHAR(6),
   -- other attributes
   PRIMARY KEY    (BookID, StudentID),
   FOREIGN KEY    (StudentID) REFERENCES Students
);
```

When students finish their degree – either by graduating or dropping out – the corresponding tuples must be deleted from `Students`. If records remain in `Borrowings`, it means that the student still has books from the library. If all the records are deleted when the student leaves, then there will be no way of knowing what books are owed.

On the other hand, if the student's record is deleted, then the tuples in `Borrowings` will have a `StudentID` that does not correspond to any tuple in `Students`. Instead, it is better if deleting the student's record is not allowed until all corresponding entries in `Borrowings` have been removed (because the student has returned those books). In this case, we say that deletion of the `Students` tuple has to be **restricted** by the referencing tuples in `Borrowings`.

The third example considers updating information and uses the same `Students` and `Borrowings` relations. Suppose that it is decided to change all `StudentIDs`, perhaps to harmonise with another database.

If a `StudentID` is changed in `Students`, the corresponding entry in `Borrowings` could be left with an invalid value for its foreign key. Clearly, the referencing tuple in `Borrowings` must be updated with the same value. In this case, we say that the update is **cascaded**.

Finally, consider the `Courses` and `Registrations` relations we have met several times before, with `Courses` listing university courses and `Registrations` recording the students who are currently enrolled on them.

```
CREATE TABLE Courses (
   Course      CHAR(8),
   Credits     INTEGER,
   -- other attributes
   PRIMARY KEY  Course
);
CREATE TABLE Registrations(
   StudentID   CHAR(6),
   Course      CHAR(8),
   PRIMARY KEY (StudentID, Course),
   FOREIGN KEY (Course) REFERENCES Courses,
   FOREIGN KEY (StudentID) REFERENCES Students
);
```

Suppose that a course is being revised, with the name and credits being changed, but other fields remain the same. It would be very unusual to change course specifications when there were still students studying on a course, so we should only allow such a change when there are no students currently registered for the course. So, the update in `Courses` must be restricted if there exist any referencing tuples in `Registrations`.

It should be clear from these examples that there are two actions possible when an updating operation – deletion or modification – affects the primary key of a relation that is referred to by another relation:

- for the update of the target tuple to be **cascaded** to the referencing tuples; or
- for the update of the target tuple to be **restricted** if referencing tuples exist.

SQL allows the specification of foreign key rules as follows:

```
CREATE TABLE <relation name> (
   <attribute definition>,
   <primary and candidate keys definition>,
   @<foreign key and foreign rules definition>
);
<foreign key and foreign key rules definition> ::=
      FOREIGN KEY (<set of attributes>) REFERENCES <relation name>
         ON DELETE <option>
         ON UPDATE <option>
<option> ::= CASCADE | RESTRICT
```

For instance, for the first example, the definition of Components becomes:

```
CREATE TABLE Components(
   ModelID      CHAR(8),
   ComponentID  CHAR(8),
   -- other attributes…
   PRIMARY KEY   (ModelID, ComponentID),
   FOREIGN KEY   (ID) REFERENCES Employees
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

The syntax of DDLs in real implementations is usually extended to provide support for the expression of other kinds of integrity constraints. For instance, SQL supports the definition of constraints between attributes of different relations by means of boolean (truth-valued) expressions. Such mechanisms are presented in Chapter 4 of the subject guide.

To return to the concept that started this discussion of database integrity, we can say that a database is considered **correct** if it satisfies the logical **and** all of the integrity rules.

3.9 Conclusions

This chapter presented the main aspects of the relational model and how they are operationalised in a relational DBMS. The relational model is a theory by means of which the information related to a real life application can be modelled. The main advantage of the relational model consists in the synergy between its simplicity and its power of expression. As a result, the relational model was almost universally adopted as the theoretical basis for database systems; although challenges from other models are growing, powered by web technologies and scales, for now, it remains the *de facto* standard data model.

The next chapter presents the most popular language that implements the relational model, SQL. In the main you will learn how to create, query and maintain a database. The last chapter of Volume 1 of the subject guide and the first chapter of Volume 2 will then present and answer the question: how can a successful (that is, fit for purpose) database be developed?

3.10 Overview of the chapter

In this chapter, we described the relational model in detail, starting with basic terminology and concepts, and then looking at how data structures are defined and how data is added, manipulated and retrieved in the model. Finally, we introduced the idea of the integrity of data in a relational system.

3.11 Reminder of learning outcomes – concepts

Having completed this chapter, and the Essential readings and activities, you should be able to:

- describe how a real-life system can be modelled within a data model
- describe the relational model and the way it is used in a relational DBMS; be familiar with the terminology of the relational model
- describe the concept of domains
- describe the concept of relations and discuss the properties of relations
- discuss, in general terms, how the relational data objects are operationalised (used in a relational DBMS)

- describe each of the operators of relational algebra
- be able to express natural language statements, representing information to be inferred from relations, as relational algebra expressions
- explain the way relational algebra is used in the context of DBMSs (including the optimiser)
- present different types of inconsistencies that can exist within a relational model
- define and classify integrity constraints
- define the concepts of candidate, primary, alternate and foreign key
- discuss the issue of null values
- describe how the definition of generic integrity constraints is operationalised, including the foreign key rules.

3.12 Reminder of learning outcomes – key terms

Having completed this chapter, and the Essential readings and activities, you should understand the following terms:

- Atomic or scalar value
- Attribute/field and tuple/record
- Base relation
- Built-in (system-defined) types and user-defined types
- Candidate key
- Cardinality and degree
- Common attribute
- Data Definition Language (DDL)
- Data dictionary
- Data Manipulation Language (DML)
- Data representation
- Degree
- Derived relation
- Domain
- Domain-constrained operations
- Entity integrity
- Foreign key
- Foreign key rules, restrict, cascade
- Integrity constraints
- Key
- Named relation
- Primary and alternate keys
- Query result
- Redundant data
- Referencing/Referenced relation, referencing/referenced tuple
- Referential diagram
- Referential integrity
- Relation (having heading and body)

- Relational Database Management System (RDBMS)
- Relational variable
- Simple and composite candidate key
- Snapshot
- View.

3.13 Test your knowledge and understanding

3.13.1 Sample examination questions

a. Consider the following table.

Head of state	**Spouse**
Benjamin Henry Sheares, President of Singapore	Yeo She Geok Sheares
Margaret of Austria, Governor of the Habsburg Netherlands	Prince John Philbert II
Henry VIII, King of England	Catherine of Aragon Anne Boleyn …

i. Rewrite the table as a relation. [2] [Shortening names to save time is acceptable.]

ii. What is the cardinality of the relation? [1]

iii. What is a candidate key? What are the candidate keys for this relation? For each, say if it is simple or composite. [4]

iv. Comment on whether the Head of state attribute in the relation is truly scalar [2]

v. Which, if any of the following statements are True. [2]

1. Reversing the tuples in the relation makes it invalid.
2. Reversing the tuples in the relation makes a new relation.
3. Adding a tuple that is identical to one already there makes a new relation.
4. Adding a tuple that is identical to one already there makes the relation invalid.

vi. This constraint has been added to the relation. Explain what it does. [7]

```
FOREIGN KEY (Spouse) REFERENCES People
   ON DELETE RESTRICT
   ON UPDATE CASCADE
```

b. 'NULL is relational theory's equivalent of FALSE.'

i. Is the above statement correct? [1]

ii. If it is correct, explain why NULL is used instead of FALSE. If it is incorrect, give a better definition. [2]

iii. Give an example of a tuple that uses NULL correctly. [2]

iv. Which, if any, of the following may accept a NULL value:

1. A candidate key.
2. A foreign key. [2]

Notes

Chapter 4: SQL

4.1 Introduction

This chapter considers the SQL database language standard, defining a large subset of its language. It then goes on to describe the differences between a table in SQL and a relation in relational theory as well as SQL equivalents of the relational operators. A brief explanation is given of views, snapshots and stored procedures.

4.1.1 Aims of the chapter

The aims of this chapter are to give you an introduction to SQL, explain Data Definition Language (DDL), summarise data integrity constraints, before concluding with views and stored procedures.

4.1.2 Learning outcomes

By the end of this chapter, and having completed the Essential readings and activities, you should be able to express a complex database system in both standard SQL and in the dialect of at least one DBMS. More specifically, you should be able to:

- define data objects at the conceptual level
- define domains
- list and describe the main predefined data types of SQL
- define integrity constraints
- define views at the external level
- implement natural language queries
- alter data contained in the database
- describe stored procedures, in general terms.

4.1.3 Essential reading

- Date, C.J. *An introduction to database systems*. (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Chapters 4 and 9 (1999 edition: Chapters 4 and 8).

and/or

- Connolly, T. and C.E. Begg *Database systems: a practical approach to design implementation and management*. (Pearson Education, 2014) 6th edition, global edition [ISBN 9781292061184 (pbk)], Chapters 6–8 (1999 edition: Chapters 13 and 14).
- The SQL Manual.

4.1.4 Further reading

- Date, Appendix B (1999 edition, Appendix A).
- Ramakrishnan, R. and J. Gehrke *Database management systems*. (McGraw-Hill, 2002) 3rd edition [ISBN 9780072465631 (hbk); 9780071231510 (pbk)], Chapter 5.
- Documentation for DBMS of your choice (see Chapter 1: Introduction to the subject guide for some suggestions and web addresses).

4.1.5 References cited

- Connolly (1999), p.736.

4.2 Introduction to SQL

The most popular database language is, by far, SQL (Structured Query Language). Most major commercial and open-source relational DBMSs implement it in one form or another, so it is important for you to know its main features. This will necessarily represent only a subset of the standard – at the time this subject guide went to print, the documentation for the most recent version of SQL – SQL:2011 – was over 1,400 pages long. This is complicated by the way in which different DBMSs will implement different subsets of different versions of SQL. In this chapter, we can only give you a glimpse of what SQL is. Should you be interested in finding out more, the best reference is usually the Manual for the SQL dialect you will be using for implementing the examples and activities.

You are strongly recommended to ensure you have access to a relational DBMS for this course that can be easily run on your own computer. Of the free and open-source DBMSs that implement reasonable amounts of SQL, the most popular (at the time of writing) are MySQL, PostgreSQL and SQLite. Any of these will suffice for this course and we shall refer to the dialects these use over the course of this chapter.

SQL was developed in the 1970s by IBM, but it was not officially a standard until it was adopted by the American National Standards Institute (ANSI) in 1986 and the International Organization for Standardization (ISO) a year later. Since, it has had several revisions, with early revisions usually designated with a hyphen, for instance SQL-86 and SQL-92, and more recently with a colon, such as SQL:2011. The name is officially pronounced as the letter names spoken separately, but is also commonly pronounced 'sequel'.

SQL is a relational language in that it is founded on the relational model – more specifically on relational algebra. However, it is not a pure implementation of it; some features of the relational model were left aside, whereas some features not belonging to the relational model were added.

SQL is not a specific implementation, but is instead an abstract specification of a relational database language. Specific DBMSs provide their own implementation of SQL, called dialects, which comply with the standard to a varying degree. Ideally, there should be complete compliance with the standard, but, this is not the case in practice.

Vendors have different views upon the importance of different aspects of SQL. Accordingly, they only implement the 'important' aspects, leaving aside the other ones. Also, the syntax of the language may present slight variations from the standard.

Vendors may also try to promote their own implementation with 'extra' (non standard) features. Therefore, even though implemented in SQL, databases developed in different dialects of SQL may not be straightforwardly portable from one system to another. There is a clear disadvantage to a vendor whose users find it easy to move to a different platform, so there is little incentive to ensure full compliance.

Standard SQL has two main components: the data definition (DDL) and data manipulation (DML) component.

The DDL supports:

- definitions at the conceptual level
    - **relations** are well supported (**base relations**, in fact);
    - support for domains is weaker;
- definitions at the external level;

    - views – essentially relations defined on other relations, their extension need not be explicitly stored in the database;
- definitions at the internal level;
    - indexes – these will not be considered until Volume 2 of this subject guide.

The DML is used for database query and maintenance (data insertion, deletion and modification). The DML implements relational algebra, in that relational algebra expressions can be used for data manipulation. SQL also provides support for integrity constraints definition.

SQL is a fourth-generation language. One of its big advantages over third-generation languages – which makes it so powerful for data management – is its ability to manipulate sets: its operators are set level, or set at a time. Moreover, because it is meant to be quite high level, no explicit support for the flow of control is provided in the main operators – there are no IF-THEN statements or loops in these expressions, meaning it 'lacks computational completeness' (Connolly, 1999, p.736).

This means that SQL has to allow for some of its statements to be embedded in programs written in third-generation languages. That is, SQL provides primitives that can be accessed from inside such programs, so the standard SQL provides embedded SQL. If no complex manipulation of data is needed, all that is required being data definition, query and updating, SQL can be used interactively.

In the previous chapter, we drew a distinction between a relation and the less restricted concept of a table. SQL uses tables, rows and columns to implement the relational model elements of relation, tuple and attribute, respectively. Two important differences that arise from this are worth mentioning again: SQL allows duplicate tuples, and the rows and columns of a table are ordered.

Constants are called literals and they can be numeric or non-numeric. All non-numeric values have to be specified within quotes. SQL commands, and table and column names are case insensitive, but, for clarity, it is conventional to write keywords in upper case and user-defined words in upper or mixed case letters. Multiple space, tab and line break characters are treated as a single space, so you can use indentation to improve readability. Comments are also useful to make your SQL easier to read. A comment in SQL starts with two dashes (– –) and continues to the end of the line.

In presenting the syntax of SQL we shall employ a BNF (Backus Naur Form) notation which uses the following conventions:

- all SQL keywords are written in UPPERCASE letters
- syntax elements (or place holders), are enclosed within < and >
- optional parts of a statement are enclosed within `[` and `]`
- alternative options are represented as `{option1 | option2 | …}`
- the symbol `::=` means 'is defined by' or 'is'.

SQL makes extensive use of a particular syntactical structure – a list of one or more elements of the same kind, with members separated by commas. This is represented in BNF as:

```
<comma-list-of-elements> ::=
  {<element> | <element>, <comma-list-of-elements>}
```

To make the presentation of the syntax more concise, we shall also use the following convention:

- a list of one or more elements, separated by commas is represented as `@<element>` where:

```
@<element> ::= {<element> | <element>, @<element>}
```

For illustration purposes, the presentation of each issue is accompanied by examples. You are advised to try to devise your own set of examples to augment these. This is the best method of learning a new language: it will help you to better understand and to memorise.

It is not sufficient to devise examples with pen and paper – you must experiment with whatever SQL dialects and DBMSs are available to you.

- Use generic SQL to guide your experimentation.
- Use these two subject guide volumes and the documentation to identify differences from the standard.
- Adapt the examples for use in your DBMS.
- Extend the examples and **explore further**. It is difficult to become truly fluent in a language without experimentation.

4.3 The Data Definition Language (DDL)

The DDL component should support the creation and definition of the relational data object – domains and relations. For a long time, SQL domains were restricted to being, in essence, new names for built-in-types, allowing the specification of constraints, but greatly limiting their intended use in relational theory. SQL-99 was the first to permit user-definition of types and, for simpler instances, the use of more complex built-in types to be available to domain definitions. Partly as a result of this, support for domains in DBMSs varies. MySQL, for example, has no support for domains at the time this subject guide goes to press, while PostgreSQL fully supports domains and user-defined types.

SQL's DDL goes beyond relational theory by defining a greater variety of integrity constraints. It also provides support for practical aspects of defining a database not covered by the theoretical model – security rules, for instance – that will not be covered here, but will be considered in the second volume of this subject guide.

4.3.1 Domains

Domains in SQL are defined in terms of data types, and most uses of them are likely to be based on the built-in data types, also known as **basic, primitive, system-defined** or **predefined**. Figure 4.1 lists the basic data types in SQL.

Type	Keywords (for declarations)
Character (string)	CHAR, VARCHAR, CLOB
Binary	BINARY, VARBINARY, BLOB
Exact numeric	NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT
Approximate numeric	FLOAT, REAL, DOUBLE PRECISION
Date-time	DATE, TIME, TIMESTAMP
Interval	INTERVAL
Boolean	BOOLEAN
XML	XML

Figure 4.1. The predefined data types of SQL. Boolean types were added in 1999, XML in 2006. Bit types were removed in 2003.

Beyond these predefined data types, constructed types are permitted in domain declarations. These are generally compound forms, made up of multiple instances of basic types. The permitted constructed types are ARRAY, MULTISET, REF and ROW. The precise distinction between types in modern SQL and domains in SQL and relational theory will not be further explored here.

In this subject guide, we only introduce the most commonly encountered data types. For a broader introduction, we suggest referring to one of the course texts (although be warned that this is one of the cases where the older your copy of these books the more relevant information will be missing).

Character types

Data of the type **character** is represented by a string, such as "Data of the type character is represented by a string". The character set to be used for the string can be defined, and includes the entirety of Unicode. There are three character data types, depending on the length of the strings to be stored and how they are to be stored.

`CHAR` (optionally `CHARACTER`) is a string with a maximum length specified after it, such as `CHAR(10)`, which can be up to 10 characters long. All values placed in a `CHAR` field will be allocated the same amount of storage space, so 'short' and '-longest-' will both take the same amount of space.

VARCHAR (optionally `CHARACTER VARYING`) is also a string with a maximum length, but unlike `CHAR`, `VARCHAR` does not pre-allocate space. Instead, the size of the string is stored with the string. As a result, strings that are much shorter than the limit take up much less storage space, but those that are as long as is allowed will use more space than they would represented as `CHAR`. This means that, unless all the strings in a column are close to the maximum length, less storage space is likely to be used.

`CLOB` (optionally `CHARACTER LARGE OBJECT`) is for use with strings that are longer than the largest string acceptable to `CHAR` and `VARCHAR` (which, in turn, is implementation dependent). It behaves in general like a `VARCHAR`.

Be warned, there is no requirement for a database to alert the user to an attempt to insert a string longer than the maximum declared for that column. Instead, the entry will be truncated (shortened) so that it fits. For example, if a column has been declared as `CHAR(10)`, then setting a value to "`this string is too long`" will result in "`this string`" being stored instead.

The selection of data type also limits the scalar operators that can be applied to its data. Figure 4.2 shows some of the SQL operators that can be applied to character data types.

Implementations. MySQL and PostgreSQL both provide CHAR and VARCHAR, but call their CLOB-like data type TEXT. SQLite maps all character data types into a single TEXT type that is effectively a VARCHAR.

Operator	Description	Example
CHAR_LENGTH(<string>) →<INTEGER>	Returns the number of characters in the string	CHAR_LENGTH("EXAMPLE") →7
<string> || <string> →<string>	Concatenates two strings	"EXAM" || "PLE" →"EXAMPLE"
LOWER(<string>) →<string>	Converts to lower case	LOWER("EXAMPLE") → "example"
TRIM([LEADING|TRAILING|BOTH] [<string>] FROM <string>) →<string>	Remove characters from the beginning and or end of string	TRIM(BOTH "E" FROM "EXAMPLE") → "XAMPL"

POSITION(<string> IN <string>) →<integer>	Returns the position of one string within another	POSITION("AMP" IN "EXAMPLE") →3
SUBSTRING(<string> FROM <integer> [FOR <integer>] [USING <integer]) →<string>	Returns part of string with start point and length of result specified	SUBSTRING("EXAMPLE" FROM 2 FOR 3) →"XAM"

Figure 4.2. Some operators and functions that work with character data types.

Implementations. Most of these functions are universally provided in implementations, although the syntax may vary. For example, for the SUBSTRING function, the USING part is dropped in all cases, while MySQL makes the keywords FROM and FOR optional, so that SUBSTRING("EXAMPLE", 2, 3) has the same effect. A more significant example is TRIM in SQLite, which is replaced with three functions, LTRIM (the equivalent of the LEADING keyword), RTRIM (equivalent of TRAILING) and TRIM (=BOTH), each takes two arguments that are the string to be trimmed and, optionally, the string to remove. The default, as for the standard version of the function, is to remove spaces.

The concatenation operator varies widely, and should probably be avoided. Microsoft SQL SERVER uses + instead, Oracle support varies, and in MySQL, CONCAT("EXAM","PLE") must be used. Even in systems where it is available, you should be careful using it – it can have unexpected effects unless both arguments are strings.

Exact numeric types

These data types represent numbers in an exact way. Integers are a special case of exact numbers, having no decimal places, and can be represented more economically if they are treated differently. The three integer datatypes are `SMALLINT`, `INTEGER (INT)` or `BIGINT`. The standard does not specify the size limits of these types, but only that the maximum value that can be stored in a `SMALLINT` should be smaller than the maximum stored in an `INT`, which in turn is smaller than for `BIGINT`.

Implementations. In most current implementations, a `SMALLINT` uses 2 bytes (-32,768 – 32,767), INT 4 bytes (up to 2.1 billion) and BIGINT 8 bytes (up to 9.2×10^{18}). Oracle and MySQL allow any of these to be declared as `UNSIGNED`, which limits their values to positive integers and so doubles the maximum value they can store.

Allowing fractional values to be exact requires, in SQL, a guarantee of a certain **precision** and **scale**. By precision, we mean the total number of significant (relevant) digits, whereas by scale, we mean the number of decimal places. For instance, 765.4321 has a precision of 7 and a scale of 4, while -0.012 has a precision of 4 and a scale of 3. Since these attributes are specified based on the decimal system, how the resulting data should be stored as binary data is not as directly obvious. For this reason, SQL provides two very similar data types, NUMERIC, which has a precision exactly as specified, and `DECIMAL`, which has a precision at least as good as is specified, but optionally better. The syntax for specifying the two is the same:

```
NUMERIC [ ( <precision> [, <scale>] ) ]
DECIMAL [ ( <precision> [, <scale>] ) ]
```

Implementations. In almost all implementations, these data types are synonyms.

The most useful numerical operators are likely to be familiar already – arithmetical addition +, subtraction -, multiplication * and division /. The comparison operators are also the same as they are in mathematics and most programming languages: <, >, =, <=, >=. They return a boolean value.

Values provided for a numeric data type that are more precise than the type specification will be rounded to fit, while values that are too large (or too negative) will be reduced to the limit permitted by the data type.

Other types and type conversion

Most of the other types in SQL behave as you would expect them to in general-purpose programming languages. You should be familiar with approximate numeric types such as the FLOAT from other experience (if not, you are strongly encouraged to look them up). `DATE`s and `TIME`s are supported, along with `TIMESTAMP`, which holds a date and time. Just as strings can record their character set, times and timestamps can record the time zone to which they apply. INTERVALs are also provided for the difference between two temporal values (so that you can talk about 45 minutes rather than just clock times such as 12:45).

Boolean types are used for true and false values, although the special nature of relational theory means that we have a three-value logic of `TRUE`, `FALSE` and `UNKNOWN`, with the last represented by a null indicator.

Converting between types is possible using the CAST operator. Its syntax is:

```
CAST (<values> AS <data type>)
```

So that:

```
CAST (12.3 AS INTEGER)
→ 12
```

Domains and types

Domains in relational theory have a rather complicated relationship with domains and types in modern SQL. The CREATE DOMAIN command allows the definition of a domain and its constraints, allowing the full range of basic types, along with more complex types and user-defined forms. Implementations vary widely in the extent to which these are available and, where they are, the extent to which the constraints are enforced. The syntax is as follows:

```
CREATE DOMAIN <domain name> [AS] <data type>
             [DEFAULT <default value>]
             [@<domain constraint>]
             [<collate clause>];
<domain constraint> ::= CONSTRAINT <name>
  CHECK <integrity constraint> <constraint characteristics>
```

Note: <data type> is restricted to predefined data types. The default value specified must be a constant, a built-in function with no arguments or null. The ability to use a function for the default allows the current user (`USER`) or date (`CURRRENT_DATE`) to be recorded at the time the field is modified.

The collate clause relates to how to handle character data types, leaving the domain constraint, which is specified using a boolean expression of any complexity as `<integrity constraint>`. The constraint characteristics govern when the constraint is to be checked or enforced.

A simple example might be a domain for a working week, where we want to constrain the allowable number of hours per week and specify a default:

```
CREATE DOMAIN Hours_Per_Week SMALLINT
     DEFAULT 40
     CONSTRAINT Legal_Hours VALUE > 5 AND Value < 48;
```

A domain can be destroyed using the `DROP DOMAIN` statement, or its definition can be changed, including adding new integrity constraints using `ALTER DOMAIN`.

Much of the logic of the relational model's domains is now part of SQL's data type syntax, while other parts, such as the integrity constraints are available through other means. In general, it is best to check the particular DBMS you are using before deciding whether and how to use domains in your own databases.

4.3.2 Base relations

Create a table

Relations are implemented as tables in SQL, and constructed using the `CREATE TABLE` statement. The table can be created from scratch, by reference to other data or tables, or based on a user-defined template or **table type**. For creating the table from scratch, the syntax is:

```
CREATE TABLE <table name>
   (@<column definition>,
     <primary key definition>,
     [@<candidate key definition>,]
     [@<foreign key definition>,]
     [@<table integrity constraint definition>]
   );

<column definition> ::=
   <column name> {<data type> | <domain>}
   [{DEFAULT <default value> | <generator>}]
   [NOT NULL] [UNIQUE]
   [@<check constraint >]
   [COLLATE <collation name>]

<check constraint> ::= CHECK ( <expression> )

<primary key definition> ::= PRIMARY KEY (@<column name>,)

<candidate key definition> ::= UNIQUE (@<column name>,)

<foreign key definition> ::=
   FOREIGN KEY (@<column name>,)
   REFERENCES <base table> [(@<column name>,)]
   [ON UPDATE <option>]
   [ON DELETE <option>]

<option> ::= {CASCADE | SET NULL | SET DEFAULT | NO ACTION}
```

```
<table integrity constraint definition ::=
   [CONSTRAINT <constraint name>] <check constraint>
```

There are a few points to clarify what may look like quite a complicated syntax:

- Columns:
  - A table needs at least one column.
  - A column must have a name and a type.
  - A default value may be specified for a column.
  - Optionally, a default value can be generated based on the expression `<generator>`; to avoid complicating matters further, we will not give the syntax of this or discuss it further here.
  - The `NOT NULL` keyword prevents a column from having a value set to `NULL`.
  - The `UNIQUE` keyword prevents duplicate values for the column.
  - Check constraints evaluate an expression of any complexity and, provided that it returns a boolean value, use the results to validate new values.
- Exactly one primary key must be specified – values in the columns specified must be unique for each entry in the table.
- Candidate keys may be specified using the UNIQUE keyword.
- Foreign keys:
  - Specifying foreign keys is optional, and the effect of doing so varies between DBMSs.
  - Specifying `ON UPDATE` and `ON DELETE` rules is optional.
  - The `CASCADE` option behaves as described in the Chapter 3 of the subject guide.
  - The `NO ACTION` option behaves as `RESTRICT` in Chapter 3 of the subject guide.
  - `SET NULL` and `SET DEFAULT` allow the value to be reset to `NULL` or default when the relevant update or delete has occurred.
- Table integrity constraints are tested before permitting changes to the table – as with column constraints, they can be expressions of any complexity, as long as they return a boolean value.
- There are times when a table is only needed as a short-term place for storing data, in which case a temporary table may be used. We will not consider these in any detail, but the syntax is very similar to normal syntax:

```
CREATE {GLOBAL | LOCAL} TEMPORARY TABLE <table name>
   (@<column definition>,
      <primary key definition>,
      [@<candidate key definition>,]
      [@<foreign key definition>,]
      [@<table integrity constraint definition>]
   ) [ON COMMIT {PRESERVE | DELETE} ROWS]
```

Creating a table based on an existing table is also quite similar, with the column definition clause replaced with three components:

1. An optional list of column names for the new table.
2. The query that generates the relation/table.

3. A keyword to indicate whether the query is to provide just the column definitions or the data as well.

The syntax for this clause looks like this:

```
<subquery clause> ::=
   [ (@<column name>,) ] AS <table subquery> WITH [NO] DATA
```

If column names are not given, they are taken from the column names in the result of the query.

We will revisit this form of `CREATE TABLE` once the syntax for querying tables has been introduced.

To illustrate some of these points, we shall expand on an example of `CREATE TABLE` introduced in the previous chapter:

```
CREATE TABLE Borrowings (
   BookID          CHAR(10),
   StudentID       CHAR(6),
   DueDate         DATE NOT NULL,
   RenewCount      INT DEFAULT 0 CHECK (RenewCount < 20),
   PRIMARY KEY     (BookID, StudentID),
   FOREIGN KEY     (StudentID) REFERENCES Students
           ON UPDATE CASCADE
           ON DELETE NO ACTION,
   FOREIGN KEY     (BookID) REFERENCES Books
           ON UPDATE CASCADE
           ON DELETE NO ACTION
);
```

This example shows a table recording a book borrowed by a student (as with most of our examples, this is not necessarily how one would structure this in a complete, working system). Column constraints are used here to check that the book always has a due date and to ensure that it is not renewed more than 20 times.

Activity

Referring to the discussion of foreign key rules in the previous chapter, explain why `ON UPDATE` and `ON DELETE` are set as they are in this example.

Implementation. Support for `CREATE TABLE` is good in recent versions of most DBMSs. MySQL offers several storage engines for tables, and we strongly recommend you set it up to use InnoDB by default. If you have a MySQL database that uses the older MyISAM storage format, constraints will be ignored. In PostgreSQL, `RESTRICT` and `NO ACTION` are not quite synonyms, but they are close enough for most purposes to consider them the same.

Alter a table

The definition of a table can also be altered at any point after it has been defined. The following changes can be made:

- adding or removing columns
- adding or removing constraints
- adding or removing a default value for a column.

These are all carried out using the `ALTER TABLE` command, so, to add a column, the syntax is:

```
ALTER TABLE <table name> ADD [COLUMN] <column definition>
```

Where the [COLUMN] is just optional syntax to make the statement easier to read, and column definition is the same as for CREATE TABLE.

Removing a column is also simple:

```
ALTER TABLE <table name> DROP [COLUMN] <column name>
  [{RESTRICT | CASCADE}]
```

The RESTRICT or CASCADE keywords here define the behaviour of the system if there are foreign keys or other references to the column to be dropped. In such cases, removing the column could affect the referential integrity of the database, and appropriate measures should be taken to avoid leaving the database in an inconsistent state.

- RESTRICT specifies that if such references exist, the ALTER TABLE command should not be executed.
- CASCADE specifies that all referring columns should be dropped as well as the one being explicitly removed here.

Adding or removing table constraints can work in a similar way:

```
ALTER TABLE <table name> ADD [CONSTRAINT <constraint name>]
   <constraint definition>
ALTER TABLE <table name> DROP [CONSTRAINT] <constraint name>
   [{RESTRICT | CASCADE}]
```

Note: <constraint definition> here includes table integrity constraints, but also key specification, including the primary key. These should be changed with caution.

You can also change the default for any column:

```
ALTER TABLE <table name> ALTER [COLUMN] <column name>
   {SET DEFAULT { <default value> | <generator> }
      |DROP DEFAULT }
```

There are some other uses of ALTER TABLE in the standard, but their use is rare enough that they can be ignored for now.

Activity

Try to write out the syntax for all the forms of ALTER TABLE in a single set of statements, in a similar way to the CREATE TABLE syntax description above. Why are so many clarifying keywords, like CONSTRAINT and COLUMN optional? What does that tell you about the rules for the uniqueness of names in SQL?

Destroy a table

A table is destroyed using the DROP statement:

```
DROP TABLE <table name> [{CASCADE | RESTRICT}];
```

If CASCADE is specified, then all data objects that refer to the table (and all objects that refer to those, and so on…) will be destroyed as well. If RESTRICT is specified, then if there are any references to the table, then the statement will not be allowed to be executed.

So far, you have learned how to define, change and delete relational data objects in SQL. Just as the main data objects in relational theory are called base relations, the result of a CREATE TABLE command is called a **base table**. A base table has the following properties:

- Its definition is stated explicitly, and it is not dependant on the definition of other tables.

- It is physically stored in the database.

You should now be able to define the schema of a database in SQL. The next section describes how data can be manipulated, using the DML.

Activity

Either using an example from this subject guide or making up your own example, try to model a simple system in SQL, for instance a library or a shop. What tables do you need? What columns? Write CREATE TABLE statements for each of the tables you have designed. Do these statements work directly in your choice of DBMS, or do they need to be modified?

Save the database that you make in this activity – you can use it for further activities later in this chapter.

The data manipulation language consists of two components. They define statements for retrieving data and statements for updating data, respectively. They make use of relational algebra statements.

4.3.3 Retrieval

A database stores data. This data needs to be accessed by different applications or users for different purposes. It is essential to be able to retrieve data from the database.

Although queries were originally considered as part of a DML, they do not themselves change the data content of the database. For this reason, the SQL standard no longer classifies read-only queries as part of the DML. Since this change doesn't affect the syntax, and requires the `SELECT` statement to appear in two different categories, we have chosen to preserve the older classification here. Correctly, however, a `SELECT` statement without an `INTO` clause is an 'SQL-data' statement rather than a DML one.

Although queries were originally considered as part of a DML, they do not themselves change the data content of the database. For this reason, the 2003 edition of the SQL standard removed read-only queries from the DML. Since this change does not affect the syntax, and splits the `SELECT` statement to appear in two different categories, we have chosen to preserve the older classification here. Correctly, however, a `SELECT` statement without an `INTO` clause is an 'SQL-data' statement rather than a DML one. In your outside reading, you will find that some books and online resources have embraced the change, while others have not.

Query. A statement that specifies what data is to be retrieved (by the DBMS) from the database is called a **query**.

In SQL, all queries are specified using a `SELECT` statement. The syntax for the core of a `SELECT` statement looks like this:

```
SELECT        [DISTINCT] {* || @<column name>,}
   FROM        <table reference>
   [WHERE      <where condition>]
   [GROUP BY   @<column name>,]
   [HAVING     <having condition>]
   [ORDER BY   @<column name>,];
```

`SELECT` is a powerful and versatile statement, and has many aspects that are not reflected in the syntax we have given here. `SELECT` statements can, as we shall see, use the output of other `SELECT` statements as part of their

queries, and their outputs can be combined. The ANSI standard also allows for sampling a subset of very large tables, but this is a feature that is currently only rarely implemented.

Rather than attempting to explain the full syntax and power of the `SELECT` statement at once, we shall build towards it, by a series of examples, starting simply and getting increasingly complicated. For these examples, we use tables derived from the previous relational algebra examples. The tables, fields and foreign key relationships are shown below, but you will also find the tables spelt out and with sample data in an Appendix to Volume 2 of this subject guide and on the VLE, where there are also files for uploading into a DBMS to experiment with.

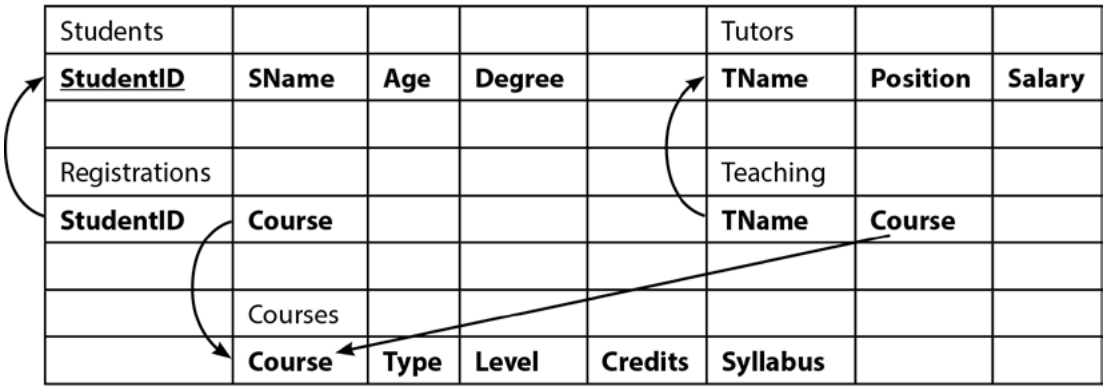


Figure 4.3: The set of relations for the examples in this section.

Relational algebra operators

Three specific operations can be implemented straightforwardly:

- **Restriction**: The relational algebra expression

  ```
  <relation name> WHERE <restriction>
  ```

  is implemented as

  ```
  SELECT * FROM <table name> WHERE <condition>
  ```

  Here, * is an instruction to return all the columns from the selected table. The statement that selects all Level 1 courses is:

  ```
  SELECT * FROM Courses WHERE Level=1;
  ```

- **Projection**: The relational algebra expression

  ```
  <relation name> [<list of attributes>]
  ```

  is implemented as

  ```
  SELECT @<column name>, FROM <table name>;
  ```

  For instance, to see the name and salary of all tutors, the statement is:

  ```
  SELECT TName, Salary FROM Tutors;
  ```

- **Join**: We have encountered two types of join in relational algebra. The natural join, which combines relations only where the attributes they have in common match:

  ```
  <relation name> JOIN <relation name>
  ```

  This is implemented in SQL as an explicit NATURAL JOIN:

  ```
  SELECT * FROM <table name> NATURAL JOIN <table name>;
  ```

So, to see which students have registered on which courses, we can use:

```
SELECT * FROM STUDENTS NATURAL JOIN Registrations;
```

The Θ-join, which takes the Cartesian product of relations and filters it according to a conditional expression can easily be implemented, but only once we have met the SQL's Cartesian product syntax.

- **Set-based operators**: can be used to combine the results of two selects, so to get all modules taught by professors or which give one course credit (the fourth example in the relational algebra tasks of the previous chapter), the syntax would be:

```
SELECT Course FROM Tutors NATURAL JOIN Teaching
     WHERE Position="Professor"
  UNION
     SELECT Course FROM Courses
        WHERE Credits=1;
```

That is an unnecessarily complicated way of answering that question, and in practice set operations can usually be replaced by much simpler relational ones:

```
SELECT Course FROM Tutors NATURAL JOIN Teaching NATURAL
JOIN Courses
   WHERE Position="Professor" OR Credits=1;
```

- **Cartesian product**: The Cartesian product is the combination of each row in a table (with *m* rows) with all the rows in another table (with *n* rows) to produce a very large combined table (with *mn* rows). In SQL, it is referred to as a CROSS JOIN and it is the default form of join.

```
SELECT DISTINCT * FROM <table name> [CROSS] JOIN <table
name>
```

The `DISTINCT` keyword here ensures that there are duplicate rows (i.e. that the result is a legal relation). This sort of join is particularly useful as part of a Θ-join; that is, when it has a restriction placed on it by a `WHERE` clause. A natural join can be constructed in this way:

```
SELECT   * FROM Students JOIN Registrations
   WHERE  Students.StudentID=Registrations.StudentID;
```

There are several things to note about this syntax. Firstly, since there are two columns in the combined relation with the same name (`StudentID`), they have to be distinguished by naming the table that they come from as well as the column; for example, `Students.StudentID`. This also means that tables can easily be joined even when the matching columns have different names, or when some sort of transformation must be carried out on one of them before matching can happen.

Eliminating duplicate rows

Unlike a relation, a table can have duplicate rows (if it has no primary key). If we wish to avoid duplicates appearing in the results of a query, then this must be explicitly stated using the `DISTINCT` keyword, so if we want to know the ages of students, but not how many are of each age nor how old each student is, the query is simple:

```
SELECT DISTINCT Age
   FROM Students;
```

Note that there is no equivalent for this in relational theory, since any operation necessarily returns only unique tuples.

Sorting results

In relational theory, there is no explicit ordering of tuples, but SQL tables are ordered and, as a result, they can be sorted. A SQL table can be sorted by fields in ascending order (with the keyword ASC) and descending order (with the keyword DESC). Sorting depends on the data type of the affected field, and the sorting order for text – especially accents – will be affected by the text's **collation**, which is usually a reflection of the language.

To sort tutors by their salary, starting with the highest paid, the statement is:

```
SELECT      *
   FROM      Tutors
   ORDER BY  Salary DESC;
```

To sort students by age (youngest first), and then, for students with the same age, to sort them by name:

```
SELECT      *
   FROM      Students
   ORDER BY  Age ASC, SName ASC;
```

What is important to remember here is that the first column in the ORDER BY clause is used first, then the second, and so on. If there are several entries with the same value for all the columns in a sort clause, their final order depends on the implementation. Usually, they will remain in the order in which they appear in the parent table.

The columns for ORDER BY need not appear in the resulting table, as in this example:

```
SELECT     SName
   FROM    Students
   ORDER BY Age ASC;
```

Some conditional expressions

Since null values are different to a boolean false, they cannot be found using conventional logical tests. Instead, there is a special syntax for `IS NULL` and `NOT NULL`:

```
SELECT   Course
   FROM  Courses
   WHERE Syllabus IS NULL;
```

Sometimes, you may want to test whether a value is one of several options. One way to do this is to write each option separately, thus:

```
SELECT    *
   FROM    Students
   WHERE   Degree='Maths' OR Degree='Biology'
      OR Degree='Chemistry' OR Degree='Physics';
```

Slightly neater is to use SQL's set notation and the IN operator:

```
SELECT     *
   FROM  Students
   WHERE Degree IN {'Maths', 'Biology', 'Chemistry', 'Physics'};
```

This way of doing things becomes neater for larger sets, but can also be used with the results of a sub-query, so that the 'set' here can be, for example, a column from another table.

As will be clear by now, conditional expressions can be combined in the same way that they can in most algebraic and programming language, using the operators `AND`, `OR` and `NOT` and using brackets to group them together.

Calculated fields

The columns in the results of a query need not be direct copies of columns in a source table. They can also be calculated or transformed, or they can be simple values or system variables. For instance,

```
SELECT    6 AS Option_no,
     CURRENT_DATE AS Generated_Date,
     Salary+10000 AS Proposed_Salary
   FROM Tutors;
```

The `AS` keyword allows a column to be assigned a name, and is particularly useful either where columns do not come directly from tables, and so have no obvious default name, or where there are multiple columns with similar or the same name, when using `AS` can help clarify which is which. Since natural joins are based on column names, sometimes renaming columns is essential for future joins to work.

Aggregate functions

Aggregate functions summarise information from several rows. This is something that is useful in many applications such as calculating average coursework marks, finding the fastest car in a list, or calculating the total cost of a shopping list. Such functions are commonly seen in spreadsheet applications. SQL provides a set of functions, called aggregate functions, for the computation of such summary information. They include a variety of set-theory functions and statistical calculations, but the most general, and commonly-used, are:

`COUNT`	Counts the number of values
`SUM`	Adds numeric values
`AVG`	Computes the mean (simple average) of numeric values
`MIN`	Returns the lowest value
`MAX`	Returns the highest value.

For example, if we wanted to know the number of students, their average age and the age of the oldest student in the Students table of previous examples, the query would be:

```
SELECT   COUNT(*) AS Number_of_students,
     AVG(Age) AS Average_age,
     MAX(Age) AS Oldest
FROM   Students;
```

`COUNT`(*) indicates that all triples should be counted, and the standard provides a syntax for filtering the results so that only some triples are counted. The simplest way to reduce the items counted is to use `DISTINCT` to limit the results to only the different values. For instance, to find out how many different degree programmes students are registered on, the query could be:

```
SELECT   COUNT(DISTINCT Degree) AS Number_of_Programmes
FROM     Students;
```

Often, rather than wanting to summarise the data for a whole table, we want to break it down and summarise different categories or subgroups in the data.

To do that, we need to group rows and then perform the aggregate functions on these groups. These operations are implemented using the `GROUP BY` clause. For example, to find the number of students enrolled on each degree programme, we can perform the following query:

```
SELECT   Degree,
     COUNT(*) AS Number_enrolled
FROM   Students
GROUP BY Degree;
```

Here, the `GROUP BY` Degree clause causes the table to be partitioned into groups of students with the same value for their Degree field. `COUNT` is then applied to each group. The result is a table consisting of one row for each degree programme.

An important restriction applies – **each item in the select column list has to have a single value for the whole group**. If that were not the case, then the result would not be a simple table, and some files would have to contain multiple values, at odds with relational theory. That restriction can only be guaranteed in a few cases:

- if the item in the column list is a field explicitly named in the `GROUP BY` clause
- if the item in the column list is calculated from one or more fields named in the `GROUP BY` clause (and no other fields)
- if the item is a constant or a value calculated from elsewhere (such as the current date)
- if the item uses the columns in an aggregate function (which will summarise multiple values as a single result).

In our example, Degree is the only column from Students that may appear in the column list. This means that the following would be incorrect:

```
SELECT    Degree, Age,
     COUNT(*) AS Number_enrolled
FROM      Students
GROUP BY  Degree;
```

If our aim with this query was to see for each degree programme how many students there were of each age, the correct query would be:

```
SELECT    Degree, Age,
     COUNT(*) AS Number_enrolled
FROM      Students
GROUP BY  Degree, Age;
```

Activity

Using aggregate functions, how would you find out the average salary of tutors? How would you find the average age of students enrolled on each course? (Hint: you will need a join for the second of these.)

Filtering groups

We have seen how the `WHERE` clause can be used to restrict queries, allowing rows to be left out of the results when they fail to satisfy a set of criteria. Conditions described in the `WHERE` clause are applied to **rows**, not groups of rows, so they cannot be used with aggregate functions. Instead, SQL provides a similar clause that applies to groups instead – `HAVING`.

For example, if we were interested in the popularity of degree programmes that attracted older students, we could try the following query:

```
SELECT    Degree,
     COUNT(*) AS Number_Enrolled,
FROM      Students
GROUP BY Degree
HAVING  AVG(Age) > 25;
```

Here, as in our earlier example, only the degree programme and the number of students is displayed, but only degrees with an average student age above 25 are shown. Note that, as with the `WHERE` clause, there is no need for the fields or functions that appear in the clause to be mentioned explicitly elsewhere, as long as the fields used appear in the source tables.

It can sometimes be confusing to decide whether a restriction should appear in a `WHERE` or a `HAVING` clause. In general, all restrictions based on aggregate functions will use `HAVING`, while operations on simple fields, even if they will subsequently be aggregated, will use `WHERE`. For example, if, for each degree programme, we wanted to find the average age of students who were older than the usual UK undergraduate age, the query might be:

```
SELECT   Degree,
     AVG(Age) AS Average_Age
FROM    Students
WHERE   Age > 21
GROUP BY Degree;
```

Although it is often stated that the difference between `WHERE` and `HAVING` is simply between one clause operating on rows and one on groups of rows, the true distinction is a little simpler. `WHERE` restrictions are applied early in processing to reduce the amount of data that has to be manipulated afterwards. That means that conditions specified in a `WHERE` clause are considered before the grouping operations in `GROUP BY` have been carried out – so it would be impossible to evaluate aggregate functions at that point. `HAVING`, on the other hand, is applied later and so has access to grouped data. This distinction is important because, in practice, there are other sorts of calculated information that can only be filtered using a `HAVING` clause.

Subqueries

One of the great strengths of the `SELECT` statement is the ability to use other `SELECT` statements inside the main query. This is something that SQL inherits from relational theory, and slightly extends.

For example, to find students whose age is above the current average, we need two queries – the first to calculate the average, and the second to find the students with a higher age. Instead of taking these queries separately, we can use the one within the other directly:

```
SELECT    *
FROM      Students
WHERE     Age > (SELECT AVG(Age)
          FROM   Students);
```

This is a case where relational theory has been extended – we would expect the result of a query to be a table, not a numerical value, so by relational theory, the > test should be invalid. In SQL, however, a table consisting of one column and one row can be treated as a scalar value.

We saw earlier how the `IN` operator allows us to check that a value belongs to a provided set of possibilities. This operator can be combined with a subquery that generates the set of candidates.

For example, to list all students enrolled on Level 1 courses, we could write:

```
SELECT   SName, Course
FROM     Students
   NATURAL JOIN Registrations
WHERE    Course IN (SELECT Course
        FROM Courses
        WHERE Level = 1);
```

This example is only for illustration – it would have been simpler to have a single `SELECT` with an extra join clause:

```
SELECT   SName, Course
FROM     Students
     NATURAL JOIN Registrations
     NATURAL JOIN Courses
WHERE   Level = 1;
```

`SELECT`s can be nested to any depth, so a subquery in an `IN` clause can itself contain a subquery. Such constructions are sometimes necessary, but are not used often, and can make queries harder to read.

More multi-table queries

We have already seen some of SQL's join behaviour in the section above on relational operations. In the simplest form of the syntax, tables can be joined in SQL just by listing them in the FROM clause:

```
SELECT Tutors.TName, Salary,
   Teaching.TName, Course
FROM   Tutors,
   Teaching;
```

This join results in the Cartesian product – every row in the first table is combined with every row in the second.

If there were two tutors in the `Tutors` table – Mark Taylor and Ani Rai – and two courses in the `Teaching` table (with each tutor teaching one course), then the result of the query might look like this:

Tutors.TName	Salary	Teaching.TName	Course
Mark Taylor	27823	Mark Taylor	Programming1
Mark Taylor	27823	Ani Rai	Databases1
Ani Rai	32010	Mark Taylor	Programming1
Ani Rai	32010	Ani Rai	Databases1

This sort of join, also called a cross join, can quickly result in large, meaningless results unless it is constrained. A more intuitive query might be:

```
SELECT Tutors.TName, Salary,
     Teaching.TName, Course
FROM   Tutors,
     Teaching
WHERE  Tutors.TName=Teaching.TName;
```

This would return the more interesting result:

Tutors.TName	Salary	Teaching.TName	Course
Mark Taylor	27823	Mark Taylor	Programming1
Ani Rai	32010	Ani Rai	Databases1

The nature of the join can be made explicit using the `JOIN` operator:

```
SELECT TName, Salary, Course
FROM   Tutors JOIN Teaching USING (TName);
```

Not only is this notation clearer about what is going on, but it also removes the duplicated `TName` column.

This sort of join is called an inner join. An even more succinct way of specifying the inner join described above is to use the `NATURAL JOIN` operator. This matches all the columns in the tables that have the same name:

```
SELECT TName, Salary, Course
FROM   Tutors NATURAL JOIN Teaching;
```

Since the syntax is a little clearer and more intuitive than the alternatives, we use it freely in this subject guide. However, do remember that this is quite an easy operator to get wrong. Since the query does not name the fields to use for the join explicitly, they can easily (and silently) fail to match expectations. This is particularly an issue:

- if tables use columns with generic names such as `Description` or `ID` – these can occur in multiple tables without implying a foreign key.
- if there is more than one reason for associating two tables, there may be a genuine foreign key relationship, but not the intended one. For instance, if each course has a moderator – a tutor who doesn't teach on the course but evaluates it, they may have their `TName` associated with the table as well. Since no explicit relationships are given with `NATURAL JOIN` there is no way to specify this.

It is important to remember with inner joins that if a field that is acting as a foreign key in the join has a value of `NULL`, then the row affected will never be joined, since `NULL=NULL` evaluates to `false`.

Sometimes rather than looking for matching rows in another table, we are looking for rows that have no matches. For example, to find which tutors have not been assigned to teach any courses, we could use `IN` with a subquery:

```
SELECT TName, Salary
FROM   Tutors
WHERE  TName NOT IN (SELECT TName
        FROM   Teaching);
```

Another option would be to use another type of join – the **outer join**. Outer joins also use a join criterion, but do not exclude any row from the tables – even if there is no matching row. Instead, the relevant values are simply returned as `NULL`. There are asymmetric outer joins that only preserve rows from the first or the second tables, and these are specified using the `LEFT` or `RIGHT` keyword. For example:

```
SELECT TName, Salary, Course
FROM   Tutors
   LEFT JOIN Teaching USING (TName);
```

Suppose that a new tutor has joined – Selene Maughan – and she has yet to be assigned a class. In that case, the query above would result in something like this:

TName	Salary	Course
Mark Taylor	27823	Programming1
Ani Rai	32010	Databases1
Selene Maughan	22000	NULL

So, to find tutors with no teaching assignments, the full query could be:

```
SELECT TName, Salary, Course
FROM   Tutors
   LEFT JOIN Teaching USING (TName)
WHERE  Course IS NULL;
```

Since subqueries return tables, any table in the `FROM` clause can be replaced with a `SELECT` expression.

```
SELECT SName, Age
FROM   Students
   LEFT JOIN (SELECT   Degree, AVG(Age) Degree_average_age
         FROM   Students
         GROUP BY Degree)
   AS Degree_Ages
   USING (Degree)
WHERE  Age > Degree_average_age;
```

Activity

What does this query do? Can you think of any other ways of getting the same result?

Checking existence

It can be useful, especially when defining constraints, to check whether a particular query returns any results for each of a set of values. A simple example within a normal query would be to find which students are registered on a Level 3 course using the following query:

```
SELECT SName
FROM Students INNER JOIN Registrations USING (StudentID)
WHERE EXISTS (SELECT *
      FROM  Courses
      WHERE Courses.Course = Registrations.Course
         AND Level=3);
```

Here, the `EXISTS` predicate returns a value equivalent to true if the subquery yields any results and false if it has no results.

Activity

How many other queries can you write that get the same result?

Combining results

Although it is possible to combine tables and logical restrictions within `SELECT` statements themselves, it is also sometimes useful to use set operators to combine query results. The three available operators are `UNION`, to combine all results, `INTERSECTION`, to include only the rows that are common to the different results, and `EXCEPT` for the difference between the results (all the rows in the union excluding those in the intersection). In all cases, as in relational

theory, the results tables used have to be type compatible – they need to have comparable columns.

So, for example, to find all students who are more than 25 years old or are studying Maths, we could use the following query:

```
( SELECT SName
   FROM   Students
   WHERE  Age>25 )
UNION
( SELECT SName
   FROM   Students
   WHERE   Degree='Maths' );
```

Clearly, for simple examples such as this, combining the constraints in the `WHERE` clause is a cleaner option:

```
SELECT SName
FROM   Students
WHERE  Age > 25 OR Degree='Maths';
```

These set operators become more obviously useful when the queries draw their information from different tables, meaning that a single query would require many joins and subqueries to produce the same effect. The examples here are too simple to show this clearly.

These are just some of the more useful aspects of querying in SQL. You are strongly encouraged to explore on your own, trying out all of these categories in at least one SQL-based DBMS. Since no `SELECT` query can write to the database (except the `SELECT…INTO` syntax), you can try out anything you want, without any risk to the data stored. In the next section, however, we shall turn to operations that do alter the contents of the database. These operations can rewrite or remove data, and should not be used on a production database without first understanding what they do.

4.3.4 Updates

There are three basic operations in updating a database:

- insertion
- deletion
- modification.

Insertion

An insertion operation inserts rows into a table. The data for these new rows can be supplied by the command itself or can come from elsewhere in the database.

The syntax for the first type of insert – using user-specified values is:

```
INSERT INTO <table name> [(@<column name>)]
VALUES      @(@<value>);
```

This command creates a new row for each of the sets of values in brackets (`@values`). If a `<column name>` list is not provided, it is assumed that all the columns in the table are to be supplied, in order. Some examples:

```
INSERT INTO Students
VALUES  (14021, "Vincente Tomi", 29, "Politics"),
        (14022, "Yuan Tseh Lee", 21 "Chemistry");
```

```
INSERT INTO Students (Sname, Age, Degree)
VALUES    ("Enayat Khan", 28, "Music");
```

If a column does not appear in the column name list, then the field is given a NULL value for inserted rows unless the column has been defined with a default. In general, ID column values are likely to be generated by a simple counter, so omitting the field as in the previous example would result in an appropriate value being added.

The second type of insertion, using data already in the database, has the following syntax:

```
INSERT INTO <table name> [(@<column name>)]
     <compatible select statement>;
```

To be compatible, the SELECT statement must return a table with the same columns that would appear in the VALUES clause of the other form of INSERT.

```
INSERT INTO Registrations
   (SELECT StudentID, Course
   FROM   Students CROSS JOIN Courses
   WHERE  Degree="Computer Science"
        AND Level=1);
```

Activity

What does this statement do?

Deletion

A deletion removes rows from a table, according to certain conditions.

```
DELETE FROM <table name>
[WHERE      <condition>]
```

The WHERE clause here works just as it does in a single-table SELECT statement, and a good way to check which rows will be deleted is by running a query with the same WHERE clause before executing the DELETE statement. As in SELECT statements, if the WHERE statement is omitted, then all rows are included – the table is emptied.

```
DELETE FROM Students
WHERE       Age<18;
```

Although the statement itself must refer to only a single table, subqueries are still legal. For example, to remove students who are not registered on any course, the following query may be used:

```
DELETE FROM Students
WHERE       StudentID NOT IN (SELECT DISTINCT StudentID
             FROM   Registrations);
```

Modification

Existing values in the database are modified by means of an UPDATE statement. Its syntax is:

```
UPDATE  <table name>
SET     @{<column name>=<value>}
[WHERE  <condition>];
```

The WHERE clause can be used to narrow down which rows are to be updated. If no clause is specified, the statement is applied to the whole table.

Within the set clause, each named column can be set to a constant or a calculated value. For example:

```
UPDATE  Students
SET     Age=Age+1, Degree="Computing and Information Science"
WHERE   Degree="Computer Science";
```

Activity

Return to your exercises from the previous chapter. Can you write all the relational expressions as SQL statements? Can you think of any queries that you cannot express? Why?

4.4 Integrity constraints

The following types of integrity constraints were introduced together with the statements for data definition (`CREATE DOMAIN` and `CREATE TABLE`).

- **Domain integrity constraints**. These are constraints that affect all values of any column defined on the declared domain. For example, here we declare `Hours_Per_Week` as a domain with values limited to being between 10 and 48; we then use that domain in the `Working_hours` column of an `Employees` table:

```
CREATE DOMAIN  Hours_Per_Week  SMALLINT
       DEFAULT       40
       CHECK       Value > 10 AND Value < 48;
CREATE TABLE    Employees (
       Working_hours   Hours_Per_Week;
       ...
);
```

- **Column integrity constraints**. These behave like the domain constraints, but are specified directly for a column rather than a domain. This makes sense if the restriction will only happen once in a database. If it will happen more than once, a domain constraint is more economical. Using a column constraint, the table in the above example could have been specified as follows:

```
CREATE TABLE    Employees (
       Working_hours   SMALLINT
             CHECK    Working_hours > 10
                AND Working_hours < 48
       ...
);
```

- **Base table integrity constraints**. These specify constraints on the attributes of a table, allowing a constraint based on more than one column. For example, if the weekly pay of an employee is to be limited, the table definition might look like this:

```
CREATE TABLE    Employees (
       Working_hours  SMALLINT
             CHECK    Working_hours > 10
                    AND  Working_hours < 48,
       Hourly_rate DECIMAL(3,1)
             CHECK    Hourly_rate > 8.5,
       ...
```

```
        CONSTRAINT  Maximum_pay
             CHECK       Working_hours
                  * Hourly_rate < 1000
);
```

Whenever an update operation (`UPDATE` or `INSERT`) is attempted, the DBMS checks the relevant integrity constraint. If one or more of them is violated, then the update operation is rejected. If all the constraints are satisfied, then the update is performed.

- **Database integrity constraints**. These are imposed on the database as a whole. As a result of this, the constraint can call on data in any table or combination of tables. The syntax for creating or removing a database integrity constraint is as follows:

  ```
  CREATE ASSERTION <constraint name> CHECK (<boolean
  expression>);
  DROP ASSERTION <constraint name>;
  ```

  The expression can use aggregate functions, for numerically summarising table information, or existentially-qualified algebra expressions, for asserting the existence or absence of a given relationship.

  For example, if every student must be registered for at least one course:

  ```
  CREATE ASSERTION Minimum_course_registration CHECK (
        NOT EXISTS (SELECT *
           FROM    Students
           WHERE   StudentID NOT IN
                     (SELECT StudentID
                     FROM   Registrations))
  );
  ```

  If we wanted to ensure that all students were registered for at least four courses, one way to write the assertion would be:

  ```
  CREATE ASSERTION Minimum_course_registration CHECK (
        (SELECT MIN(*)
        FROM   (SELECT   COUNT(Course)
           FROM    Students
              LEFT JOIN Registrations
                 USING (StudentID)
           GROUP BY StudentID))
        >= 4);
  ```

Here, we are only interested in the student signed up for the lowest number of courses. If that student has registered for at least four courses, then the criterion is fulfilled. As is often the case with database constraints, there are many other ways to write the same assertion.

Usually, if you were to express an integrity constraint in natural language, phrases such as 'all x must satisfy the condition that', 'some x must exist where y is true' would be used.

Although we have introduced these constraint types as if their terms of reference are limited – the column constraint to its column, the table constraint to the table and so on – in fact, since the constraint in all cases is allowed to be any standard boolean expression, the definition can take in the whole database at any time, whether a database constraint or a column constraint.

4.5 Views

4.5.1 Introduction

We have seen how SQL's equivalent of relations – tables – can be defined by means of a DDL, resulting in structures that are physically stored within the database. However, as we have already discussed, the ANSI/SPARC architecture of database systems requires that different views of a database can be presented to different users. That is, the users need to see different parts, possibly overlapping, of the same database. In some cases, such functionality could be implemented by splitting the database, but this solution is not viable for all cases – for example, two users might need access to different parts of the same table. Since it is important that data is only explicitly stored once in the database, there is need for a different sort of structure, and that is the role of the SQL view. Views represent the way users see the stored data.

View. A view is a named relational expression.

Views can be thought of as virtual relations, because:

- they define a relation (the result of a relational algebra expression is a relation)
- they need not be physically stored (and so are virtual).

In both relational theory and SQL, it should be clear how views work. Any reference to a relation or a table can be replaced with an expression that returns a relation or table – this is how subqueries usually work. The view mechanism is based on a substitution procedure – when a statement comprising a name of a view is executed, the name is replaced with the view's definition.

Since this substitution procedure can be applied recursively, views can be defined on other views.

The syntax for creating and destroying a view in SQL is:

```
CREATE VIEW <name>
AS     <expression>
      [WITH CHECK OPTION];

DROP VIEW   <name> <behaviour>;
<behaviour>::= RESTRICT|CASCADE
```

The two parts of the definition are the <name>, which is needed to be able to refer to the view, and the `<expression>`, which is just a `SELECT` query. `WITH CHECK OPTION` places a restriction on the ability to update tables using the view, something that we shall return to later.

4.5.2 Retrieving data using views

For retrieving data, a view behaves in exactly the same way as a table with the same structure and content.

Employees				
Employee_ID	Name	YearOfBirth	Role_ID	Hours_per_week
Roles				
Role_ID	Role_name	Hourly_pay		

Figure 4.3. Employee and Department tables.

Consider the tables `Employees` and `Departments` shown in Figure 4.3. Suppose that we wished to allow a particular set of users to view information about the employees and their weekly pay. The view definition might be:

```
CREATE VIEW Employees_weekly
AS      SELECT Employee_ID,
             Hours_per_week * Hourly_pay AS Weekly_pay
          FROM   Employees JOIN Roles USING (Role_ID);
```

Now `Employees_weekly` can be used as if it were a base relation, with retrieval being simply a matter of:

```
SELECT * FROM Employees_weekly;
```

4.5.3 Advantages of using views

This approach gives many advantages as follows.

Logical data independence

Views can improve the logical data independence of a database. Applications that use views have no need to model the data structures that the views are using, which means that underlying tables can be changed without changing the behaviour of the applications.

Complete logical data independence can never be achieved – changes in the structure of data in a database, or even its presence or absence will always have the potential to affect client applications to some extent. Nonetheless, views provided an extra layer of abstract structure. As long as a view can generate the same information before and after a change – as long as information is not lost – logical data independence can be preserved.

Suppose it was intended to revise the database in Figure 4.3. Specifically, a new `Employees` table would be created with more and clearer information:

Employees2					
Employee_ID	First_names	Last_name	Date_of_Birth	Role_ID	Hours_per_week

Figure 4.4. Revised Employees table, with the name field divided and a date rather than year of birth.

Any application that used our `Employees_weekly` view would be unaffected as long as the definition were adjusted to point to `Employees2` rather than the older table, but what of applications that accessed the `Employees` table directly?

```
CREATE VIEW Employees
AS     SELECT Employee_ID,
           First_name || " " || Last_name Name,
           EXTRACT(YEAR FROM Date_of_Birth) YearOfBirth,
           Role_ID,
           Hours_per_week
        FROM   Employees2;
```

Since it is quite easy to recreate all the fields of the old table from the new, more detailed table, we can create a view that to all external appearances behaves like the original table. The view `Employees` is now equivalent to the original table `Employees`.

Implementation. Readers who try to reproduce this example in their own DBMS may find errors from the use of the standard SQL operator, ||, which concatenates (joins) strings. In MySQL, there is a setting to enable support for the operator, and otherwise the function CONCAT must be used. In MS SQL, the + operator is a direct replacement and should be substituted for the double pipe.

Different formats of data

Views represent a way of customising a database, because they allow users to see data in a way that best fits their own requirements. For example, some users might want to use a person's date of birth, whereas for others, the current age might be simpler.

Security

The view mechanism is, implicitly, a security feature, since users see only data that is relevant to them, with other information invisible to them, and the actual structure of the database concealed. This issue is discussed in greater detail in Chapter 2 of Volume 2 of this subject guide.

Programming shorthand (macro definition)

A view can be seen as a shorthand for a complex expression and so functions like a macro, simplifying database programming activity.

While views may look externally as if they are identical to a table, there are some very important differences. We will see below how those differences may be seen in practice, but there is also an important question of performance. Since a view is only a virtual table, it is regenerated each time it is requested. Even a simple-seeming view could easily be performing multiple joins, and running any number of operations on the data before it is presented.

Storing views

A view can be expected to run only marginally more slowly than the `SELECT` expression it replaces (the slight difference being due to the small extra step of rewriting the view as a `SELECT`). This is almost inevitably slower than reading from a stored table, even before optimisations such as indexing are taken into account. SQL provides for a form of stored view that is regenerated whenever its parent tables are modified. The standard calls this view a **snapshot**, but you may also see it referred to as a **materialised view**. These views provide all the advantages of a conventional view, but allow retrieval of data to be much faster.

This speed and convenience comes at a cost. Since the view is written to storage at the point when it is created, it records the state of the data at a particular time. If data in the tables is based on changes, the view may become out of date. If the view is recreated every time this happens, the speed savings may be lost. Strategies for reducing this cost are available in DBMSs that implement this feature, but the problem remains, and the situations where snapshots confer the biggest advantage are generally those where the database is read very many more times than its data is altered.

Activity

Investigate whether your chosen DBMS supports snapshots or materialised views. At the time of writing, MySQL does not, and nor does MS SQL, although it does provide ways of indexing views. Explore the syntax and try the examples from this section, creating snapshots rather than views. Try to put enough data into the tables that the performance differences begin to be noticeable.

4.5.4 Updating data using views

We have described views as being 'equivalent' to tables, but their virtual nature means that this equivalence is not complete. While a view may look like a table for the purpose of querying, modifying the data that the view embodies is more complicated. To explain this, we will first define the problem more formally, and then give an example of the problem.

Adopting a functional notation, we can express a view as an operation on a database:

```
View = f(DB)
```

Updating a view then becomes:

```
Update(View) = Update(f(DB))
```

When we update a table, the problem ends there. The table stores the data, and so the Update operation can be executed. For views, however, f(DB) is not stored anywhere – it does not physically exist and may depend on various values distributed around the database. The only way we can perform Update is if we can deduce a function Update' that can be applied to the stored parts of the database, such that:

```
f(Update'(DB)) = Update(f(DB))
```

In practice, a DBMS will implement a set of rules that allow an Update' function to be deduced from a given Update operation. Such a function does not exist for all updates of all views, as we shall see in the examples below, and so the DBMS will place restrictions on the sorts of views and updates that will result in success. The SQL standard itself has also specified such restrictions at various times. Interested users should read the relevant section in Date (Chapter 'Views', section 'View Updates', although if your edition predates 2008, some aspects will be out of date). There is considerable variation between DBMS implementations on this issue.

The problem of interpreting an update can be seen quite clearly with the Employees view introduced above. Let us say that an employee, Lucy Keynes, has recently married and taken her husband's surname, Mendoza, keeping her old surname as a middle name. She will now be known as Lucy Keynes Mendoza. If we wanted to change the table `Employees2` directly, the update operation might be:

```
UPDATE    Employees2
SET   First_names="Lucy Keynes"
     AND Last_name="Mendoza"
WHERE    First_names="Lucy"
     AND Last_name="Keynes";[1]
```

[1] *This version only works if the employee's name is unique. If the* `Employee_ID` *is known, then using this in the* `WHERE` *clause would be safer.*

The Employees view, though, only has Name, a field created by joining the first and last names from the source table. The obvious update command would look like this:

```
UPDATE   Employees
SET  Name="Lucy Keynes Mendoza"
WHERE    Name="Lucy Keynes";
```

If `Employees` were a table, then this command would work without difficulty, but in this case, Name has been generated from two fields. How should the DBMS decide which part of the name string should be considered part of `First_names` and which part should be `Last_name`? There is simply not enough information in the system to make this possible.

By contrast, if Lucy were to request more hours per week, then the command:

```
UPDATE   Employees
SET      Hours_per_week=40
WHERE    Name="Lucy Keynes";
```

is easy to interpret – `Hours_per_week` is a field in the table that the view refers to, and has been presented directly. In most implementations, such an update would work.

4.6 Stored procedures

The core of SQL is constructed around relational theory and the operators that it requires. You should be familiar, from other courses, with other ways of processing information beyond those that can be easily made using `SELECT`, `UPDATE` and `INSERT`. In practice, most applications that use a database will have client software that connects to the database, retrieves data and then processes it using a language that supports iteration and recursion, more advanced flow control and other aspects of modern procedural programming.

The division of work between the database server and a client application can be clean and efficient. For example, it may mean that heavy processing can be distributed amongst multiple connecting machines rather than overloading the server. On the other hand, the server/client separation may mean that more data must pass through the network, and more separate queries are sent to the DBMS than is necessary.

Stored procedures present an alternative approach, providing the advantages of non-trivial, procedural programming within the database server itself. This allows the DBMS software to pre-compile and optimise the code, and for both the amount of information transmitted across the network and the number of client/server interactions to be minimised.

Stored procedure languages – their syntax, implementation details and advantages and disadvantages – are implementation-specific, although a lot is in common between implementation. A full exploration of this topic is out of the scope of this course, but a good database engineer should know when and how to use them. We would strongly encourage you to read documentation and online tutorials for your chosen implementation. Of the free DBMS implementations, PostgreSQL is particularly strong in this area, and has an implementation that is close enough to Oracle's PL/SQL that converting between the two is relatively easy.

4.7 Conclusion

SQL is a large and complex standard, and the many implementations add further complexity. We have only touched on the most basic aspects of it in the course of this chapter. Nonetheless, this core should be enough to allow you to develop quite complicated databases. Through practice and through further reading you should be able to identify the areas that you want or need to explore in greater detail, and of course you will need to understand how what we have introduced here is expressed in your chosen DBMS.

Exercise is by far the most effective way of learning the materials in this chapter, and will help you to turn the theoretical grounding you have received here into a practical skill and the basis for advanced and useful knowledge. If you have any doubts about any of the concepts introduced here, explore them, changing the examples that we give or trying your own or others from the internet or from textbooks.

4.8 Overview of the chapter

In this chapter, the SQL standard was introduced, and a large subset of its language was defined. The differences between a table in SQL and a relation in relational theory were explained, and SQL equivalents of the relational operators (and more) were described. We also briefly explained views and snapshots and the use of stored procedures.

4.9 Reminder of learning outcomes – concepts

Having completed this chapter, and the Essential readings and activities, you should be able to express a complex database system in both standard SQL and in the dialect of at least one DBMS. More specifically, you should be able to:

- define data objects at the conceptual level
- define domains
- list and describe the main predefined data types of SQL
- define integrity constraints
- define views at the external level
- implement natural language queries
- alter data contained in the database
- describe stored procedures, in general terms.

4.10 Reminder of learning outcomes – key terms

Having completed this chapter, and the Essential readings and activities, you should understand the following terms:

- Domain and column integrity constraints
- Inner and outer join
- Query
- Snapshot/materialised view.

4.11 Test your knowledge and understanding

4.11.1 Sample examination questions

a. Consider the following two tables:

```
CREATE TABLE Album (
   AlbumID         INTEGER,
   Name            VARCHAR(120),
   Price           DECIMAL,
   PRIMARY KEY     (AlbumID)
);
CREATE TABLE Song (
   SongID            INTEGER,
   Name              VARCHAR(120),
   DurationInSeconds INTEGER,
   AlbumID           INTEGER,
   PRIMARY KEY       (SongID),
   FOREIGN KEY       (AlbumID) REFERENCES Album
);
```

i. What does VARCHAR(120) mean here? [2]

ii. How do these tables relate? Describe which parts of the SQL above specify this. [3]

In order to get a list of song and album information together, I ran the following query:

```
SELECT * FROM SONG NATURAL JOIN ALBUM;
```

iii. Explain why my query gave the wrong results. [2]

iv. Write a better query or suggest a change to the tables that would make it work. [2]

v. How would I sort the songs in order of duration (shortest first)? [2]

b. 'View is just another word for query.'

i. Is the statement above true? If so, why are both words used? If it is not true, what is the distinction? [3]

ii. Why might a view be useful? [2]

c. SQL provides three set-theory operators.

i. Name them and describe or draw what each does. [6]

```
SELECT * FROM Balloons WHERE Shape="Round" or
Colour="Red"
```

ii. Rewrite the above query to use one of the set operators. [1]

iii. Which version of this query is better? Why? [2]

Chapter 5: Designing relational database systems

5.1 Introduction

This chapter explores the process of designing a database system, given a real-life system to be modelled. Beginning with the Entity/Relationship (E/R) model, this is adapted to ensure compatibility with the relational one, before creating a relational model. Normalisation of the resulting structure is then shown in order to reduce anomalies.

5.1.1 Aims of the chapter

The aims of this chapter are to introduce you to relational database systems, before moving on to conceptual modelling of an E/R model, and the transformation of this model into a relational one. Finally, the chapter concludes with normalisation.

5.1.2 Learning outcomes

By the end of this chapter, and having completed the Essential readings and activities, you should be able to carry out the logical design of a relational database system. More specifically, this means that you should be able to:

- Devise an E/R model for a real-life system:
  - Identify:
    - entities of the system and their corresponding type
    - attributes for each entity and their corresponding type
    - relationships between entities in the model
    - constraints on the relationships in the model
    - type hierarchies between entities.
  - Eliminate flaws of the E/R model.
  - Draw the model in an E/R diagram.
- Transform an E/R model into a relational model.
  - Eliminate the E/R structures that are incompatible with the relational model.
  - Transform the new E/R model into a relational model.
- Normalise a relational model:
  - Identify:
    - possible update anomalies of a relation
    - functional dependencies of a relation
    - multiple dependencies of a relation
    - join dependencies of a relation.
  - Check whether a relation is in a given normal form.
  - Perform a non-loss decomposition of a relation into a set of relations of a higher normal form.

5.1.3 Essential reading

- Date, C.J. *An introduction to database systems.* (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Chapter 14 and then Chapters 11–13 (1999 edition: Chapter 13 and then Chapters 10–12).

and/or

- Connolly, T. and C.E. Begg *Database systems: a practical approach to design implementation and management*. (Pearson Education, 2014) 6th edition, global edition [ISBN 9781292061184 (pbk)], Chapters 12–15 (1999 edition: Chapters 5–8).

5.1.4 Further reading

- Chen, P. 'The Entity-Relationship Model – Toward a Unified View of Data', *ACM Transactions on Database Systems*, 1(1) (1976), pp.9–36.
- Connolly, T. and C.E. Begg *Database systems: a practical approach to design implementation and management*. (Pearson Education, 2014) 6th edition, global edition [ISBN 9781292061184 (pbk)], Chapters 10 and 11 (1999 edition: Chapter 4).
- Date, C.J. *An introduction to database systems.* (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Appendix A.
- Elmasri, R. and S.B. Navathe *Fundamentals of database systems*. (Pearson, 2016) 7th edition [ISBN 9781292097619 (pbk)], Chapters 3 and 4.
- Ramakrishnan, R. and J. Gehrke *Database management systems*. (McGraw-Hill, 2002) 3rd edition [ISBN 9780072465631 (hbk); 9780071231510 (pbk)], Chapters 2 and 19.
- Zaniolo, C. 'A New Normal Form for the Design of Relational Database Schemata', *ACM Transactions on Database Systems*, Volume 7(3) 1982, pp.489–99.

For more on the development process:

- Stevens, P. and R. Pooley *Using UML: Software engineering with objects and components*. (Addison-Wesley, 2006) 2nd edition [ISBN 9780321269676] , Chapter 4.

5.2 Introduction to relational database systems

Developing a database system is a complex process and requires a lot of knowledge and skills. Depending on the size and complexity of the application, it may involve a single developer or hundreds of people. Although the roles and number of the people involved will, of course, depend on the nature of the application, there are some elements that are common to all development processes.

The field of software engineering proposes many models for how to structure the activity of software development, and there are important differences between those approaches that attempt to do a perfect job in one go and the various models in which an initial prototype is successively improved. Nonetheless, these models tend to have many phases of development in common – the difference being largely about when the phases take place, how often they are carried out, and where there are opportunities to repeat phases or groups of phases.

The main activities in the development process usually include:

- **Analysis**, in which the real-life system to be modelled or supported is analysed, resulting in a statement or specification of the requirements. The whole phase can also be called requirements specification.
- **Design**, where the verbal specification from the analysis phase is turned into an implementable model. Formally, this is achieved using semantic and formal or semi-formal models.
- **Implementation**, in which the implementable model is realised in a practical form.

- **Testing**, in which validation and verification are carried out to ensure that the implementation matches the model and the real-life requirements.
- **Maintenance**, which includes activity to correct faults and bugs, and to improve efficiency, after the product has been deployed.

Figure 5.1 illustrates a development process for a database system. The phases in a development process can progress strictly in order, with each phase occurring once, following a pattern called the Waterfall model, or loops can be inserted, allowing for more sophisticated models.

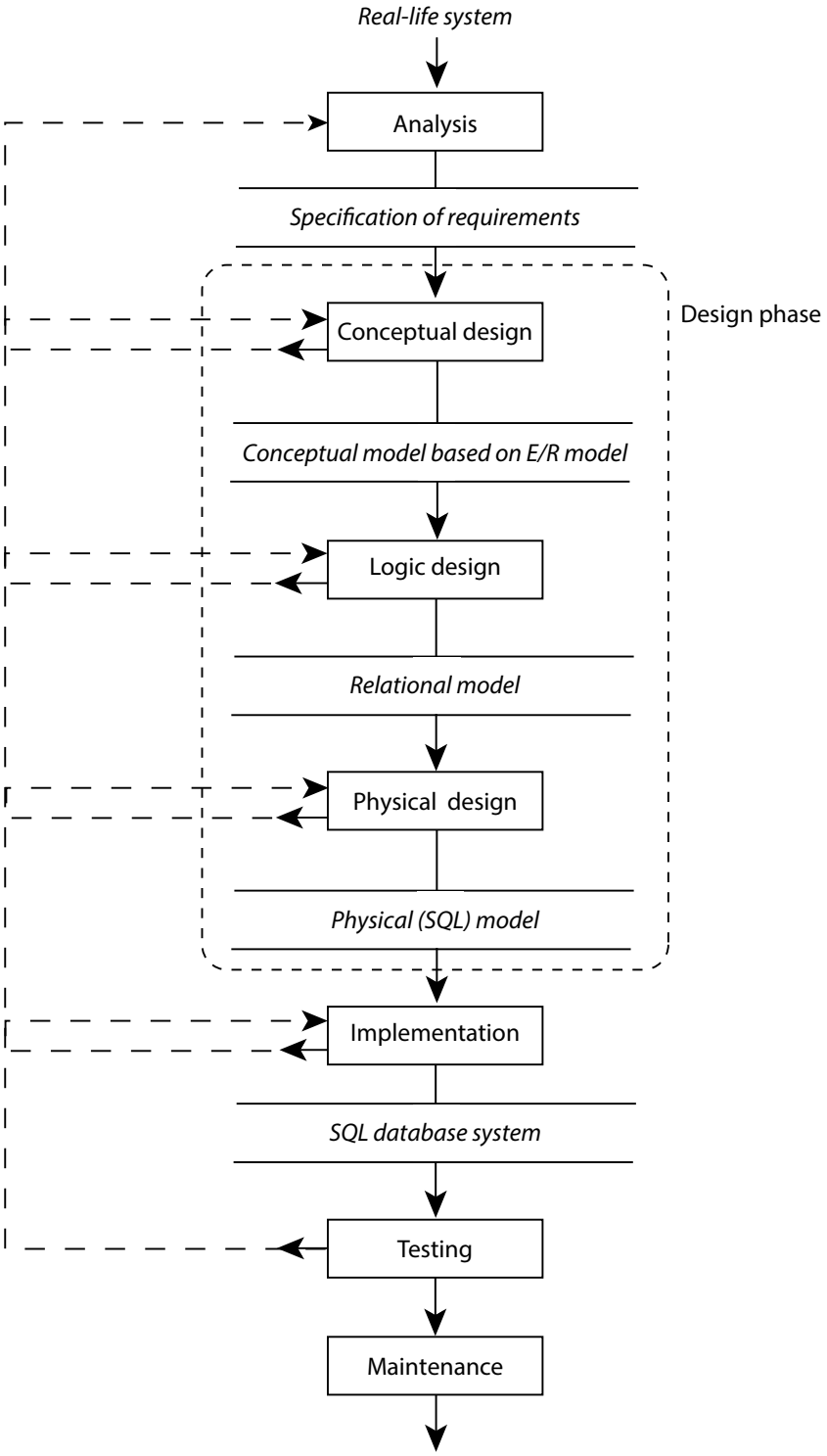


Figure 5.1. A development process for database systems. Rectangles represent processes and text between parallel lines, products. Dashed arrows provide options for iteration and looping. The design phase has been divided into several processes, grouped in a dashed box.

In this chapter we will look at the design phase, which is enclosed in a dashed box on the diagram. The SQL you already know will help with the implementation phase, as will Chapter 2 of Volume 2 of this subject guide.

The design phase of a database system usually follows a **top-down approach** – it starts with high-level design that largely ignores the details of implementation, and then it progressively adds implementation detail until it contains fully low-level design. For database systems, a top-down design phase can be divided into three distinct stages:

Semantic design. Sometimes also called **conceptual modelling**, this stage involves a high-level, abstract view, avoiding implementation details and focussing instead on the real-life information that must be modelled. The **Entity/Relationship (E/R) model** is the most common conceptual modelling technique, and will be considered in some detail below. It focuses on the **entities** – things or objects – in a system, and the **relationships** between them.

Logical design. This is performed at a lower level of abstraction, in this case creating a model that is implementable according to relational theory.

Physical design. At this stage, account is taken of the specifics of the DBMS to be used and the characteristics and requirements of the deployed system. There are likely to be two aspects to this process:

1. Adapting the logical model to better reflect the DBMS, either because the DBMS does not implement the full relational model, or because it offers extra modelling richness that will prove useful.
2. Designing the physical data structures – how the data is actually going to be stored and retrieved. This is something that depends heavily on the DBMS to be used, but is likely to involve indexes and caches as well as decisions about the structure of the tables themselves.

5.3 Conceptual modelling – the E/R model

5.3.1 Introduction

A **top-down** approach to design begins with a specification of the system that we want to implement, but the specification is at a high level of abstraction, usually using natural language. The approach ends with a model of the system that is ready for implementation. For database systems, the first stage of this approach is represented by **semantic design**.

Semantic design is a process of modelling – representing a real-life system by means of a set of concepts or semantic objects. We have introduced this stage using the term 'semantic' and also 'conceptual'. Both aspects are present in this stage, and the choice of name between the two is arbitrary.

The conceptual design process is carried out within a theory or model such as the entity/relationship model. A semantic model (or theory) provides a finite set of concepts by means of which a real-life system can be represented. By limiting the range of conceptual models available, semantic design simplifies and clarifies the result. By contrast, a natural language description of a real-world system is likely to be ambiguous – and those ambiguities can be hard to spot in a paragraph of text.

The most popular semantic model used in relational database design is the entity/relationship model. In the E/R model, the real world is represented

as entities, each of which is characterised by attributes, and between which relationships are established. Figure 5.2 summarises these concepts.

Concept	Description	Examples
Entity	A thing (that we want to model) that can be uniquely identified and that has some sort of independent existence.	Student, tutor, course, car, invoice, planet
Attribute	Information that describes an aspect of an entity.	Name, address, salary, title, fuel efficiency, value, diameter
Relationship	A connection or dependency between two entities.	[student] is taught by [tutor], [student] is enrolled on [course], [car] is made by [manufacturer], [planet] was visited by [mission]

Figure 5.2. Core concepts of the E/R model.

These concepts are not defined in the E/R model in a formal way – since they involve interpreting real-world objects and ideas, strictly formal modelling is difficult. One consequence of this is that the things being modelled can be correctly modelled in more than one way. For instance, the relationship from Figure 5.2 that a planet 'was visited by' a particular mission could also be modelled as an entity representing the visit. This sort of decision may be arbitrary, but it may also have an effect on the implementation. Similarly, an address could be an attribute, as suggested in the table, but it could also be an entity, and members of a household could be linked to it using a '**lives in**' relationship.

An E/R model of a real-world system is not usually intended to be directly implementable. This is partly because E/R theory is not directly implemented by a DBMS, but also because the aim of this stage of modelling is to focus on the system to be modelled rather than on the details of implementation. It is important to capture as much in the model as possible before involving technical concerns that might distort the model. This lack of a path for direct implementation means that the model must be translated into a more implementable model. In this chapter, after describing the construction of an E/R model, we will present a method for translating it into a relational model which, as we have seen, is much closer to the models used in DBMS implementations.

The E/R model is an abstract model, and does not depend on any particular symbolic representation. This makes it easier to translate it to other logical models such as the relational model or an object-oriented model. However, it is impossible (and probably undesirable) to model data and not make some constraints on the results, and there are certainly programming and data representation paradigms for which the model is unsuitable. These need not concern us here.

Apart from its concepts, the E/R model also includes a diagramming technique, the entity relationship diagram (ERD). In the next section, we introduce each of the concepts of the E/R model together with their associated ERD notation. You will find the concepts and diagram components described in the relevant chapters of both **Date** and **Connolly and Begg**. Those interested in further reading should read the paper that first proposed this model by Peter Chen, listed at the head of the chapter.

5.3.2 Core concepts

Entities

Chen (1976) defines an entity as 'a "thing" which can be distinctly identified. A specific person, company, or event is an example of an entity.' This definition is necessarily loose, and the decision of which parts of a real-life system should or should not be characterised as an entity is for the database designer. She or he will decide based on the nature of the system being modelled, on personal experience, and also on personal preference. In the context of database systems, the one guiding principle is this: an entity is something about which we need to store information.

Before we go further, it is important to distinguish between an **entity instance** and an **entity type**. For those familiar with object-oriented programming, this is analogous to the difference between an object and its class. The 'entities' we have listed up till now are entity types – they are concepts that represent a set of entity instances that share a set of common characteristics. For example, the entity type `Planet`, having attributes for `mean radius` and `maximum distance` from star, might have an entity instance Mars, with attribute values or 3,390 km and 249 million km respectively. Here on Earth, an entity instance of type person might be `Name: J.C.T. Jennings; Age: 10; Address: Linbury Court School, Dunhambury, Sussex, England`.

An entity type has a list of attributes, but (usually) has no values for them. An entity instance is a set of values for those attributes. Entity instances may have some values unknown or not specified (comparable with `NULL` in relational theory), but it is not permitted for them to have all their values unknown.

When designing a system, especially at this high level, we would expect to concentrate on entity types rather than instances, and the shorthand **entity** is used to mean an entity type.

There are two classes of entity types, based on their relationship to other entities:

Weak entity type. A weak entity type is one whose existence depends on the continued existence of other entities. For example, a customer's bank account depends on the continued existence of the person. If the customer were to die or, less dramatically, move their custom to another bank, then the account might be closed and their records deleted.

Strong entity type. A strong entity type is easiest to define as one that is not weak. It is an entity type whose entries will not be invalidated by the removal of any other entity.

It is perhaps an obvious point, but it should be made clear that these distinctions concern the existence of records in a database or objects in a system, not (necessarily) their real-world counterparts. An `employee` type might be weak because it depends on `factory`. If a factory is closed, and its instance removed, relevant employees may cease to exist as a result. Although this may represent the sacking of people in the real world, it certainly does not indicate that the people concerned no longer exist. When deciding whether an entity type is weak or strong, it is important not to expect that weak entities will physically disappear – they simply lose interest for the system that has been modelled.

Figure 5.3. Strong entities, with a single-line border, and weak entities, with a double-line border.

A strong entity type is represented in diagrams as a rectangle, labelled with the entity's name. A weak entity is the same, but has a double line for the rectangle. Figure 5.3 illustrates this with some examples.

Attributes

Each entity type includes a number of properties, called **attributes**. To put it another way, a set of attributes defines an entity type. For example, we might say that entity type `Student` has the attributes `name`, `address`, `date of birth`, `degree programme` and `enrolment date`. On the other hand, we might also say that the set of `name`, `address`, `date of birth`, `degree programme` and `enrolment date` defines the entity called `Student`.

As in relational theory, an attribute is defined on a **domain**, where a domain is the set from which an attribute can draw values. For example, the property `degree programme` is likely to draw from a set of programmes offered by the university, so that its domain could be defined as set `{'CS', 'CIS', 'IT'}` (if these were the only programmes on offer).

In diagrams, attributes are shown as ellipses, connected to their entities by a line.

Attributes can be characterised in various ways (as shown in Figure 5.4):

- **Simple versus composite**. A **simple attribute** has no internal structure within the application, it is atomic or scalar (like **all** values in relational theory). A **composite attribute** is one that has an internal structure that can be broken down into further attributes, themselves either simple or composite. For example, since a date can be broken down into a day, month and year, it can be seen as a composite attribute. The same is true of an address, which consists of separate semantically meaningful elements. These further attributes are shown as ellipses attached to the composite attribute. Since a composite attribute may be made up of further composite attributes, there may be several layers of these.
- **Single or multi-valued**. A **single-valued attribute** is one that, for a given instance, takes one value from the associated domain. For example, although a student's name may change, it is unlikely that the student will have multiple (full) names at the same time. Similarly, in most circumstances, the student will have a single date of birth.
- A **multi-valued attribute** can have more than one value associated with it. For example, a student may have several telephone numbers. Multi-valued attributes are represented by a double-lined border for their ellipse.
- **Base or derived**. A **derived attribute** can be deduced from a set of attributes already present in the system. The attributes used for that calculation need not all attach to the same entity. This may be simply that age can be calculated from a date of birth, but it might also be a more complicated deduction; for example, a student's fees might be based on a

combination of the student's nationality and degree programme, but also on parental income and other factors from attributes attached to various other entities in the system. Derived attributes are indicated by having a dotted or dashed border line for their ellipse.

A **base attribute**, on the other hand, cannot be deduced from other attributes and is the raw information for the system.

- **(Primary) key**. As in the relational model, it is usually important that entity instances can be uniquely distinguished, and identifying the minimal attribute or set of attributes required to identify an instance is important. The **primary key**, then, is the attribute or attributes that uniquely identify any instance of the entity type. In a diagram, an attribute that is used as a primary key is underlined.

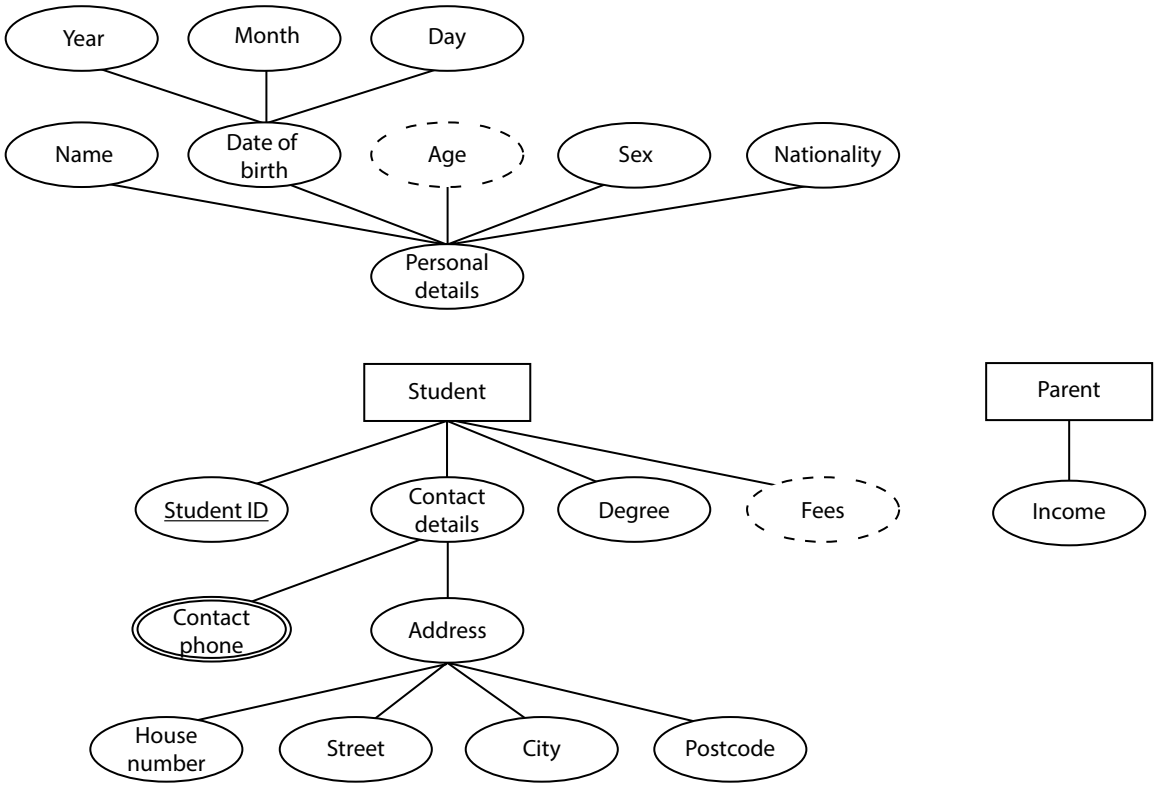


Figure 5.4. Attributes of the Student entity, including simple, compound, multi-valued and derived entities.

Relationships

Chen (1976) defined a relationship as 'an association among entities'. An example of such a relationship might be that a student 'is registered' with a department or 'is enrolled' on a degree programme.

As for entities, there is a difference between relationship instances and relationship types. The principle is the same – for example, the **relationship type** `Enrolled` might have a **relationship instance** that relates a student who has `StudentID` 112843 with the `Degree programme` IT. In this example, the entities being related are of different types, but that is not always the case. Consider the case where new students are assigned mentors – other students who have already been at the institution for longer. In that case, the relationship `Mentors` would connect two entities of the same type.

A relationship type is an association between entity types. It can be said to be **defined on** entity types. For instance, the relationship type `Mentors` is defined on one entity type, `Student`; whereas the `Enrolled` relationship

type is defined on two – `Student` and `Degree programme`. Just as is the case of entities, it is normal to shorten 'relationship type' to just 'relationship'.

Entities involved in a relationship are called **participants**. A relationship is represented as a diamond labelled with the name of the relationship. It is linked to the participating entities by line segments. If one entity in the relationship is strong and the other weak, then the diamond is drawn with a double line.

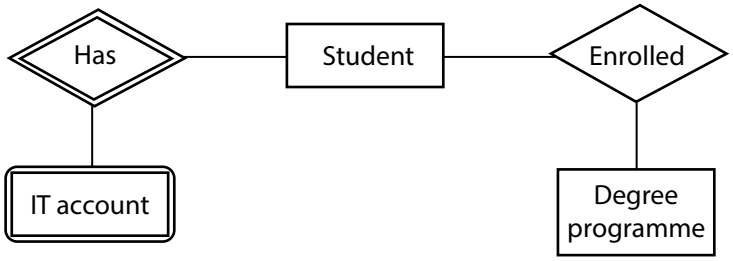


Figure 5.5. Relationships between strong entities and strong and weak entities.

E/R diagrams can get complicated and hard to read quite quickly, and so a whole system may be represented on multiple diagrams, and each diagram may only show some of the information for the entities being drawn. In particular, an entity's attributes may not all be shown on the main diagram showing relationships.

The number of participating entity types represents the **degree** of the relationship. Most commonly, relationships are of degree two – a **binary** relationship. The relationships in Figure 5.5 are both binary. The `Teaches` relationship in Figure 5.6 connects a `Tutor` with a `Course` and a number of `Students`. Since there are three entity types involved, this is called a **ternary** relationship. The `Mentors` relationship introduced earlier is **unary** or **recursive**, since it involves only one relationship type.

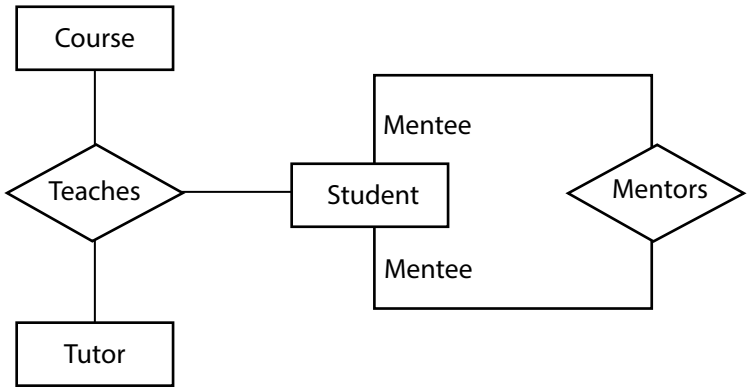


Figure 5.6. Ternary and unary (recursive) relationships, with roles labelled for the unary case.

It is important, to avoid confusion, that this way of counting the degree of a relationship only takes account of the entity types involved, not the number of entity instances. For example, although `Mentors` is a unary relationship (it only involves the `Student` entity type), it involves two entity instances – the mentor and the mentee.

One way to clarify the participation of entity instances in a relationship – especially in a recursive one – is to give the instances role names. These roles can be indicated by labelling the appropriate line segments. This is illustrated in Figure 5.6, with the mentor and mentee roles being made explicit.

Role names are also useful where the same entities are linked by more than one relationship, especially where they also involve different subsets of entity

instances. For example, there may be many tutors in a department, but only one of them is head of department at a time, and the roles are different even though they connect the same entity types. Figure 5.7 illustrates these relationships and shows how role labels clarify the situation.

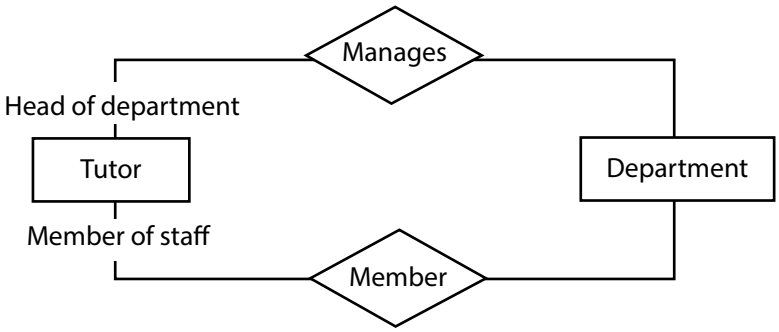


Figure 5.7. Relationships with named roles.

Relations can also have attributes. For example, we might want to associate some extra information with a tutor's membership of a department – joining date and position within the department. The attributes are shown in the same way as for an entity (Figure 5.8).

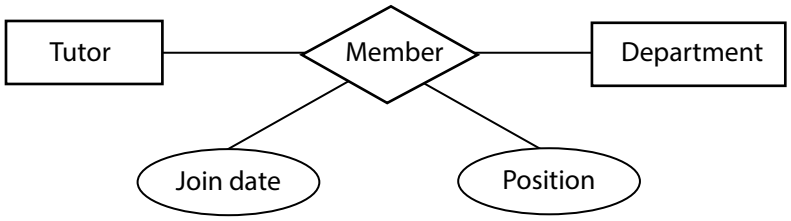


Figure 5.8. Relationships with attributes.

Some prefer not to allow relationships to have attributes and instead place an extra entity into the model – as in Figure 5.9. This is a matter of preference – allowing relationships to have attributes can make simpler diagrams, while banning them makes the underlying E/R model simpler. Whichever path is taken, the relational model that we produce and the resulting database design will be identical. We leave it for the reader to experiment, read around the subject and decide which approach to take.

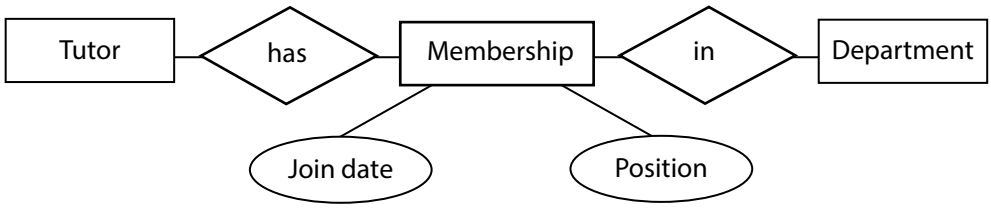


Figure 5.9. Avoiding relationships with attributes by replacing the relationship with a new entity.

Constraints on relationships

In this section, we look at two types of constraint:

- **participation constraints**
- **cardinality constraints**.

If all instances of a particular entity must have a given relationship, that entity's participation in the relationship is **total**. If participation is not always required, then its participation is **partial**.

As an illustration, consider a relationship modelling mothers and their children for a hospital maternity unit (Figure 5.10). Every child must have a mother (for our purposes here), and so every instance of the entity `Child` is involved in the 'has mother' relationship – participation is total. It is not the case that every adult in the database has a child (and if some are men, they are unlikely to be mothers), and so there are instances of the entity Adult that do not participate in the 'has mother' relationship – their participation is partial. Total participation may be distinguished in diagrams by replacing the usual single line that connects the entity to its relationship with a double line.

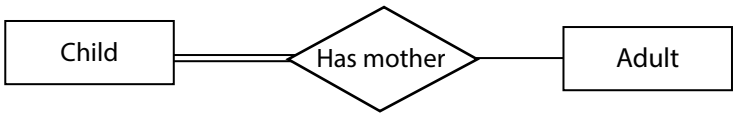


Figure 5.10. Partial and total participation in a relationship.

Cardinality constraints are concerned with the number of entity instances that are permitted in a given relationship. Cardinality constraints are generally reserved for binary relationships. This is because they are most useful when preparing the design for the relational model, at which point all relationships will have to be transformed to binary ones. This will be explained in the next section.

The most important information when designing a relational database for the cardinality of a relationship is whether more than one instance of an entity may be involved in the same relationship. In the preceding example, the Child was defined as having exactly one mother, while the mother might have multiple children. For a binary relationship `R`, with two entity types, `E1` and `E2`, each can either permit **one** instance or **many** instances. This gives four possibilities:

		E2	
		1	**Many**
E1	**1**	1:1	1:m
	Many	m:1	m:n

Since 1:m and m:1 are, in effect, the same thing, we are left with three common cardinality constraints:

- **One-to-one (1:1) cardinality**. This specifies that for any instance `e1` of `E1`, there is a single instance `e2` of `E2` such that the relationship `e1 R e2` exists. So, for example, if a course requires each student to submit a single, unique dissertation (and resubmission or retaking is not available), then each student will have at most one dissertation – none if they do not take the course. Each dissertation must have one and only one author. This cardinality is shown with the numeral `1`, as shown in Figure 5.11.

Figure 5.11. Cardinality constraints for the 1:1 relationship between Student and Dissertation.

- **One-to-many (1:m) and many-to-one (m:1) cardinality**. A one-to-many cardinality constraint specifies that, for any instance `e1` of `E1`, there may be several instances of `E2 – e21, e22, e23...` – such that `e1 R e21`, `e1` $R_{e}2_{2}$, `e1` $R_{e}2_{2}$, etc; but that for any instance of `E2`, there is only one instance e1 of `E1` that for which `e1 R E2`. This occurs quite often; for

example, where a teacher is the personal tutor of many students, but each student only has one personal tutor. As before, the entity with cardinality `1` is labelled with a `1`, but in this case, the other side of the relationship is labelled with an m, indicating an unspecified 'many' entities. This is illustrated in Figure 5.12. The many-to-one constraint is the same as the one to many, but with `E1` and `E2` swapped around (`m` on the right in the diagram).

Figure 5.12. A many-to-one relationship.

- **Many-to-many (m:n) cardinality**. A many-to-many cardinality constraint specifies that, given `e1` from `E1`, there may be several instances of `E2` – $e2_1$, $e2_2$, $e2_3$… – such that `e1 R` $e2_1$, `e1 R` $e2_2$ and so on. It also specifies that, given `e2` from `E2`, there may be several instances of `E1` for which the same relationship holds. One example, shown below, is of students and the courses on which they are enrolled. Since each student enrols on many courses, and each course can have many students, this is a many-to-many relationship. Since the number of students on a course and the number of courses a student takes are not related, labelling both cardinalities m could be confusing, and so a new variable is named, *n*.

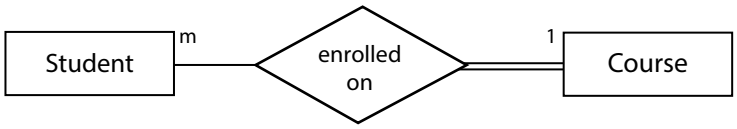


Figure 5.13. A many-to-many relationship.

Although often it is enough simply to indicate whether the cardinality is 1 or many, sometimes having more detail is useful. A more precise way to specify a cardinality constraint is to provide a minimum and a maximum value for the number of instances in the relationship. There are several conventions for doing this, but one of the clearest looks like this:

`(<min>, <max>)`

Figure 5.14 illustrates the use of this notation for the three relationships that we have seen in Figures 5.11–13.

For the simple one-to-one relationship of student to dissertation, the only difference is that we can now use the (min, max) notation to express the fact that every dissertation in the system must have been submitted by a student, but that not every student will have submitted a dissertation yet.

For the many-to-one personal tutor relationship, we can represent limits on the number of students associated with each tutor. In this case, each tutor must be the personal tutor of between 5 and 20 students.

Finally, for the many-to-many relationship indicating course enrolment, we show that students may be enrolled on up to eight courses at any time. We also show that a course must have at least five students enrolled, but that there is no maximum number who can enrol (indicated by keeping the *m* for many).

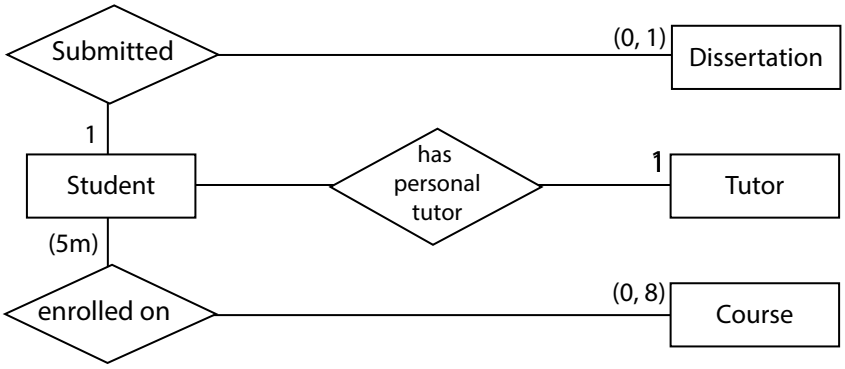


Figure 5.14. Three relationships with (max, min) cardinality constraints.

Activity

What might the cardinality be for the following relationships:

- Between students and their correspondence addresses?
- An 'Advertises on' relationship between websites and advertisers?
- Between drivers and driving licences?
- Between novels and characters in them?

Cardinality constraints are an important component of database design. Review the database objects you have designed so far and how they relate. Can you work out what cardinality constraints they have? Are some combinations more common than others?

Type hierarchies

We have already noted the distinction between entity types and entity instances – `Student` entity instance belongs to the entity type `Student`. It is also possible for instances to belong to more than one type. For example, a `Student` instance is also of type `Person`. Since every `Student` is of type `Person`, but not every `Person` is a `Student`, we say that `Person` is a **supertype** of `Student` and `Student` is a **subtype** of `Person`.

There are several ways of modelling types and subtypes, but we will consider only a model in which a type can only have one immediate supertype. In this model, a subtype inherits all the properties of its supertype – a `Student` has all the properties associated with a `Person`. Subtypes also participate in all the relations that the supertypes participate in. Another consequence of this model is that it can always be drawn as a tree (see Figure 5.15).

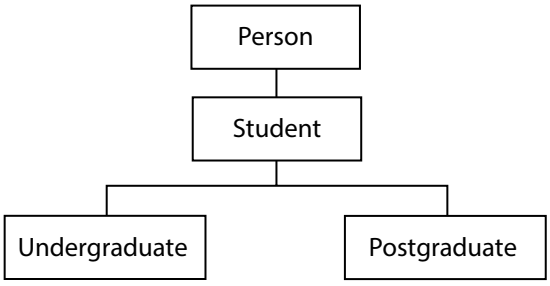


Figure 5.15. A type hierarchy represented as a simple tree.

5.3.3 Possible flaws

As with any modelling exercise, it is important to capture all the relevant characteristics of a system in the model. There is always a danger that a model that looks complete is missing vital information. Although there are many ways in which this can happen, there are two common patterns that are generic problems – that is, they are independent of the application area – and are often the cause of difficulties. These patterns are called **fan traps** and **chasm traps**.

Fan traps

The fan trap is named after its appearance in diagrams, as multiple entities connecting to a single entity. The problem arises if indirect relationships shown on the diagram, via the single connecting entity, stand in for direct relationships. In many cases, one does not need to show the direct relationship, but often, the diagram is missing vital information.

Consider two simple E/R diagrams. Figure 5.16 shows students enrolled on a degree programme. For each degree programme available, there is exactly one administrator responsible for coordinating the programme. There is an implicit relationship between students and administrators – each student will have an administrator responsible for her or his programme, but labelling this extra relationship would tell us no more than the current diagram shows.

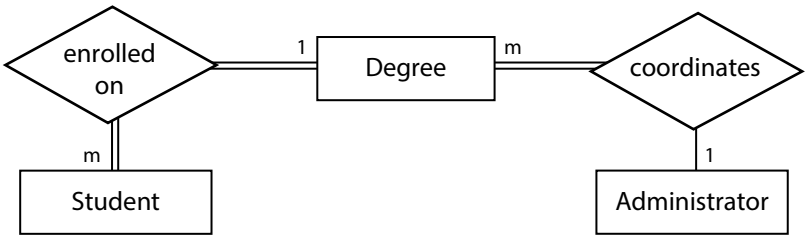


Figure 5.16. The implied relationship between Student and Administrator is unambiguous here. Given entity instances for all relationships – for example, for any given Administrator, one can trivially find all the students on the programmes they manage.

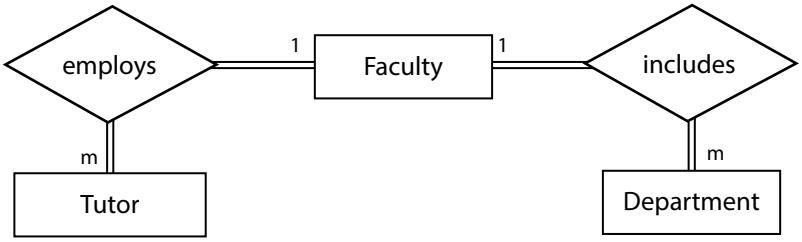


Figure 5.17. With two 1: m relationships (the faculty employing tutors and including multiple departments), the relationship between Department and Tutor is now ambiguous. Associations between Department and Tutor are now no longer deducible.

Figure 5.17 shows a slightly different case. Here, we have university departments that are organised into faculties. Each faculty contains at least one department and each department is part of exactly one faculty. The faculty maintains a list of all tutors that it employs. In this scenario, the implicit relationship is between the tutors and departments, but it is now impossible to tell from the model which tutors are employed in which department.

This problem arises because instead of a 1:1 relationship, we have a 1:m, or more specifically, we have two relationships of 1:m where the entity with a cardinality of 1 is common to both. This creates a fanning out of possibilities

illustrated in Figure 5.18. In this example, we have three tutors, T1, T2 and T3. All are associated with faculty F1, which has two departments, D1 and D2. From the relationships in the model, it is impossible to connect a tutor to the appropriate department.

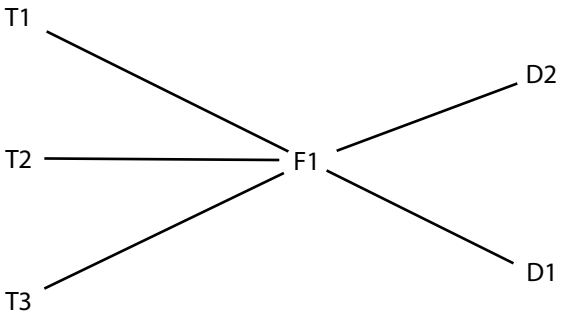


Figure 5.18. Three tutors, T1, T2 and T3 are all employed in faculty F1, but which department are they in?

Since the problem is that a hidden relationship is masked by a pair of 1:m relationships fanning outwards, the ambiguity can be resolved by redrawing the diagram to change which entity is the central one to which the others connect. Figure 5.19 illustrates one resolution of the fan trap. As Figure 5.20 shows, the resulting relationships are clear and unambiguous.

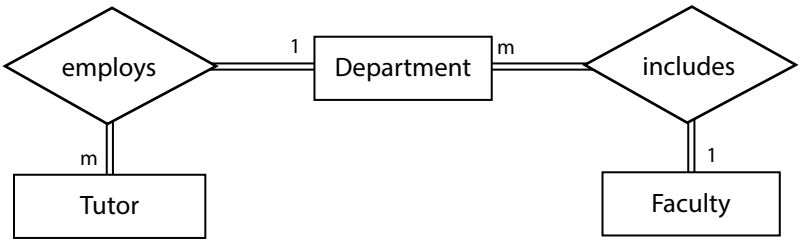


Figure 5.19. Resolving the fan trap involves re-arranging the entities, removing one of the explicit relations and making the implicit relation explicit.

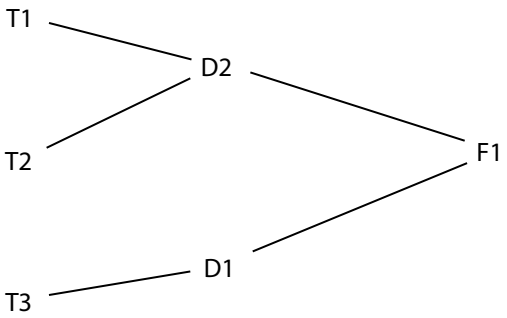


Figure 5.20. Using the example used in Figure 5.18, we can see that it is now possible to identify both the department and faculty for any given tutor. Graphically, the fan or cross shape has disappeared.

Chasm traps

In these examples, the participation of entities is total – all tutors have a department, all departments have a faculty, and so on. What happens if this is not the case? How would it change matters, for example, if some tutors were employed directly by the faculty and had no departmental links. Figure 5.19 would have to be modified as shown in Figure 5.21, to reflect this change.

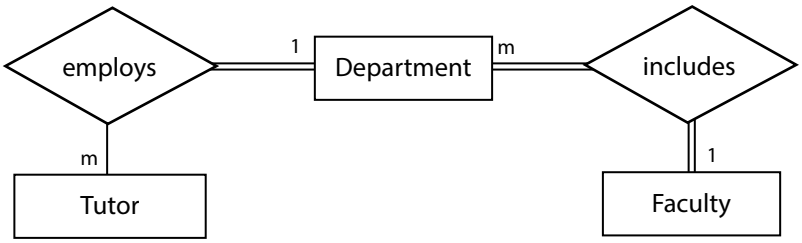


Figure 5.21. A slightly altered example – now not all tutors are attached to a department.

In this situation, is it always possible to work out which faculty employs every tutor? Clearly not – if there is no departmental link, then there is no chain of relationships to follow to get to the faculty. This is illustrated in Figure 5.22, which shows the gap that gives this problem its name – the **chasm trap**.

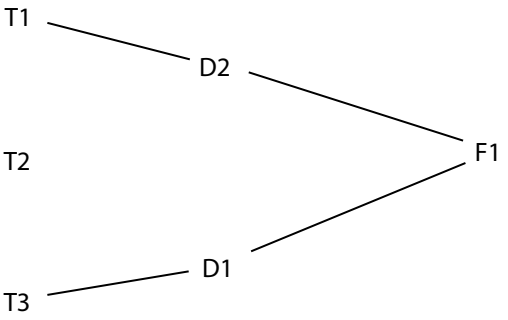


Figure 5.22. An illustration of the chasm trap. Tutors T1 and T3 in this example are employed directly by departments D1 and D2. Tutor T2, however, is employed directly by the faculty. It is impossible to record this relationship if only the relations shown in Figure 5.21 are used.

A chasm trap occurs when both:

1. Two entities are linked by a pathway that travels through at least one other entity.
2. Participation of at least one of the entities on the path is partial for at least one of the relationships.

Since the problem is the lack of connection between two entities, the solution is simply to make that implicit connection explicit. Figure 5.23 shows the E/R diagram that results from that strategy.

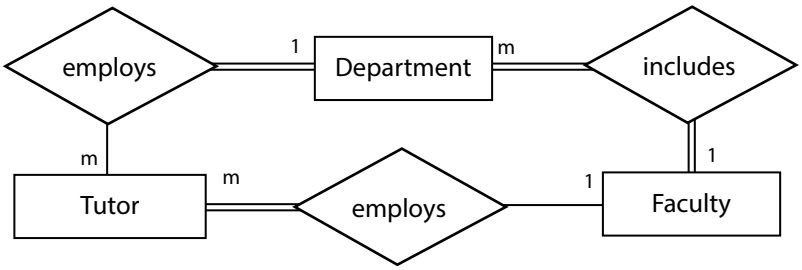


Figure 5.23. Adding the third relationship to the model explicitly resolves the chasm trap.

Given the risks posed by fan and chasm traps, an obvious question would be 'why not always make implicit relationships explicit?' Certainly, that would avoid the risk of missing information or ambiguity. One answer is that doing so is less economical, in terms of the representation and of storage in the implemented database. Another answer, which relates to questions of normalisation that will be considered later in this chapter, is that if we store information that can be deduced from information already present elsewhere in a database, there is a greater risk that the two will become out of sync, and result in an inconsistent database. For that reason, it is generally good practice

to minimise the amount of 'redundant' information; that is, information that could be worked out from the rest of the database. Only if the rest of the database is incomplete, as in the case of the chasm trap, should the data model be expanded to include the missing information.

Developers should be aware that in some cases, a chasm trap may be a potential rather than an immediate problem. If all tutors are currently assigned to departments, there is no chasm trap, but if there is the potential for that situation to change, the model should be designed as in Figure 5.22, to avoid the risk.

5.3.4 Conclusion

Developing any good conceptual model, including an E/R model, depends on a good understanding of the real-life system you are modelling. If the details of the system are not properly understood, then there are risks of serious mistakes in the model. For example, without knowing that a tutor might be employed by a faculty and not by a department, a developer could easily create a chasm trap and not realise it.

As we have already noted, there is no single, unique way to model a system using the E/R approach. Different analysts may treat the same system in different ways. This means that there is no 'correct model' to compare your attempts with: some models will be incorrect or incomplete in how they model the system, but there are usually many different correct and complete models.

Activity

Taking one of the systems you have modelled already, experiment with making changes to your model. Which relationships can be turned into entities? Can any entities become relationships? Are all the versions you can devise equally good, or do some make more sense or seem more consistent?

The E/R model forms the foundation on which the relational modelling is built. Decisions taken at this stage influence the logical modelling stage. It is important to focus on the real-life system during conceptual modelling; it is best to avoid letting knowledge of the logical modelling that will come next influence your conceptual model – the analysis should be kept 'clean' and abstracted from implementation issues.

The next section presents ways to transform your E/R model into a relational model.

5.4 Transforming an E/R model into a relational model

The E/R model represents a clear description of the system to be implemented as a database. It is not readily implementable because no DBMS implements its concepts directly. Because of this, it must first be transformed into a model that is directly supported by DBMSs – the relational model.

This mapping is not entirely deterministic – there is no unique relational model corresponding to a given E/R model. There are two main reasons for this: some elements of the relational model, such as foreign key rules, are not specified in the E/R model; and some E/R configurations can be represented in more than one way in the relational model.

As a result of this, there are different approaches to the mapping process. The approach we describe here is close to the one described in **Connolly and Begg**. It consists of two stages:

1. Transform the E/R model into an equivalent E/R model, removing structures that are not supported by the relational model.
2. Transform the resulting E/R model into a relational model.

To make this easier to explain, we explain the second stage first, but the order above is the one that will be used in practice.

5.4.1 Entities

When mapping entities from an E/R model to a relational model:

- Entities are mapped to base relations in the relational model.
- All of an entity's simple attributes map into attributes of the base relation.
- Compound attributes cannot be represented and should usually be turned into simple attributes.
- The primary key of the entity becomes the primary key of the relation.
- If the entity is weak, then the relation must contain a foreign key to the strong entity on which it depends. The foreign key must be part of the entity's primary key.

Other candidate keys for the relation will not be contained in the E/R model, and cannot be deduced.

Figures 5.25 and 5.26 give an example of an E/R diagram and the relational model that results from applying the rules above. For the table in Figure 5.26, the attributes that constitute the primary key – Name and Date of Birth – are underlined. This example is to illustrate the process only – in real-life applications, this is not a very strong primary key, since there is a risk of two people having the same name and being born on the same day.

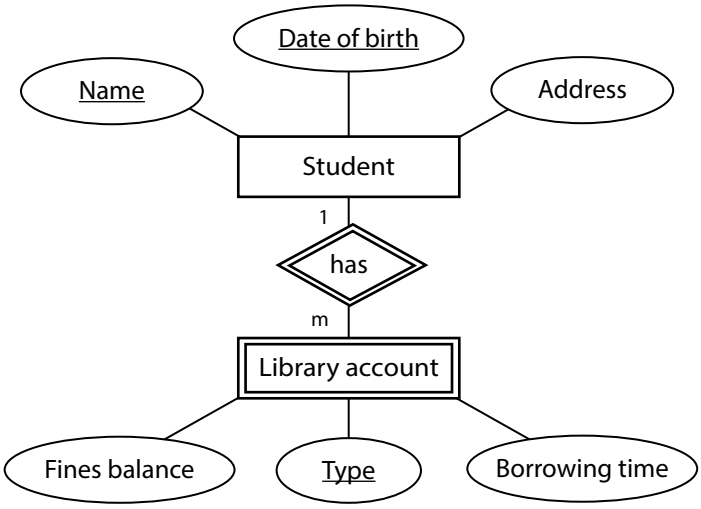


Figure 5.25. Simple E/R diagram with one strong entity and one weak entity. The resulting relational model is shown in Figure 5.26.

`Student`

Name	**Date of birth**	**Address**

`Library account`

Name	**Date of birth**	**Type**	**Fines balance**	**Borrowing limit**

Figure 5.26. A relational model corresponding to the above E/R diagram. Attributes that constitute the primary key are underlined.

Activity

Look at the Student entity from Figure 5.4. Which attributes would you now be able to map into a relational model? What sorts of attributes do not work? How might those be mapped into a relational model? We will address these issues shortly, but first, consider what strategies you might use.

5.4.2 Relationships

The mapping of relationships is slightly more complicated. In general, one participant in the relationship will carry the primary key of the other as a foreign key. The example in Figure 5.26 illustrates this, with the primary key (`Name` and `Date of birth`) from `Student` being incorporated into the primary key of Library account as a foreign key.

- One-to-one relationships:
  - If participation is partial on both sides, choice of which entity has the foreign key is arbitrary.
  - If participation of one entity is total, and the other partial, the entity that has total participation, E1, say, carries the foreign key. The foreign key fields in E1 can be constrained to ban null values, enforcing the total participation.
  - If participation is total on both sides, choice of the entity with the foreign key is arbitrary. Consider whether the two entities can be merged into a single entity.
- One-to-many relationships (where E1 is the 'single' and E2 the 'many' entity):
  - Represent as a foreign key in E2 referencing E1.
- Many-to-many relationships cannot be mapped directly into a relational model. We address this issue later.

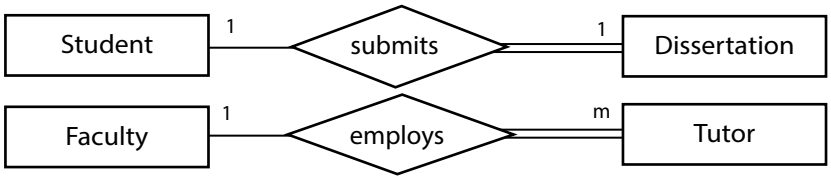


Figure 5.27. An example of a 1:1 relationship and a 1:m relationship. Figures 5.28 and 5.29 below show implementations of these using the relational model.

`Student`

Student name	**Student DOB**	**Degree**

`Dissertation`

Title	**Student name**	**Student DOB**	**Mark**

Figure 5.28. For a 1:1 relationship, one entity takes the other as a foreign key. If one of the entities has total participation in the relationship – here Dissertation – that will be the one to take the foreign key.

`Faculty`

Faculty name

`Tutor`

Tutor name	**Tutor DOB**	**Faculty name**	**Salary**	**Role**

Figure 5.29. For a 1:m relationship, the 'm' entity takes the foreign key.

5.4.3 Type hierarchies

We have introduced a type system where the properties of a supertype are inherited by its subtypes. Since relational theory has no type inheritance, this behaviour must be recreated where necessary. How this should be done depends on whether the subtypes are **disjoint**. What this means is whether it is possible for an entity to belong to multiple subtypes.

To take a simple example, the class Resident of people living in an area has two subtypes: Owner-occupier, consisting of people who own the property they live in, and Tenant, who pay rent to a landlord. These subtypes, shown in Figure 5.30, are mutually exclusive – they are disjoint.

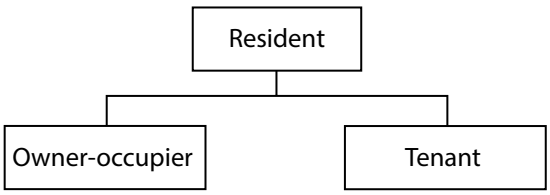


Figure 5.30. Two disjoint subclasses of resident – Owner-occupier and Tenant.

By contrast, the type hierarchy shown in Figure 5.15, with Undergraduate and Postgraduate as subtypes of Student, although **usually** disjoint, is not necessarily so. It is rare for postgraduate students to be undergraduate students at the same time, but this does not break any rules for a part-time PhD student to enrol on a part-time undergraduate degree in a different field.

If the types are disjoint, each subtype should be given a separate base relation which includes all the attributes that would have been inherited from the parent type. This approach makes sense because there is no need to duplicate information in this system. Although the same attribute names are duplicated in the different relations, they are never filled in twice for the same entity.

Figure 5.31 shows the base relations resulting from this treatment of the subclasses of Resident.

`Owner-occupier`

Name	**Address**	**Property value**	**Has mortgage?**

`Tenant`

Name	**Address**	**Property value**	**Landlord-name**	**Agents**

Figure 5.31. Figure 5.30 adapted to a relational model. Note the repeated fields.

On the other hand, if an entity might belong to multiple subtypes – if it is not disjoint – the parent type should also be represented as a base relation, with the primary key of the parent acting as a foreign key in the subtypes. This ensures that information is not duplicated for entities that appear in more than one relation.

Figure 5.32 illustrates this approach for the Student example.

`Student`

Student ID	**Name**	**Date of Birth**	**E-mail**	**Outstanding fees**

`Undergraduate`

Student ID	**Personal tutor**	**Year of study**

Postgraduate

Student ID	**Supervisor**	**Graduate school contact**

Figure 5.32. Adapting a non-disjoint type hierarchy requires at least one extra base relation. Note here that only the primary key occurs in multiple tables, and no information is duplicated, even if the same student is recorded as both an undergraduate and a postgraduate.

Activity

What would the effect be of a student enrolling on multiple undergraduate or postgraduate degrees? If you think there would be a problem, how would you fix it? If not, what additional fields would cause problems, and how would you model those safely?

5.4.4 Limitations

Having considered what can and what cannot be mapped to a relational from an E/R model, we can now assess why there is a need for a preliminary stage before the transformation can be carried out. To summarise the limitations:

- Composite, multi-valued and calculated attributes are not supported.
- Relationships are represented using foreign keys, so they:
  - cannot have attributes
  - must be binary or unary – they cannot connect more than two entities
  - must be either 1:m or 1:1 – m:n cannot be represented.
- Recursive relationships are not supported.

In order to prepare the E/R diagram for conversion to a relational model, then, we must first adapt for these limitations.

5.4.5 Preparing the E/R model

Before it is mapped into a relational model, the E/R model must be transformed into an equivalent E/R model that only uses structures that are supported by the relational model. It may be the case that no change is needed. Alternatively, it may be the case that transforming the model to accommodate these limitations does not result in a fully equivalent model. Usually, the changes are superficial and unimportant, but caution is recommended.

Flatten composite attributes

A composite attribute is an attribute that has a structure of its own – an attribute with attributes. Since relational theory requires attributes to be scalar, compound attributes are not supported. The simplest resolution to this is simply to attach all the simple attributes directly to the entity. Figures 5.33 and 5.34 show this process.

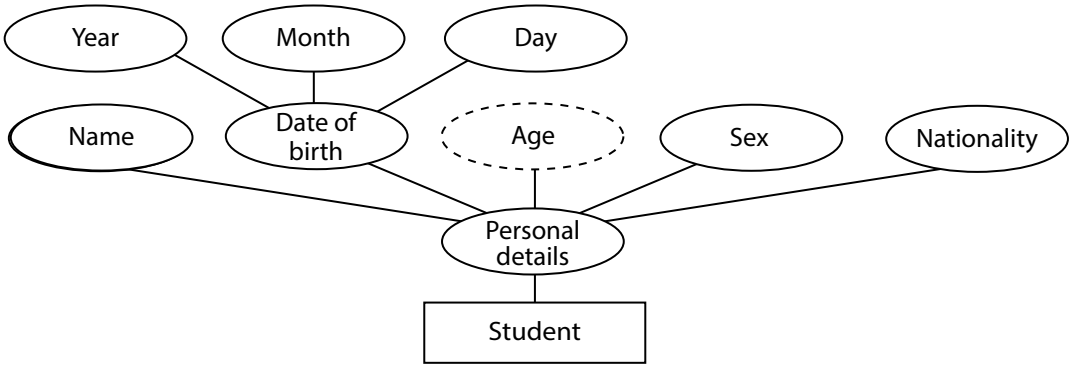


Figure 5.33. An extract from the E/R diagram from Figure 5.4, showing compound attributes.

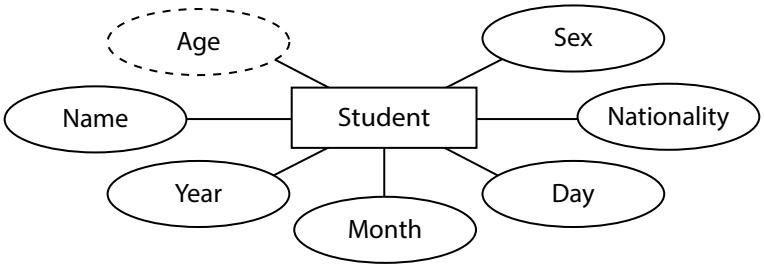


Figure 5.34. A 'flattened' version of Figure 5.33, by attaching all simple attributes to the entity rather than to a compound attribute. This can lead to naming ambiguities, and we would normally rename many of the attributes here.

The models represented in Figures 5.33 and 5.34 are not quite equivalent. In flattening the structure, we lose grouping of attributes and a label for each of those groupings. For example, Figure 5.33 groups `Year`, `Month` and `Day` together and labels them Date of birth. The richer structure can be made easier for humans to work out by using a naming convention (in this case, something like `DOB-Year`, `DOB-Month` and `DOB-Day`), or by using more complex domains for a single attribute (`DOB`). In some cases, it may be better to convert the compound attribute into an entity in its own right, so that it can maintain its own attributes, but the flattening approach is more commonly useful.

Eliminate multi-valued attributes

A multi-valued attribute is an attribute that itself has a 1: m relationship with the entity to which it belongs. The example used for this in Figure 5.4 was students and their contact telephone numbers – some students might have a fixed telephone at home and a mobile phone, or they might have term and holiday numbers, or multiple mobile phones. Since relational theory requires that data models be atomic, they cannot hold multiple values.

This is resolved by transforming the attribute into a new entity with a 1: m relationship to the parent entity, as shown in Figure 5.35.

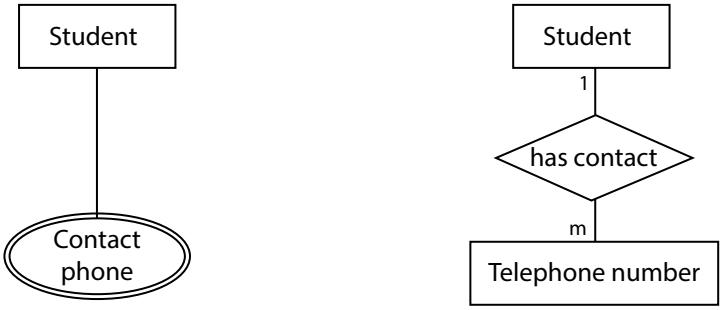


Figure 5.35. Original multi-valued attribute (left) and its resolution (right) by creating a new entity type and relationship.

Eliminate calculated attributes

Removing redundant information is an important part of database design. Calculated attributes are necessarily redundant information, since they are calculated from data elsewhere in the model. As a result, calculated attributes should be removed from the model.

Since calculated attributes are usually a reflection of use of the data rather than the structure of the data itself, they can be addressed using queries as views.

Eliminate relations with attributes

We have discussed earlier in the chapter the issue of relations having attributes. If the model includes these, they must be removed by converting the relation to an entity as shown in Figures 5.8 and 5.9.

Reduce complex relationships

A complex relationship – one which connects three or more entities – must be reduced to binary relationships. Figure 5.6 showed a ternary relationship, reproduced here as Figure 5.36.

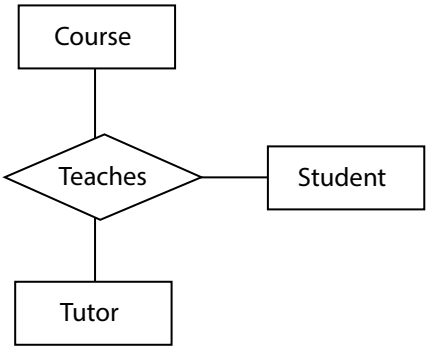


Figure 5.36. A simple ternary relationship.

There are several ways to reduce a system with complex relationships to one with only binary relationships. One is to transform the relationship itself into an entity, as shown in Figure 5.37.

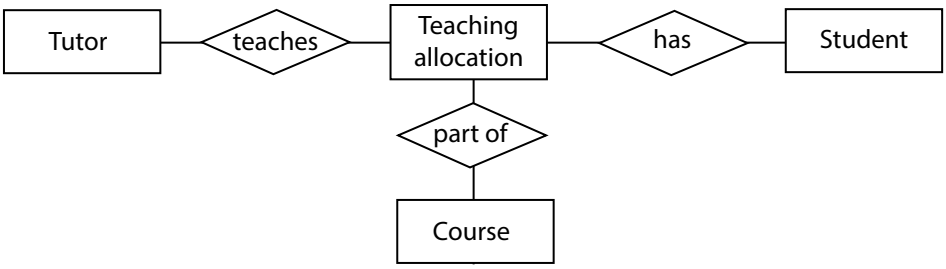


Figure 5.37. The ternary relationship can be resolved by creating a new entity (here Teaching allocation). Each of the original entities now has a binary relationship with the new one rather than with each other.

Alternatively, the implied binary relationships between each pair of entities can be taken separately turning, for example, one ternary relationship into three separate binary entities, as shown in Figure 5.38.

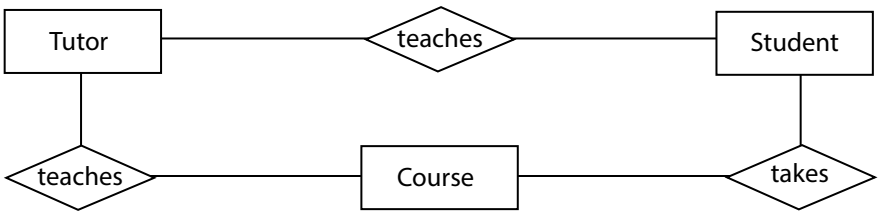


Figure 5.38. Another way to reduce the ternary relationship to binary ones is simply to connect each pair of entities involved in the relationship with an explicit binary relationship.

Eliminate m: n relationships

An m: n binary relationship between two E1 and E2 is transformed into a new entity linked to each of E1 and E2. In each case the new relationship will be m:1. For instance, the relationship teaches linking `Tutor` and `Course` might be replaced as shown in Figure 5.39.

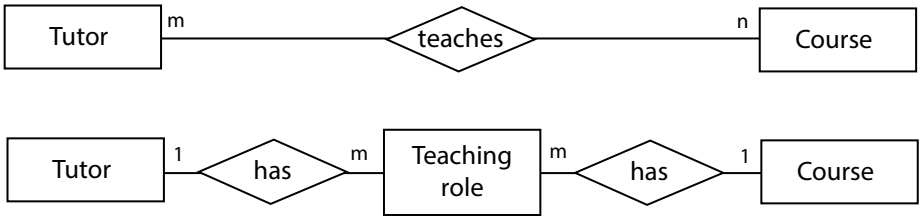


Figure 5.39. The m: n relationship shown above can be turned into two 1:m relationships connecting to a new entity (below).

It is possible to transform an E/R model directly into a relational one without the extra stage of rewriting it as described above. For that to be possible, the transformation to a relational model becomes more complicated – Date, for instance, describes such a process. Both approaches are equally reasonable, and should produce equivalent models, but we recommend the two-stage process as we find it clearer.

The relational model that results from these processes will not be ready to be implemented without a further stage. We have mentioned how important it is to control the amount and nature of repeated or redundant information in the database. The process of **normalisation**, described in the next section, is the method for ensuring that duplication is controlled.

5.5 Normalisation

The relational model resulting immediately after conceptual modelling takes place can be directly implemented, but we do not recommend it. This is because it contains redundant data. This redundancy if not eliminated can make operations that update the data more complicated or more likely to cause errors and inconsistency. Problems caused by an update operation are called **update anomalies**.

A relation causes anomalies if it has certain properties. More helpfully, update anomalies can be avoided if we ensure our relations are lacking those properties. A set of criteria that are required to avoid certain updating anomalies is called a **normal form**, and relations that satisfy those criteria are said to be in that normal form. There are several, increasingly strict, normal forms that relations can be in.

An important concept that these normal forms rely on is the treatment of **functional dependencies**. A functional dependency indicates a particular form of redundant information that can be removed from the system by

altering the relation. A given normal form will require the removal of certain sorts of functional dependencies.

In the sections that follow, we provide examples of updating anomalies, introduce the functional dependencies that cause them, and introduce the normal forms that guarantee that they can be avoided.

5.5.1 Update anomalies

This section presents some anomalies that database updates might cause in a relation. Our intention here is to show why normalisation is important.

To do this, we will use an example relation called `Contracts`, shown in Figure 5.40. The relation contains information about suppliers to a company, the parts they supply and the quantity they are supplied in. Each supplier is associated with one city only, but can supply several different parts.

`Contracts`				
Supplier	**Supplier city**	**Location Status**	**Product**	**Quantity**
Electricals Ltd	London	10	Screw	1,000
Electricals Ltd	London	10	Nut	1,000
Bits & Bobs	London	10	Socket	30
DIY House	Coventry	30	Screw	2,500
M&M Services	Leeds	40	Screw	500
M&M Services	Leeds	40	Plug	50

Figure 5.40. The Contracts relation.

Insertion anomalies

Suppose that a new supplier has been approved. The company, *Bright Sparks Co.*, are based in Manchester, a location with a status of 30. To add this information, we would expect to insert a row for the company, but here, we encounter a problem – what should we put in the more contract-related columns of the table, Product and Quantity? Since making up values just to store the details of the company is clearly wrong, the better option looks like using NULL for both, but this cannot work. As we will see later, the primary key of this table must be `{Supplier, Product}`, and columns that are part of a primary key may not be NULL.

An **insertion anomaly** arises when useful information cannot be added to a table because to do so would require information about something else at the same time, or the use of NULL values where they are not permitted. It arises because the table is being used for multiple purposes – in this case, it has information about the locations of companies, the products that they supply and the number of those that are to be bought.

Deletion anomalies

Suppose *DIY House* no longer supplies our company with screws, although it might supply other hardware in the future. The tuple corresponding to their supply must be removed, but since the information about the company itself is contained in the same relation, deleting the tuple means we will no longer store the location of the company or the location status of Coventry.

This **deletion anomaly** does not result in inconsistent information. Instead it results in the loss of information that we might not have intended to delete. There is no way to separate deleting information about a product supplied by a company from deleting information about that company.

Modification anomalies

Suppose that *M&M Services* has moved to Birmingham. The Supplier city must be changed accordingly. This information is stored in two places – in two separate tuples – in the relation. In a larger relation, the update might result in hundreds or thousands of changes. If the commands were not properly performed on all the tuples to be changed, or if the procedure aborted midway through, the database would be inconsistent, since some tuples might list the city as Leeds and others as Birmingham.

This is a **modification anomaly**, because if a modification is not performed completely correctly, it leaves the database in an inconsistent state.

Activity

How many other insertion, deletion and modification anomalies can you find in this relation?

None of these anomalies would have been possible had the relation been modelled as three relations – Contracts, Suppliers and Status, as shown in Figure 5.41. The problem is to know when a relation does or does not present any update anomalies. If a relation does present update anomalies, we must know how to remove them. These two issues will be answered in the following sections.

Contracts		
Supplier	**Part**	**Quantity**
Electricals Ltd	Screw	1,000
Electricals Ltd	Nut	1,000
Bits & Bobs	Socket	30
DIY House	Screw	2,500
M&M Services	Screw	500
M&M Services	Plug	50

Suppliers			Status	
Supplier	**City**		**City**	**Status**
Electricals Ltd	London		London	10
Bits & Bobs	London		Coventry	30
Dly House	Coventry		Leeds	40
M&M Services	Leeds			

Figure 5.41. A three-relation model for the same data as shown in Figure 5.40. This version does not present the update anomalies discussed above.

Activity

Before you move on to the next sections check whether the relations in Figure 5.41 really do have none of the anomalies we have discussed. What has been done to remove them? Is there a pattern?

Looking at Figures 5.40 and 5.41, do you think the result uses more storage space or less storage space? How would it scale for a much larger list of contracts?

5.5.2 Functional dependencies

Update anomalies are caused by redundant data. Redundant data may occur in a relation because links or **constraints** connect attributes. These constraints are the extra functional dependencies.

The `Contracts` relation in Figure 5.40 has, as its primary key `{Supplier, Part}`. This means that if we know the value of the `Supplier` and the `Part` fields, the values of all the other fields can be determined (because no other tuple in the table will have the same `Supplier` and `Part` values). That means, for instance, that if we are given the values `{Electricals Ltd, Screw}`, then we know that the other fields will be `{London, 10, 1000}`. This represents a functional dependency from `{Supplier, Part}` to `{Supplier city, Location status, Quantity}`. That is not a problematic dependency – it is part of the definition of a primary key.

However, we also know that a supplier has only one home city, and that the location status depends entirely on the city. That means that, given a supplier name, we know the city and status. These represent extra functional dependencies that are not related to the primary key. There is one dependency from `Supplier` to `Supplier city`, and another from `Supplier city` to `Location` status. These are the cause of the redundancy in the relation. The process of normalisation removes these excess dependencies.

Functional dependencies and their characteristics are clearly crucial to the process of normalisation. Before we consider the normal forms, we must define functional dependencies.

Functional dependency. Given a relation, R, and two sets of attributes, X and Y, belonging to R, we say that Y is functionally dependent on X if and only if, for any legal value of X there is exactly one value of Y associated with it.

If `Y` is functionally dependent on `X`, we can also say that `X` functionally determines `Y` or that there is a functional dependency (FD) from `X` to `Y`. We can also show the relationship symbolically as `X→Y`. We can show functional dependencies between multiple columns either as `{X, Y} → Z` or simply as `XY → X`.

In other words, once we have a value for `X`, the `Y` value is uniquely defined. The expression 'for any legal value of `X`' means that the dependency is a characteristic of the data being modelled, not just of the few data points that are in a table. To put it in relational terms, the dependency applies to relation variables rather than just relation instances.

It is not necessarily the case that if `X→Y` then `Y→X`. In the case of the suppliers and their cities, since a supplier has only one city, we can work out the city given a supplier. However, it is not the case that there is only one supplier based in each city. Given, for example, London, we could not derive a single supplier.

Of course, some dependencies are of no practical interest. For example, it is clear that `X→X`, but this tells us nothing. A **trivial** functional dependency is one where the right-hand side is a subset of the left-hand side.

Figure 5.42 shows some functional dependencies from the Contracts relation, several of which are trivial.

{Supplier, Part} → Supplier
{Supplier, Part} → Location status
{Supplier, Part} → Quantity
Supplier → Supplier city
Supplier → Supplier
{Supplier, Supplier city} → Supplier city
{Supplier, Part} → Supplier city
Supplier city → Supplier city
Supplier city → Location status
{Supplier, Supplier city, Part} → Quantity
Part → Part

Figure 5.42. Some of the functional dependencies from the Contracts relation of Figure 5.40. Some of these are trivial. Of those that are not trivial, some are still relatively uninteresting.

The left-hand side of an FD is called the **determinant**, and the right-hand side is called the **dependent**. Looking at two of the FDs in Figure 5.42 – `{Supplier, Part} → Quantity` and `{Supplier, Supplier city, Part} → Quantity` – we can see that the only difference is one attribute in the determinant. If we can remove an attribute from the determinant and the dependency still holds, the dependency is left reducible. The first of the two FDs, though, would cease to be true if either `Supplier` or `Part` were removed from the determinant. This FD is left irreducible.

Left Irreducible. If `A` and `B` are subsets of the attributes of a relation, with `A={A1, A2,…, An}`. If `A→B`, the functional dependency is left irreducible if and only if there exists no proper subset of `A, A' (A'⊂A)` such that `A'→B`.

Activity

Copy out Figure 5.42, adding some more functional dependencies of your own if you can. Which are trivial? Which are left irreducible? How many non-trivial, left irreducible FDs can you find?

From the definition of functional dependencies, various properties can be deduced, and more can be deduced from those. The most important and useful are listed below. The first three of these are known as Armstrong's inference rules.

- Reflexivity: if B ⊆ A (B is a subset of or equal to A) then A→B.
- Self-determination: A→A.
- Augmentation: if A→B then AC→BC (adding the same column to both sets does not change the dependency, because of self-determination).
- Transitivity: if A→B and B→C then A→C. A dependency deduced using this rule is called a transitive functional dependency.
- Decomposition: if A→BC then A→B and A→C.
- Union: if A→B and A→C then A→BC.
- Composition: if A→B and C→D then AC→BD.

Activity

Test the attributes out by finding examples of each in the `Contracts` relation.

A graphical representation illustrating FDs for a set of relations is called an FD diagram. An FD diagram consists of boxes linked by arrows. The determinant and the dependent are each enclosed in a separate rectangle, and an arrow connects the two boxes, running from determinant to dependent. For

instance, the non-trivial functional dependencies of the Contracts relation are shown in Figure 5.43.

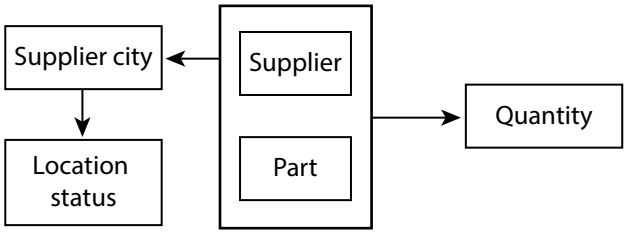


Figure 5.43. An FD diagram showing the non-trivial functional dependencies of `Contracts`.

Functional dependencies cannot be deduced from the structure of the model – they come from the meaning of the data, not its form. This means that knowledge of the real-life system is necessary to identify FDs and normalise the model.

5.5.3 Normal forms

Normal forms provide sets of attributes that guarantee that relations are well structured. These are progressive, with different normal forms having more and more tight requirements. They are defined in terms of functional dependencies, with stricter forms requiring the removal of more types of FDs. If a relation is in a certain normal form, then it will guarantee that it does not contain specific FDs.

In this section we are going to look at four normal forms: 1NF, 2NF, 3NF and BCNF. 2NF guarantees all the requirements of 1NF and then adds to them, meaning that a model that is in 2NF is also in 1NF by definition. Similarly, 3NF adds to the requirements of 2NF, and BCNF adds to 3NF. So a relation in BCNF is, by definition, in 1NF, 2NF and 3NF as well. 2NF, 3NF and BCNF are defined entirely in terms of functional dependencies.

A relation that is not in a given normal form can be decomposed into a set of relations, all of which are in the desired normal form. The decomposition process uses projection operations.

Crucially, the decomposition process for normalisation is 'non loss'; that is, no information is lost as a result of bringing the relation into a normal form.

Non-loss decomposition. A decomposition of a relation `R` into n relations, `R1, R2, …, Rn`, where `n > 1` and all the relations `R1, …, Rn` are projections of R is non-loss if and only if the natural join of `R1, …, Rn` results in the initial relation `R`.

In other words, if we break up a relation into a number of smaller relations, it should be simple to reconstruct the original relation from the pieces. A decomposition that is not a non-loss decomposition, one from which the original cannot be reconstructed, is called a **lossy** decomposition.

As an example, consider the simple relation in Figure 5.44 showing supermarkets and their products. One possible decomposition is shown in Figure 5.45, but is it lossy or non-loss?

Supermarket-Products				
Supermarket	**Location**	**Product**	**Price**	**Stock**
Carrefour	Reims, France	1kg potatoes	€1.95	200
Sainsbury's	Oldbury, UK	1kg potatoes	£1.50	500
Sainsbury's	Oldbury, UK	Baked beans	£0.25	1,000
Sainsbury's	Scunthorpe, UK	Baked beans	£0.25	100
Billa	Stoob, Austria	1kg potatoes	€1.95	35
Tesco	Scunthorpe, UK	1l milk	£0.85	2,000
…	…	…	…	…

Figure 5.44. A relation showing supermarkets and their branches, along with the products they stock.

Products			
Product	**Location**	**Price**	**Stock**
1kg potatoes	Reims, France	€1.95	200
1kg potatoes	Oldbury, UK	£1.50	500
1kg potatoes	Stoob, Austria	€1.95	35
Baked beans	Oldbury, UK	£0.25	1,000
Baked beans	Scunthorpe, UK	£0.25	100
1l milk	Scunthorpe, UK	£0.85	2,000
…	…	…	…

Supermarket-prices	
Supermarket	**Price**
Carrefour	€1.95
Sainsbury's	£1.50
Sainsbury's	£0.25
Billa	€1.95
Tesco	£0.85
…	…

Figure 5.45. A decomposition of the Supermarket-Products relation into two separate relations.

The relations shown in Figure 5.45 make little sense separately, but worse than that is that the original information cannot be reconstructed. Although every value for each tuple in Figure 5.44 has been preserved, with every attribute present at least once, and one in common between the tables for use as a join, there is no projection that could be carried out on the relations to recreate the original relation. Information has been irretrievably lost.

Activity

What is wrong with the lossy decomposition of Figure 5.45? What would the primary keys be of each relation? How do the primary keys relate to the table join needed to put them back together? Try to construct a better decomposition of Figure 5.44 and test whether it can be reconstructed. See whether your version makes intuitive sense.

All of the normal forms (up to and including BCNF) are guaranteed to be achievable for a relation, so for any given relation, there must be a non-loss decomposition that transforms it into a set of equivalent relations all of them in BCNF.

We shall encounter normal forms beyond BCNF in this chapter, but these are defined based on concepts other than functional dependency.

For the sake of simplicity in the explanations so far and to come, we make the assumption that the only candidate key in a relation is the primary key. Clearly, this will not be true in all cases, but it will make the introduction of the forms clearer.

We will refer in the sections below to the `Student-Info` relation (Figure 5.46). This represents information about students, the modules they take and the resulting grades. The primary key is `{Student id, Course id}`.

`Student-Info`							
Student ID	**Student name**	**Student address**	**Course ID**	**Course name**	**Type**	**Value**	**Result**
F122	E. Rocha	UB8	DB1	Databases	2sem	1cu	72%
F122	E. Rocha	UB8	P01	Programming	2sem	1cu	62%
F122	E. Rocha	UB8	SWE	Software Engineering	2sem	1cu	65%
F122	E. Rocha	UB8	FYP	Final Year Project	proj	1cu	NULL
P141	K. Kanou	SE14	DB1	Databases	2sem	1cu	68%
P141	K. Kanou	SE14	IS01	Information Systems	1sem	0.5cu	76%
P141	K. Kanou	SE14	DS	Decision Support	1sem	0.5cu	NULL
F87	K. Shishani	MN6	DB2	Databases	1sem	0.5cu	NULL
…	…	…	…	…	…	…	…

Figure 5.46. The relation to be used for normalisation examples.

First normal form (1NF)

First normal form. A relation is in first normal form (1NF) if and only if all the domains in which its attributes are defined contain scalar values only.

When we introduced the relational model, we described the requirement for all values in a relation to be scalar. Since this is part of the definition of a valid relation, all valid relations are in 1NF. This includes the example given in Figure 5.46.

A relation in 1NF can present all of the three update anomalies discussed in the previous section – insertion anomalies, deletion anomalies and modification anomalies.

Activity

Find at least one example of each anomaly in the `Student-Info` relation.

The `Student-Info` relation includes information about students, the courses offered, and the courses taken by each student. Experience from conceptual modelling might already suggest that three entities have been merged into a single relation. Informally, the solution to the anomalies will be to separate these entities out into separate relations.

Before we define further normal forms, let us consider the functional dependencies of `Student-Info`.

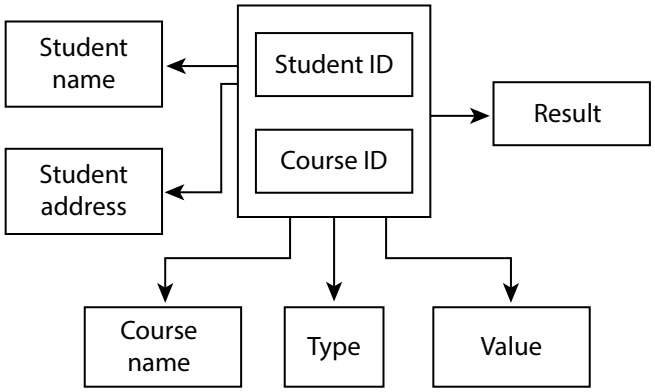


Figure 5.47. An FD diagram showing functional dependencies for the `Student-Info` relation.

Figure 5.47 shows the FDs for Student-Info. The diagram may look confusing, but the most important thing to note is that only one of the functional dependencies has the primary key as its determinant: `{Student ID, Course ID}` → `Result`. All the other FDs have determinants that are either a part of the primary key or a different attribute entirely. It is these FDs that cause the problems, and they must be eliminated to avoid them.

Second normal form (2NF)

Second normal form. A relation is in second normal form (2NF) if and only if:

- It is in 1NF; and
- Every non-key attribute is irreducibly dependent on the primary key.

If we look at the non-key attributes of `Student-Info`, we find that most of the functional dependencies on the primary key **are** reducible.

FD determined by the primary key	Irreducible FD
{Student ID, Course ID} → Student Name	Student ID → Student Name
{Student ID, Course ID} → Student Address	Student ID → Student Address
{Student ID, Course ID} → Course Name	Course ID → Course Name
{Student ID, Course ID} → Type	Course ID → Type
{Student ID, Course ID} → Value	Course ID → Value
{Student ID, Course ID} → Result	{Student ID, Course ID} → Result

Figure 5.48. Of the FDs with the primary key as determinant, only one is irreducible.

Figure 5.48 shows that for each of the functional dependencies having the primary key as a determinant, only one is irreducible. This means that Student-Info is certainly not in 2NF.

To put the relation into 2NF, we must decompose it using projection operations. If our solution is to produce separate relations for students, courses and students' involvement on courses, we must take care to ensure that our decomposition is not lossy.

We have introduced how to recognise 2NF, we have shown how to recognise a non-loss decomposition, but so far we have not considered how to go about making a non-loss decomposition, let alone one that satisfies 2NF.

Heath's theorem gives a useful first hint for how to decompose a given relation without information loss.

Heath's theorem. Given relation R consisting of three sets of attributes, A, B and C and given that the functional dependency A → B applies for R, then R is equal to the join of its projections on {A, B} and {A, C}.

In other words, a relation can be decomposed into two provided that all of the attributes in one of the new relations are functionally dependent on a set of attributes that is present in both new relations. To illustrate why this is helpful, let us take R=Student-Info.

We have seen that Student ID → Student name and Student ID → Student address, which means that we can say (by the union property) that Student ID → {Student name, Student Address}. This means that if we set A=Student ID and B={Student name, Student Address}, then A → B as is required in Heath's theorem. The remaining attributes can be the third set C={Course ID, Course name, Type, Value, Results}.

By Heath's theorem, we can decompose the relation into two relations using projections on {Student ID, Student name, Student address} and {Student ID, Course ID, Type, Value, Results}.

We can go one step further now, taking the second of these new relations as R. We know that Course ID → Course name, Course ID → Type and Course ID → Value, so we can repeat the process. Set A=Course ID and B={Course name, Type, Value} and, since A → B, we can decompose the relation using projections on {Course ID, Course name, Type, Value} and {Student ID, Course ID, Results}.

The relations resulting from these decomposition steps are shown in Figure 5.49 and are guaranteed to be non-loss, thanks to Heath's theorem. By splitting the relations whenever we spot an FD that isn't a primary key, we ensure, in the process, that the result is in 2NF.

Students

Student ID	**Student name**	**Student address**
F122	E. Rocha	UB8
F141	K. Kanou	SE14
F87	K. Shishani	MN6

Student-courses

Student ID	**Course ID**	**Result**
F122	DB1	72%
F122	P01	62%
F122	SWE	65%
F122	FYP	NULL
P141	DB1	68%
P141	IS01	76%
P141	DS	NULL
F87	DB2	NULL

Course

Course ID	**Course name**	**Type**	**Value**
DB1	Databases	2sem	1cu
P01	Programming	2sem	1cu
SWE	Software Engineering	2sem	1cu
FYP	Final Year Project	proj	1cu
IS01	Information Systems	1sem	0.5cu
DS	Decision Support	1sem	0.5cu
DB2	Databases	1sem	0.5cu

Figure 5.49. A non-loss decomposition of Student-Info, resulting in three relations in 2NF.

Usually, there will be more than one possible non-loss decomposition available, but there are some simple rules of thumb to use to help choose which to try.

- Aim for a one-to-one correspondence between the entities in real life (or at least in the system being modelled) and the relations in the model. In general, good normalisation reflects these elements of the real-life system. To do this:
  - do not keep information about two distinct entities in the same relation; and
  - do not keep information about the same entity in a single relation.
- When you have to decompose a relation, choose the decomposition that results in the fewest relations. Do not break things up more than you need to.

By applying these two rules of thumb and Heath's theorem, the decomposition shown in Figure 5.49 can be reached from `Student-Info`. The new relations are much closer to concepts in the system being modelled: `Students`, `Courses` and `Student-courses`. Only the last of these is not immediately intuitive, but is easily explained as a student's course enrolments and results.

Activity

Check that the relations in Figure 5.49 really are in 2NF. Try other decompositions of Student-Info using Heath's theorem. Check each for whether the new relations are in 2NF. If you find a decomposition that produces relations in 2NF, compare it with Figure 5.50 – what are the differences? Which do you prefer, and why?

Third normal form (3NF)

We have noted that the information for normalisation comes from the system being modelled rather than from the structures already in the model. Courses, in our example, have their value entirely determined by their type – a two-semester course (`2sem`) is worth a full course unit (`1cu`); a single-semester course (`1sem`) is worth a half (`0.5cu`), and so on. As we shall see, even though the `Courses` relation in Figure 5.49 is in `2NF`, it is still susceptible to update anomalies because of this.

- Suppose the department agreed to set up a new type of course – a short project, lasting only one semester. It will have a new `Type (semproj)` and be worth half a course unit (`0.5cu`). Until a specific course is offered, there is no way to add this information to the database.
- If `Final Year Project` is the only current course of `Type proj`, then if this module is not offered anymore, then we also lose the information that a `proj` is worth `1cu`.
- If the department makes a decision that all `2sem` will now only be worth `1.5cu`, then every tuple for every course of this type will need to be updated. If this is not performed correctly, then the database will be left in an inconsistent state.

This leaves the relation susceptible to **all** the anomalies that we are trying to avoid.

Activity

Before looking ahead, can you find anything in the functional dependencies of Courses that might explain this situation? Look at Figure 5.47 and try to identify the extra arrow.

The Courses relation is in 2NF, and yet it is still capable of suffering from deletion, insertion and modification anomalies. The reason for this lies in its functional dependencies. Figure 5.50 illustrates the functional dependencies of our new `Courses` relation.

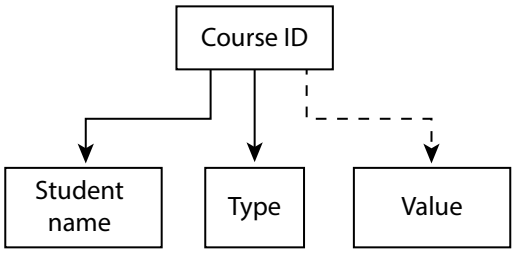


Figure 5.50. Functional dependencies for the Courses relation. The dashed line shows an indirect FD.

The problem is caused by the fact that Value is not directly dependent on the primary key, `Course ID`. The value of a module is solely determined by its type which, in turn, is determined by the `Course ID`. Although it is true that `Course ID` determines `Value`, there is an intermediate step. We can describe this formally:

`Course ID` → `Course name`

`Course ID` → `Type`

`Type` → `Value`

From these (by transitivity), we can deduce that:

`Course ID` → `Type`

Transitive dependency.If `A`, `B` and `C` are non-overlapping sets of attributes in a relation, such that `A` → `B`, `B` → `C`, and `A` → `C`, the dependency `A` → `C` is defined as transitive if and only if the only reason that it is true is because it can be deduced (by transitivity) from `A` → `B`, `B` → `C`.

In other words, if there is a direct causal link in the real world between `B` and `C`, but not between `A` and `C`, then is `A` → `C` transitive dependency. Deciding this requires knowledge of the system being modelled, since the FDs themselves will not look any different.

A transitive, or deduced, FD can be represented in diagrams using a dashed or dotted line. A relation in 2NF can still allow update anomalies if it has non-key attributes that are transitively dependent on the primary key.

This leads to the definition of a third normal form.

Third normal form. A relation is in third normal form (3NF) if and only if it is in 2NF and every non-key attribute is non-transitively dependent on the primary key.

A transitive FD whose determinant is the primary key is caused by an FD between non-key attributes. For example, suppose that `A`, `B` and `C` are disjoint sets of attributes of a relation and `A` is the primary key. The functional dependency `B`→`C` determines the transitive functional dependency `A`→`C`.

On the other hand, if a relation has no FDs between non-key attributes, then it can have no transitive dependencies. That leads to a simpler definition:

A relation is in third normal form (3NF) if and only if it is in 2NF and it has no FDs between non-key attributes.

Applying the same procedure as before, we can use Heath's theorem to decompose the Courses relation into two new relations, as in Figure 5.51.

`Courses`

Course ID	Course name	Type
DB1	Databases	2sem
P01	Programming	2sem
SWE	Software Engineering	2sem
FYP	Final Year Project	proj
IS01	Information Systems	1sem
DS	Decision Support	1sem
DB2	Databases	1sem

`Types`

Type	Value
1sem	0.5cu
2sem	1cu
Proj	1cu

Figure 5.51. A further decomposition results in relations that are in 3NF.

Figure 5.52 shows all the non-trivial FDs for the 3NF decomposition of `Student-Info`. FD diagrams make it easier to check that a relation is in 3NF – if all the existing arrows originate on the primary key and point to non-key attributes, then the relation is in 3NF.

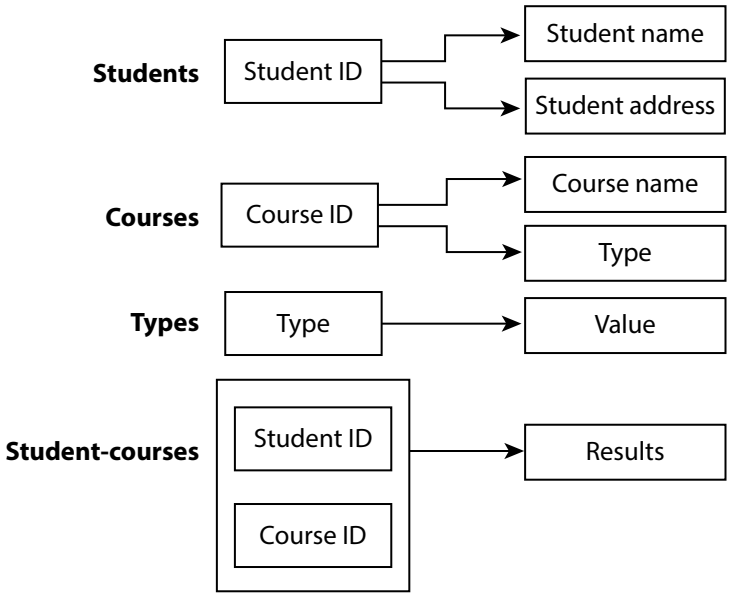


Figure 5.52. FD diagram showing functional dependencies for the relations in 3NF.

We stated earlier that multiple non-loss decompositions are often possible for the same relation. Figure 5.51 shows one possible decomposition of our earlier `Courses` relation. One way to write this solution out would be to describe the relations like this:

```
Courses (Course ID, Course name, Type) ; Types (Type, Value)
```

Another non-loss decomposition that we could have suggested would be this one:

```
Course-info (Course ID, Course name, Type) ; Course-values (Course ID, Value)
```

Although this is certainly non-loss – you can easily reconstruct the original relation – this decomposition still presents an insertion anomaly. It is impossible to represent the fact that a course type has a certain value unless there is a current course running that is of that type.

This problem does not occur with our other solution, so what has gone wrong? Our first solution is clearly better than this one, but what is the difference?

The problem arises because the two `Course-info` and `Course-values` are not independent. Updates to one cannot be made without updating both. For instance, if a type of course changes its value, then all of the courses in `Course-info` that have that type must first be identified, and then all the relevant entries in `Course` values must be updated. This procedure adds complexity and increases the risk of leaving the database in an inconsistent or incorrect state.

However, our previous solution resulted in independent relations that could easily be updated separately. Clearly, then, independence of relations is an important concept, but it is a little tricky to define.

Independent relations. A relation is independent of another if updates to either can be made without considering the other and without a risk of creating inconsistent data with the other.

This definition is a little vague at indicating how to tell that two relations are independent. To get a clearer idea, it helps to return to the FD diagrams. Figure 5.52 shows the clear separation between the relations in our original solution. Every direct FD was represented as an intra-relation constraint – as a constraint within the relation. The transitive relation was represented as an inter-relation constraint between the two new relations.

The new decomposition looks somewhat different. As Figure 5.53 shows, there is a direct FD connecting the two relations that has not been represented, while the constraint that we had previously identified as transitive remains as part of a primary key constraint.

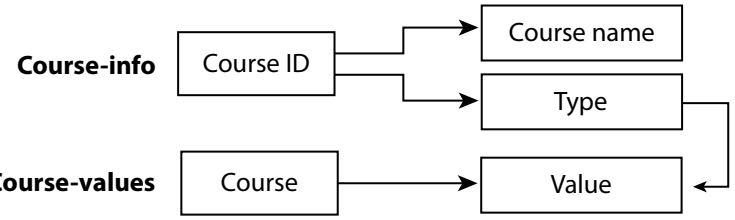


Figure 5.53. The FDs for an alternative decomposition of the Courses relation include a direct FD that connects attributes in two relations. Neither attribute is a primary key.

This indicates a strategy to use – always choose, where possible, the decomposition that consists of independent relations. This principle is known as dependency preservation.

This also gives us another definition for independent relations: a set of relations representing the decomposition of a relation are independent if every direct FD of the original relation R is represented as an intra-relation constraint. Date gives a more rigorous definition in his discussion of this issue.

Boyce-Codd normal form

So far, we have made the restrictive assumption that every relation has exactly one candidate key, but that assumption is not always valid.

Suppose we added a constraint to our `Courses` relation, requiring that there should be a one-to-one correspondence between a course's ID and its title. This requirement would force course titles to be unique – so the two database modules would have to be renamed as, say, `Introduction to Databases` and `Advanced Databases`. The candidate keys for the relation would now be `{Course ID, Type}` and `{Course Name, Type}`, and the FD diagram would be as shown in Figure 5.54.

This again makes the relation susceptible to update anomalies arising from the functional dependence between `Course ID` and `Course name`. Since we required that there should be only one candidate key, there is a problem. The Boyce-Codd normal form is designed to resolve these situations.

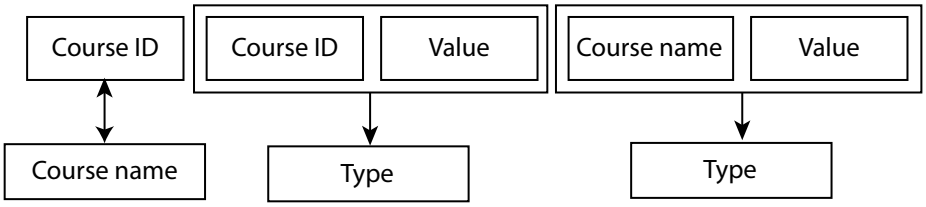


Figure 5.54. FDs for the Courses relation if `Course ID` and `Course name` have a 1:1 correspondence.

The Boyce-Codd normal form (BCNF) can be defined as follows: a relation is in Boyce-Codd normal form if and only if every non-trivial, left irreducible dependency has a candidate key as its determinant. The definition looks simpler, though, if we assume that whenever we use the term functional dependency, we always mean a non-trivial, left irreducible functional dependency (since the others are generally uninteresting).

Boyce-Codd normal form. A relation is in Boyce-Codd normal form (BCNF) if and only if every functional dependency has a candidate key as its determinant.

This definition dictates that all arrows in an FD diagram should start in candidate keys.

Boyce-Codd normal form has a slightly tighter requirement than 3NF (it is sometimes called 3.5NF), and it is always achievable. However, unlike the numbered forms so far, achieving it can bring fresh problems.

Consider the relation Treatments, consisting of patients, diseases and doctors. We make the following constraints, or **semantic assumptions**:

- a patient may have more than one type of disease
- for any given type of disease, a patient is treated by exactly one doctor
- each doctor treats exactly one type of disease, and
- several doctors may specialise in treating the same type of disease.

The FDs of this relation are presented in Figure 5.55.

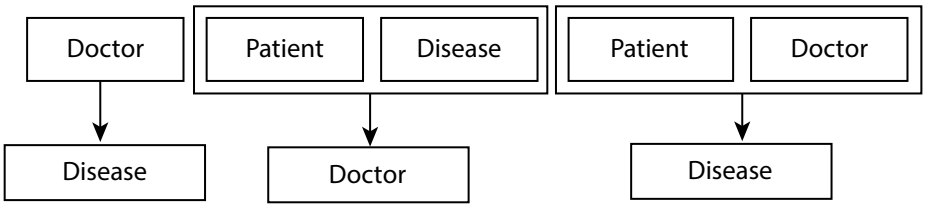


Figure 5.55. Functional dependency diagram for treatments.

This relation has two candidate keys: {Patient, Disease} and {Patient, Doctor}. Because Doctor → Disease, Treatments is not in BCNF.

The only possible non-loss decomposition of Treatments is:

```
{(Patient, Doctor), (Doctor, Disease)}
```

The resulting relations are BCNF, and the decompositions are non-loss. The problem is that the functional dependency {Patient, Doctor} → Disease has been lost in the process. Date has a similar example, which interested readers should explore.

The problem arises because two requirements are in tension – it is not always possible to produce both a non-loss decomposition into BCNF relations **and** have those relations independent. This is a pragmatic issue, and the database designer must choose what is the most important constraint.

Functional dependencies are constraints in the real world, and normalisation can be seen as a way of declaring those constraints and implementing them as part of a relational model. 1NF, 2NF, 3NF and BCNF are always achievable and almost always desirable. They are almost always enough. However in the next section, we will consider further normalisation steps.

Before we do, however, there is an elegant definition for 3NF that is applicable to relations with any number of candidate keys (and so without the restrictions that we have placed on our definitions), and which you may prefer. The definition was laid out by Carlo Zaniolo in a 1982 paper (listed in the Further reading section at the head of this chapter) proposing a new normal form.

Let R be a relation, X be any set of attributes of R and A be any single attribute of R. R is in 3NF if and only if, for every FD $X \rightarrow A$ in R, at least one of the following statements is true:

- `X` contains `A` (so $X \rightarrow A$ is trivial)
- `X` contains a candidate key of `R`
- `A` is contained in a candidate key of `R`.

Zaniolo also defines BCNF by removing the last option above.

5.5.4 Further normalisation – 4NF and 5NF

The normal forms up to BCNF are based on functional dependencies. Since functional dependencies are semantic, real-world constraints, normalising a relation captures extra real-world concepts in the model.

Higher normal forms extend this to embed real-world constraints beyond functional dependencies into our relational model. The mechanism for doing this remains the same – at each stage, relations are decomposed using projection until they satisfy the form.

Fourth normal form (4NF)

Functional dependencies are not the only cause of update anomalies, and it is possible for a relation to be in BCNF and still suffer from them.

As an example, consider the table Tuition in Figure 5.56. The information here is about courses, and the tutors and course books associated with them. There is no link between tutors and books other than that the tutor may teach a course on which that book is assigned.

`Tuition (table)`		
Course	**Tutors**	**Books**
Databases	M. Taylor	Introduction to Databases
	A. Rai	Database Design
Programming languages	A. Clarke	Pascal and Procedural Programming
	E. Montale	Prolog
	G. Greene	Practical Common Lisp
…	…	…

Figure 5.56. The Tuition table – as yet un-normalised.

Since cells in this table are not scalar (single-valued), the table is not a relation, nor is it in any normal form. Turning it into a relation results in the Tuition relation shown in Figure 5.57. This relation is, in fact, in BCNF already, but there are some interesting problems involved in updating it.

Tuition		
Course	**Tutors**	**Books**
Databases	M. Taylor	Introduction to Databases
Databases	M. Taylor	Database Design
Databases	A. Rai	Introduction to Databases
Databases	A. Rai	Database Design
Programming languages	A. Clarke	Pascal and Procedural Programming
Programming languages	A. Clarke	Prolog
Programming languages	A. Clarke	Practical Common Lisp
Programming languages	E. Montale	Pascal and Procedural Programming
Programming languages	E. Montale	Prolog
Programming languages	E. Montale	Practical Common Lisp
Programming languages	G. Greene	Pascal and Procedural Programming
Programming languages	G. Greene	Prolog
Programming languages	G. Greene	Practical Common Lisp
…	…	…

Figure 5.57. Tuition, converted to a relation, is already in BCNF.

It will be obvious just from looking at it that the relation is larger than we might expect – a lot of information is repeated. It also shows update anomalies. For example, if a new tutor, W. Dulu was assigned to the Databases course, two tuples must be added to the relation: `{Databases, W. Dulu, Introduction to Databases}` and `{Databases, W. Dulu, Database Design}`. As in previous examples, a larger relation, perhaps with a reading list of tens or hundreds of texts, would require many more excess updates and involve a higher risk of error.

The anomalies that Tuition presents are not caused by functional dependencies – there are no FDs here. Instead, what we have is a multi-valued dependency. A multi-valued dependency (sometimes abbreviated to MVD) is a generalised version of a functional dependency – an FD is a special case of an MVD.

Where an `FD A` $\rightarrow$ `B` indicates that for every value of `A`, there is a single corresponding value of `B`, an MVD from `A` to `B` indicates that for every value of `A` there is a well-defined set of values for `B`. For example, a given value for `Course` dictates that the `Books` attribute will take one of two values. There is a similar multi-valued dependency from `Course` to `Tutors`.

Multi-valued dependency. Given a relation `R` and three arbitrary, disjoint sets of attributes of `R – A`, `B` and `C` – there exists a multi-valued dependency (MVD) from `A` to `B` if and only if the set of `B` values matching an (`A, C`) pair depends only on the values of `A`.

A multi-valued dependency from `A` to `B` can also be described as `B` being multi-dependent on `A`. It can also be written as `A` $\twoheadrightarrow$ `B`.

Functional dependencies are that particular case of multiple-valued dependency, for which the set of B values is only a single value. MVDs have broadly similar properties and concepts to FDs, for example:

- If R is a relation and A, B and C are disjoint sets of attributes such that $A \cup B \cup C$ represents all the attributes of R, then $A \twoheadrightarrow B$ implies that $A \twoheadrightarrow C$
- An MVD $A \twoheadrightarrow B$ in relation R is trivial if:
  - $B \subset A$ (B *is a subset of* A)*; or*
  - $A \cup B = R$ (there are no other fields).

With these preliminaries, we can now define the fourth normal form:

Fourth normal form. A relation R is in fourth normal form (4NF) if and only if for any existing multi-valued dependency $A \twoheadrightarrow B$ (where A and B are subsets of the attributes of R), A is a candidate key of R – so all the attributes of R are functionally dependent on A.

To decompose a relation for fourth normal form, there is a more general version of Heath's theorem, called Fagin's theorem:

Fagin's Theorem. Given relation R consisting of three disjoint sets of attributes, A, B and C, then `R = JOIN ({A, B},{A, C})` if and only if $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$.

Using this theorem, we can decompose the Tuition relation into Course-Tutors and Course-Books, as shown in Figure 5.58. Both new relations are in 4NF.

Tutors	
Course	**Tutor**
Databases	M. Taylor
Databases	A. Rai
Languages	A. Clarke
Languages	E. Montale
Languages	G. Greene
…	…

Books	
Course	**Book**
Databases	Introduction to Databases
Databases	Database Design
Languages	Pascal and Procedural Programming
Languages	Prolog
Languages	Practical Common Lisp
…	…

Figure 5.58. Tuition **decomposes into** Tutors **and** Books**, both in 4NF.**

Fifth normal form (5NF)

It has been implied so far that any relation can be non-loss decomposed into two relations. This is not always possible. For example, consider the relation Courses given in Figure 5.59. This details courses, tutors and course levels. Tutors can teach certain topics and at certain levels. If a tutor can teach at Levels 2 and 3, then for each topic they teach, they can tutor on courses at those levels. Not every topic has courses at every level, so the tutor will teach at the levels at which courses are offered.

Tuition		
Course	**Tutor**	**Level**
Languages	G. Greene	Level 3
Languages	E. Montale	Level 2
Databases	G. Greene	Level 2
Languages	G. Greene	Level 2

Figure 5.59. This relation cannot be non-loss decomposed into only two relations.

The three possible two-attribute projections are shown in Figure 5.60. Although we can join together any pair of these relations and get a relation with all the attributes, the resulting content will not be the same as the original.

CT	
Course	**Tutor**
Languages	G. Greene
Languages	E. Montale
Databases	G. Greene

TL	
Tutor	**Level**
G. Greene	Level 3
E. Montale	Level 2
G. Greene	Level 2

CL	
Course	**Level**
Languages	Level 3
Languages	Level 2
Databases	Level 2

Figure 5.60. Three two-attribute relations can be created from Tuition, but no two of them would be enough to reconstruct the original relation.

In this case, the only non-loss decomposition creates three rather than two projections.

The logic that forces this to be the case is that tutor T teaches subject S at level L if and only if T teaches S and S is taught at level L and T teaches at level L. This is a particular example of a join dependency.

Join dependency. For a relation `R`, with arbitrary sets of attributes `A, B, ..., Z`. `R` satisfies the join dependency (JD) if and only if `R` is equal to the join of its projection on `A, B, ..., Z`. This can be denoted as * (`A, B, ..., Z`)

A join dependency means that a relation can have non-loss decomposition, even if that necessarily results in more than two relations.

If the join dependency would not have existed were it not for any of the attributes in a candidate key, then we say that the join dependency is **implied** by the candidate keys.

This now gives us the definition for 5NF.

Fifth normal form. A relation is in fifth normal form (5NF) if and only if all its join dependencies are implied by its candidate keys.

Consider the `MusicTeachers` relation in Figure 5.61 below, describing music schools, the instruments on which they offer tuition and the teachers who teach them. The relation is in 4NF, and there is no way to decompose it into two relations without losing information.

`MusicTeachers`		
MusicSchool	**Instrument**	**Teacher**
Steinway Music School	Piano	F. Hensel
Brahms Conservatoire	Violin	S. Geyer
Brahms Conservatoire	Violin	I. Menges
Brahms Conservatoire	Clarinet	H. Littleton
Bolton School	Piano	I. Bolton
Bolton School	Clarinet	I. Bolton
Bolton School	Violin	I. Bolton

Figure 5.61. The `MusicTeachers` relation is in 4NF, but still has update anomalies.

The information carried by this table takes many forms. For example, it indicates which teachers play each instrument, which school each teacher works for, and in which instruments each school offers lessons. If I. Bolton takes a year off from teaching for a concert tour, since the music school no

longer offers his lessons, those rows should be deleted, but if that happens, the relation no longer records the instruments he can teach (for when he returns).

Activity

Find as many other update anomalies in the `MusicTeachers` relation as you can.

The only candidate key for this relation is {`MusicSchool, Instrument, Teacher`}, but the range of information that the relation provides is a hint that there is more structure in the information than this would suggest. The anomalies arise because there is a join dependency that is not implied by the candidate key:

```
*({MusicSchool, Instrument}, {MusicSchool, Teacher},
{Instrument, Teacher})
```

The only way to remove the problem is to decompose the relation into three relations, each in 5NF, as shown in Figure 5.62.

`MusicSchoolLessons`

MusicSchool	**Instrument**
Steinway Music School	Piano
Brahms Conservatoire	Violin
Brahms Conservatoire	Clarinet
Bolton School	Piano
Bolton School	Clarinet
Bolton School	Violin

`MusicSchoolTeachers`

MusicSchool	**Teacher**
Steinway Music School	F. Hensel
Brahms Conservatoire	S. Geyer
Brahms Conservatoire	I. Menges
Brahms Conservatoire	H. Littleton
Bolton School	I. Bolton

`InstrumentTeachers`

Instrument	**Teacher**
Piano	F. Hensel
Piano	I. Bolton
Violin	S. Geyer
Violin	I. Menges
Violin	I. Bolton
Clarinet	H. Littleton
Clarinet	I. Bolton

Figure 5.62. A decomposition of the `MusicSchools` relation into three 5NF relations.

After the 3-decomposition, we are left with all relations in 5NF. 5NF eliminates all update anomalies that can be resolved by projection. It is always achievable. Like 4NF, most tables that are in BCNF are already in 5NF. This is because the conditions that cause the anomalies 5NF resolves are not common.

1NF, 2NF, 3NF, BCNF, 4NF and 5NF are not the only normal forms – others have been proposed to solve other anomalies and embody other semantic concepts in the tables, some using other decomposition methods. It is left to interested readers to pursue these further – they are not required for this course.

5.6 Overview of the chapter

In this chapter, we described the process of designing a database system given a real-life system that is to be modelled. We did this by first creating an Entity/Relationship model, then adapting that model to be compatible with the relational model, and then making a relational model. Finally, we showed how the resulting structure can be normalised to reduce the risk of various anomalies in the database.

5.7 Reminder of learning outcomes – concepts

Having completed this chapter, the Essential readings and activities, you should be able to carry out the logical design of a relational database system. More specifically, this means that you should be able to:

- Devise an E/R model for a real-life system:
  - Identify:
    - entities of the system and their corresponding type
    - attributes for each entity and their corresponding type
    - relationships between entities in the model
    - constraints on the relationships in the model
    - type hierarchies between entities.
  - Eliminate flaws of the E/R model.
  - Draw the model in an E/R diagram.
- Transform an E/R model into a relational model.
  - Eliminate the E/R structures that are incompatible with the relational model.
  - Transform the new E/R model into a relational model.
- Normalise a relational model:
  - Identify:
    - possible update anomalies of a relation
    - functio0nal dependencies of a relation
    - multiple dependencies of a relation
    - join dependencies of a relation.
  - Check whether a relation is in a given normal form.
  - Perform a non-loss decomposition of a relation into a set of relations of a higher normal form.

5.8 Reminder of learning outcomes – key terms

Having completed this chapter and the Essential readings and activities, you should understand the following terms:

- Cardinality constraints (one-to-one, one-to-many, many-to-one, many-to-many)
- Development process: analysis, design, implementation, testing, maintenance
- Entity attributes: simple, composite, single-valued, multi-valued, base, derived
- Entity, entity instance, entity type
- Entity/Relationship (E/R) model
- Fan trap, chasm trap
- Functional dependency:
  - trivial functional dependency
  - transitive dependency
  - multi-valued dependency
  - join dependency

    - determinant
    - dependent
    - left irreducible
- Independent relation
- Logical design
- Non-loss decomposition
- Normalisation, normal forms (1NF, 2NF, 3NF, BCNF, 4NF, 5NF)
- Participation constraints (total and partial)
- Physical design
- Relationship, relationship instance, relationship type
- Semantic design (conceptual modelling)
- Supertype, Subtype, disjoint subtypes
- Top-down approach
- Update anomaly: insertion anomaly, deletion anomaly, modification anomaly
- Weak entity type, strong entity type.

5.9 Test your knowledge and understanding

5.9.1 Sample examination questions

a. Normalise the following table, for an online florist, to the highest normal form you consider appropriate. Show which normal form it is in, and state what assumptions, if any, you are making. [15]

Bouquet name	Bouquet Price	Contents
Valentines	18.00	6 red roses, 3.00 each
Funeral	16.00	4 white lilies, 4.00 each
Basic	12.00	2 white roses, 3.00 each
		2 yellow freesias, 2.00 each
		1 green eucalyptus, 1.00 each
Special	25.00	4 red roses, 3.00 each
		4 white roses, 3.00 each
		2 yellow freesias, 2.00 each
		2 yellow lilies, 4.00 each
		1 green eucalyptus, 1.00 each
…	…	…

b. Produce an E/R diagram based on the following system description.

'Each applicant will send in an electronic copy of their CV and covering letter, which will be stored on the system. Using our web form, applicants also provide their address, date of birth, telephone number and email address. They also give company name, and dates for all previous employers, including a contact name, address and telephone number for each. Finally, candidates will list all their qualifications, from GCSE upwards, indicating the type of qualification, subject, grade and date.' [10]

Notes

Appendix 1: Sample answers/ Marking scheme

Chapter 2: Databases – basic concepts

Question a.

i. The conceptual level represents the logical structure of a database [1]. It specifies the information that is stored, but not how it is stored [1].

ii. The external level consists of the separate views of the database of its various users [1]. It may conceal irrelevant information and store derived information that would be regarded as redundant in the conceptual or physical level [1]. The external/conceptual mapping governs the relationship between the external and the conceptual level [1]. The conceptual level should have all the information and structures needed for the external level to derive its views [1]. [Max 3 marks]

iii. There are two mappings [external/conceptual and conceptual/physical]. [1]

Question b.

Physical independence is the ability for higher-level functionality to behave in the same way [1], even if the physical representation of the data changes [1]. Logical independence is the ability for functionality to behave in the same way [1], even if the conceptual model changes [1]. Logical independence is usually harder to achieve because the conceptual model is often very close to the model that is exposed as the external model [1]. Changing it relies on abstraction that may not be present [1]. On the other hand, physical models should be abstracted away by database management software [1]. [Max 5 marks]

Question c.

i. By a 'file-based approach', she means that separate user software [1] generates data in the form of files [1]. The files may or may not share their format between applications [1]. Directory structure and shared disk space may both be used for organisation purposes [1]. [Max 2 marks]

ii. Answering yes is probably better than no in this context, but credit can be given for either answer here, as long as the reasons make sense; namely: Yes: reduction of repeated information [1]; and labour (reduced redundancy) [1]; reduced risk of inconsistencies in the data [1]; better sharing of data [1]; independence of data from applications [1]; better control of data quality [1] [Max 5 marks] Or No: There is a considerable upfront cost in changing the system [1]; risks of new IT project [1]; impact of a single computer failure may be more wide-ranging [1]; possibility of slower speeds for all users [1].

iii. A database is a shared [1] collection of logically related, persistent data [1] – including its description [1] – as part of an information system [1]. [Max 2]

Question d.

i. Physical data independence is seen when functionality is not dependent on the way in which data is physically represented and stored in the system [1]. This requires some degree of abstraction [1] which means that at least one more abstract **model** of the information must exist between the physical layer and the users [1]. Interaction informed by that model is then independent of representation [1] [Max 3 marks for these]. Yes, the statement is True. [1]

ii. The relational model. [1] [Credit also given for E/R model.]

Chapter 3: The relational model and relational RDBMSs

Question a.

i. At least two options: [2]

Head of state	**Spouse**
Benjamin Henry Sheares, President of Singapore	Yeo She Geok Sheares
Margaret of Austria, Governor of the Habsburg Netherlands	Prince John
	Philbert II
Margaret of Austria, Governor of the Habsburg Netherlands	Philbert II
Henry VIII, King of England	Catherine of Aragon
Henry VIII, King of England	Anne Boleyn
Henry VIII, King of England	…

or:

Head of state	**Title**	**Spouse**
Benjamin Henry Sheares	President of Singapore	Yeo She Geok Sheares
Margaret of Austria	Governor of the Habsburg Netherlands	Prince John
Margaret of Austria	Governor of the Habsburg Netherlands	Philbert II
Henry VIII	King of England	Catherine of Aragon
Henry VIII	King of England	Anne Boleyn
Henry VIII	King of England	…

ii. 6 (5 is also acceptable). [1]

iii. A candidate key is an attribute or set of attributes [1] that can uniquely identify a row [1]. [Head of State, Spouse] is the only candidate key for the first relation option [2]. Depending on interpretation, the second option certainly has candidate key [Head of State, Title, Spouse], but may have [Head of State, Spouse] [2] [Max 4 marks]

iv. If the first option was taken for i., Head of state is only scalar if the name/title/territory can be taken as a single piece of information for all use cases [1]. If they will be taken separately (e.g. which state the person is head of), then it is not really scalar [1].

If the second option (or some other) was taken, the answer will depend on the columns. Key points are the consideration of whether the information will be taken as a single unit [1] by the database [1] in all use cases. [1] [Max 2 marks]

v. Only 4 is True. [2] [Half a mark each for 1. False, 2. False, 3. False, 4. True.]

vi. See Section 3.8 of Volume 1 of the subject guide.

Question b.

i. No. [1]

ii. NULL represents an absence of information. [2]

iii. Anything where it makes sense for a value to be unknown at some point, e.g. {Name: John Smith, Favourite toothpaste: NULL} [2]

iv. 2 [1 mark each for 1. No, 2. Yes]

Chapter 4: SQL

Question a.

i. `VARCHAR(120)` indicates that the field's type [0.5] is a string [0.5] of at most [0.5] 120 characters [0.5], with the amount of storage allocated depending on the string length [1]. [Max 2]

ii. The `AlbumID` field in `Song` relates the songs to the `Album` table [1] in a m: 1 relationship [1]. This is specified in the `FOREIGN KEY` line. [1]

iii. The natural join operates on fields with the same name [1]. The Name field is present in both tables, but refers to different things [1]. As stated, the query will retrieve only tracks with the same name as their album. [1] [Max 2]

iv. `SELECT * FROM SONG LEFT JOIN ALBUM USING (AlbumID)` [2]

Or rename at least one of the name fields (e.g. `AlbumTitle`, `SongTitle`) [2]

v. ORDER BY DurationInSeconds; [2]

Question b.

i. No [1]. A view is a named query [1], which means that it can be referred to again without specifying the query again [1]. It may also have its results stored [1]. [Max 3, 0 if Yes at the beginning, no matter what.]

ii. Views allow for different ways of showing the same data [1]; they provide a mechanism for separating different users' concerns [1]; and for controlling which sensitive information is seen by which users [1]. They can also be used as shortcuts for commonly-used commands [1]. [Max 2]

Question c.

i. `UNION` [1] – returns rows contained in either result [1]. `INTERSECTION` [1] – returns rows contained in both results [1]. `EXCEPT` [1] – returns rows contained in one result or the other, but not both [1].

ii.
```
(SELECT * FROM BALLOONS WHERE Shape="Round")

UNION

(SELECT * FROM BALLOONS WHERE COLOUR="Red");
```

iii. [At this stage, any sensible answer is ok, but we would be looking for:] The non-set version (WHERE/OR) is likely to be better because (depending on the DBMS optimiser): a) it carries out one query, rather than two; b) it carries out all the restrictions in one go; and c) it does not require the generation of intermediate tables. [2] [Related issues will be considered in more detail in Volume 2 of the **CO2209** subject guide.]

Chapter 5: Designing relational database systems

Question a.

Clearly, to be in 1NF, the Contents field must be split. Rows here do not look scalar. We assume, based on the information here that: a) bouquet price is the sum of the individual flower prices (and so can be calculated); b) the price of flowers is based on the type, not the colour. Different assumptions are possible and would produce different results.

`Bouquet contents`

Name	**Flower Quantity**	**Flower Type**	**Flower Colour**
Valentines	6	Rose	Red
Funeral	4	Lily	White
Basic	2	Rose	White
Basic	2	Freesia	Yellow
Basic	1	Eucalyptus	Green
Special	4	Rose	Red
Special	4	Rose	White
Special	2	Freesia	Yellow
Special	2	Lily	Yellow
Special	1	Eucalyptus	Green

Primary key: (Name, Flower Type, Flower Colour)

Foreign key: (Flower Type) References Flowers

`Flowers`

Flower Type	**UnitPrice**
Rose	3.00
Freesia	2.00
Lily	4.00
Eucalyptus	1.00

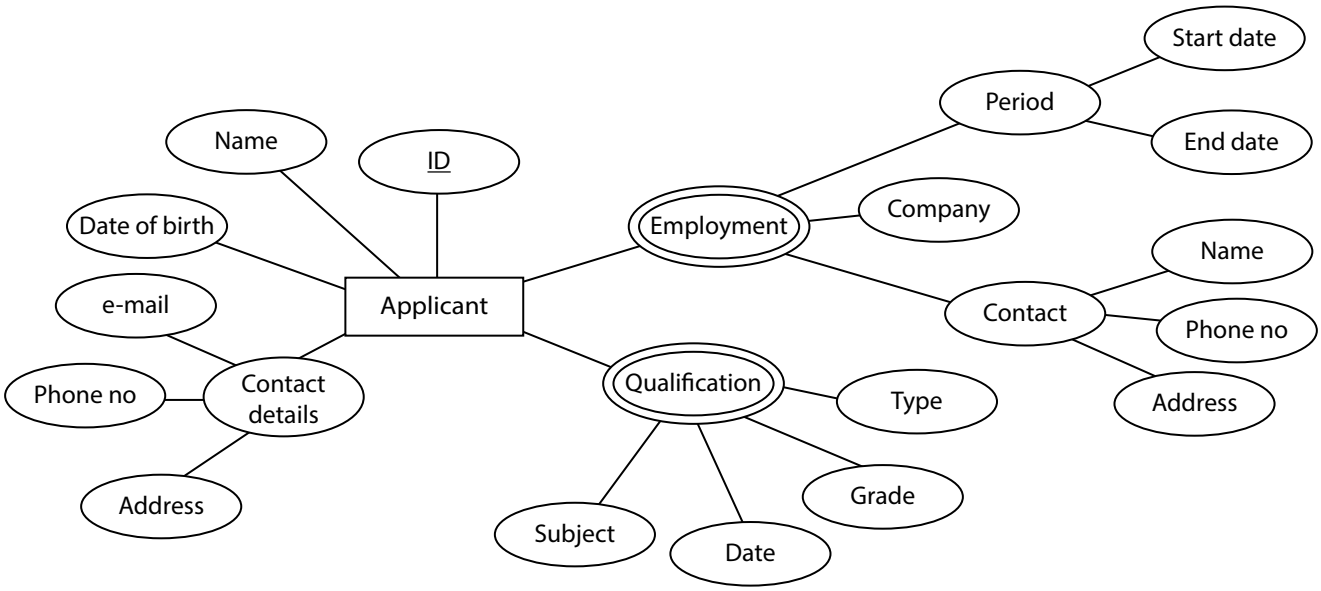
Primary key: Flower Type

Given the assumptions above, and the further observation that, although the flower type will inform the colours available, the relationship is not a straightforward multi-value dependency; these relations are in 5NF.

The Bouquet Price column in the original table would be calculated in a view, using an aggregation function.

Question b.

There are many ways to achieve this. For the one used below, only attributes are used (other than the applicant), but weak entities are also possible. We also introduce an ID field as primary key, because Name/Date of birth might not be unique; one alternative would be to use Name/Date of birth/Contact details.


Start date
Period
End date
Name
ID
Company
Date of birth
Employment
Name
e-mail
Applicant
Contact
Phone no
Phone no
Contact details
Qualification
Type
Address
Grade
Address
Subject
Date