UNIVERSITY OF LONDON | INTERNATIONAL PROGRAMMES

# Database systems: Volume 1

D. Lewis

**CO2209**

**2016**

Undergraduate study in
**Computing and related programmes**

This is an extract from a subject guide for an undergraduate course offered as part of the University of London International Programmes in Computing and related programmes. It contains sample material from Volume 1 of the subject guide, plus the table of contents for Volume 2. Materials for these programmes are developed by academics at Goldsmiths.
For more information, see: www.londoninternational.ac.uk

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London International Programmes 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.

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

www.londoninternational.ac.uk

Published by: University of London

© University of London 2016

The University of London asserts copyright over all material in this subject guide except where otherwise indicated. All rights reserved. No part of this work may be reproduced in any form, or by any means, without permission in writing from the publisher. We make every effort to respect copyright. If you think we have inadvertently used your copyright material, please let us know.

# Contents

# Contents

## Notes

# 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 | <u>£40.00</u> | 18 |
| <u>Gel Lyte VI</u> | <u>Asics</u> | <u>46</u> | <u>MFash3</u> | <u>£100.00</u> | <u>14</u> |
| … | … | … | … | … | … |

**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.

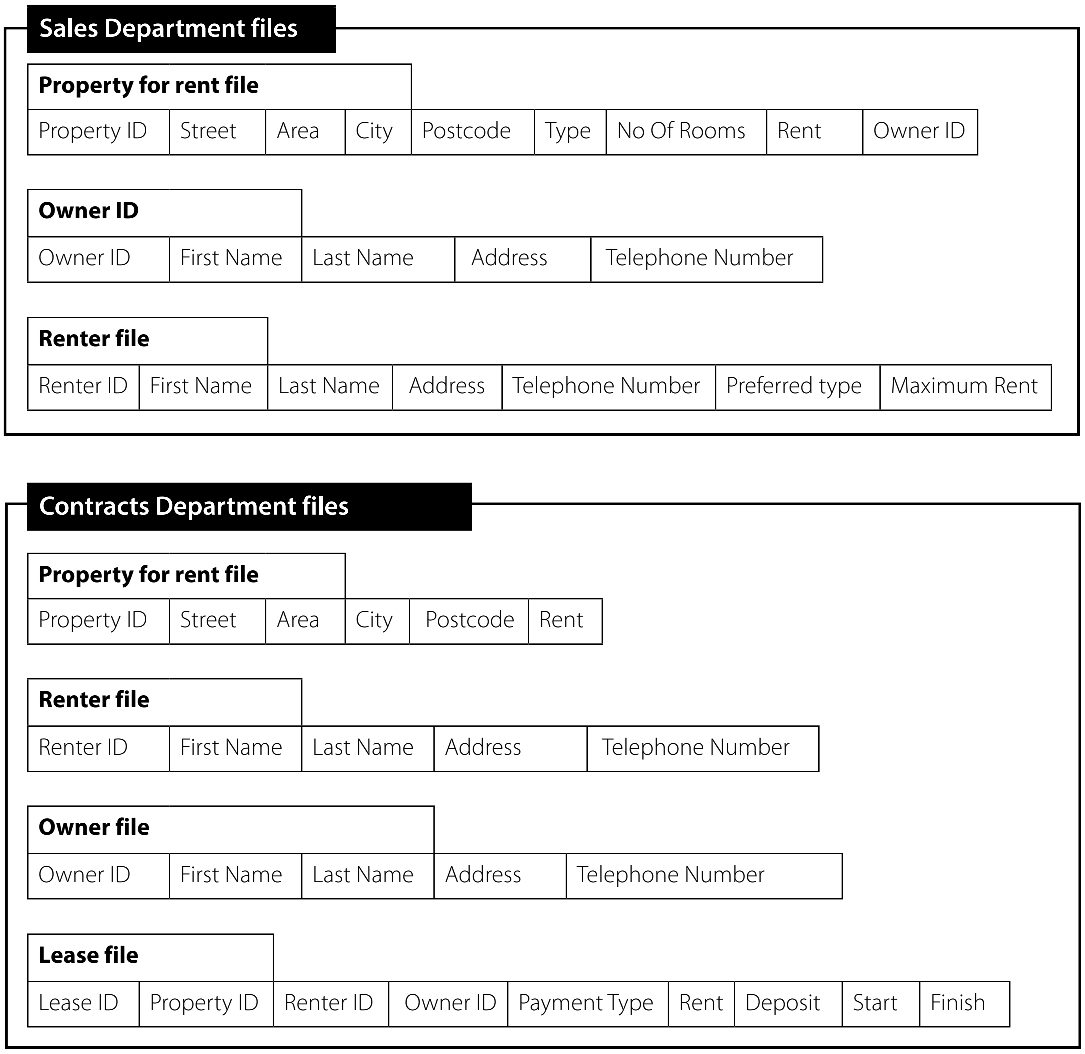

Sales Department files
Property for rent file
Property ID | Street | Area | City | Postcode | Type | No Of Rooms | Rent | Owner ID
Owner ID
Owner ID | First Name | Last Name | Address | Telephone Number
Renter file
Renter ID | First Name | Last Name | Address | Telephone Number | Preferred type | Maximum Rent
Contracts Department files
Property for rent file
Property ID | Street | Area | City | Postcode | Rent
Renter file
Renter ID | First Name | Last Name | Address | Telephone Number
Owner file
Owner ID | First Name | Last Name | Address | Telephone Number
Lease file
Lease ID | Property ID | Renter ID | Owner ID | Payment Type | Rent | Deposit | Start | Finish

**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.

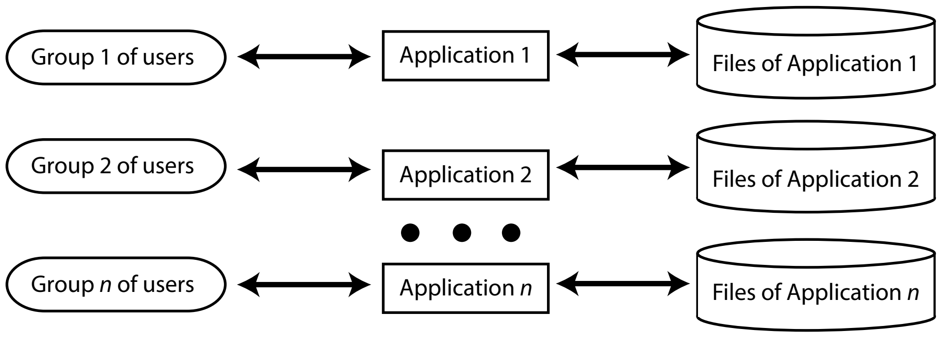


**Figure 2.4. The file-based approach.**

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.

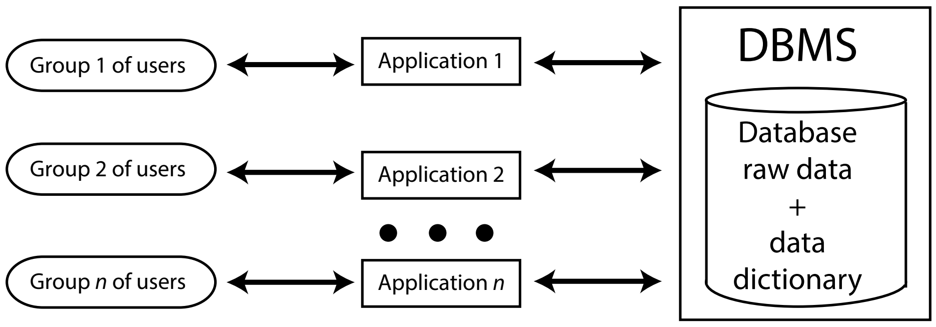


**Figure 2.5. The database approach.**

**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.

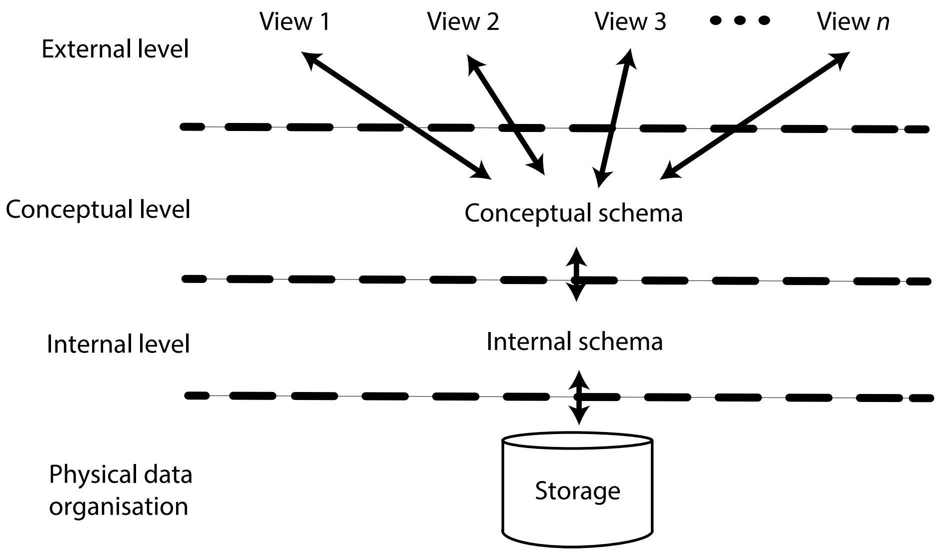


**Figure 2.6. The ANSI/SPARC three-level architecture.**

**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).

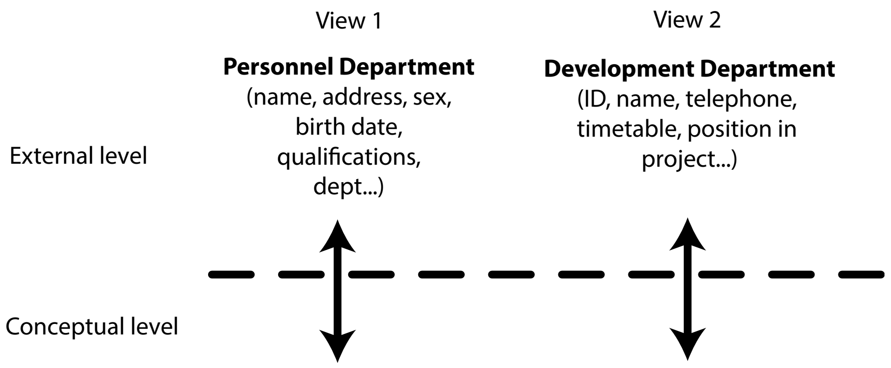

View 1
View 2
Personnel Department
(name, address, sex, birth date, qualifications, dept...)
Development Department
(ID, name, telephone, timetable, position in project...)
External level
Conceptual level

**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:

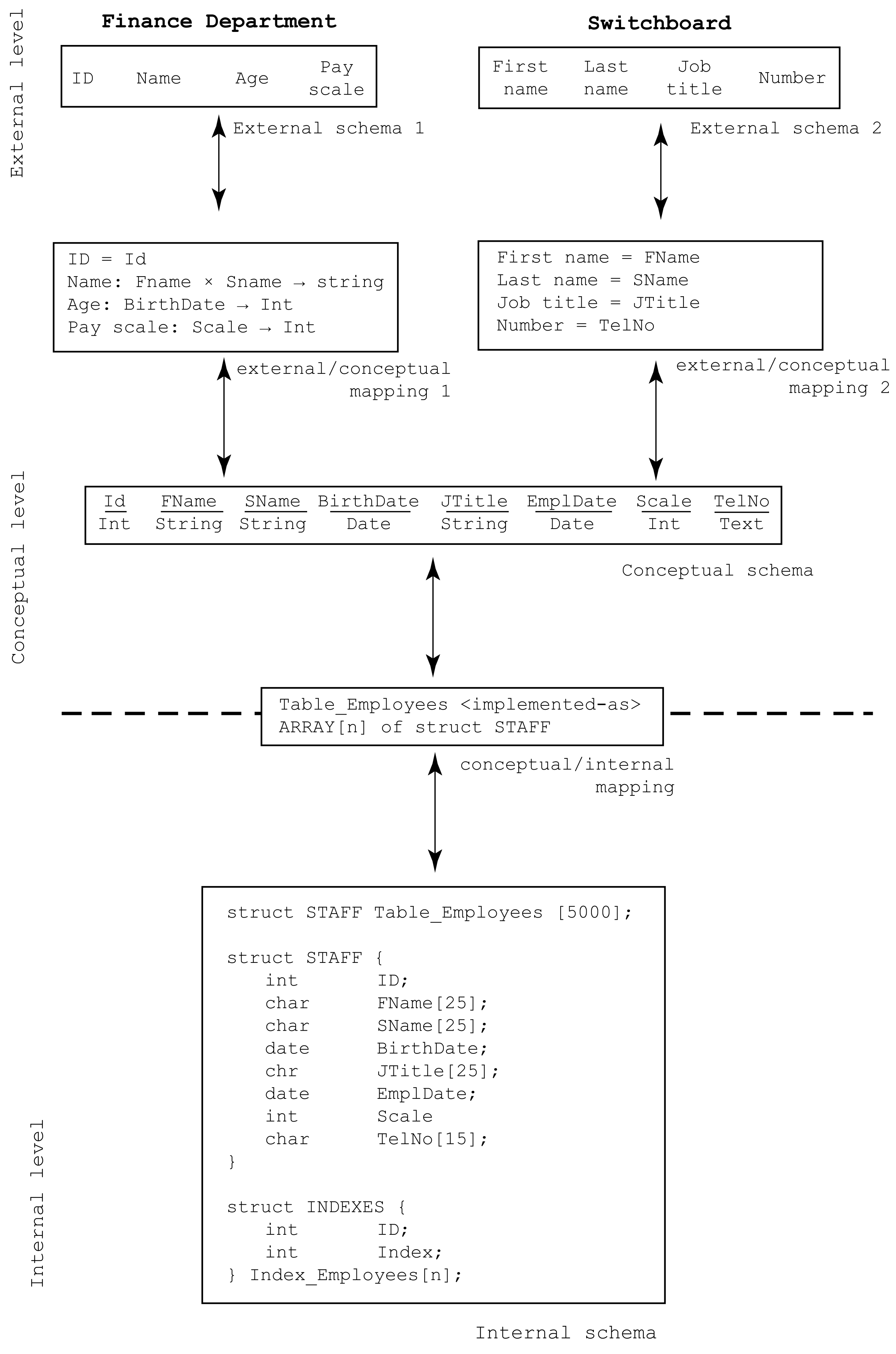
- **External/conceptual** defines the correspondence between an external schema and the conceptual schema.
- **Conceptual/internal** defines the correspondence between the conceptual and the internal level.

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.



```
struct STAFF Table_Employees [5000];

struct STAFF {
   int       ID;
   char      FName[25];
   char      SName[25];
   date      BirthDate;
   chr       JTitle[25];
   date      EmplDate;
   int       Scale
   char      TelNo[15];
}

struct INDEXES {
   int       ID;
   int       Index;
} Index_Employees[n];
```

**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

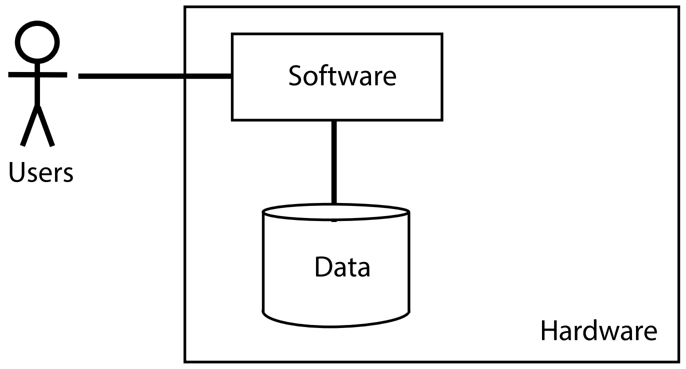


**Figure 2.10. The components of a database system.**

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.

*[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.*

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

### 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.

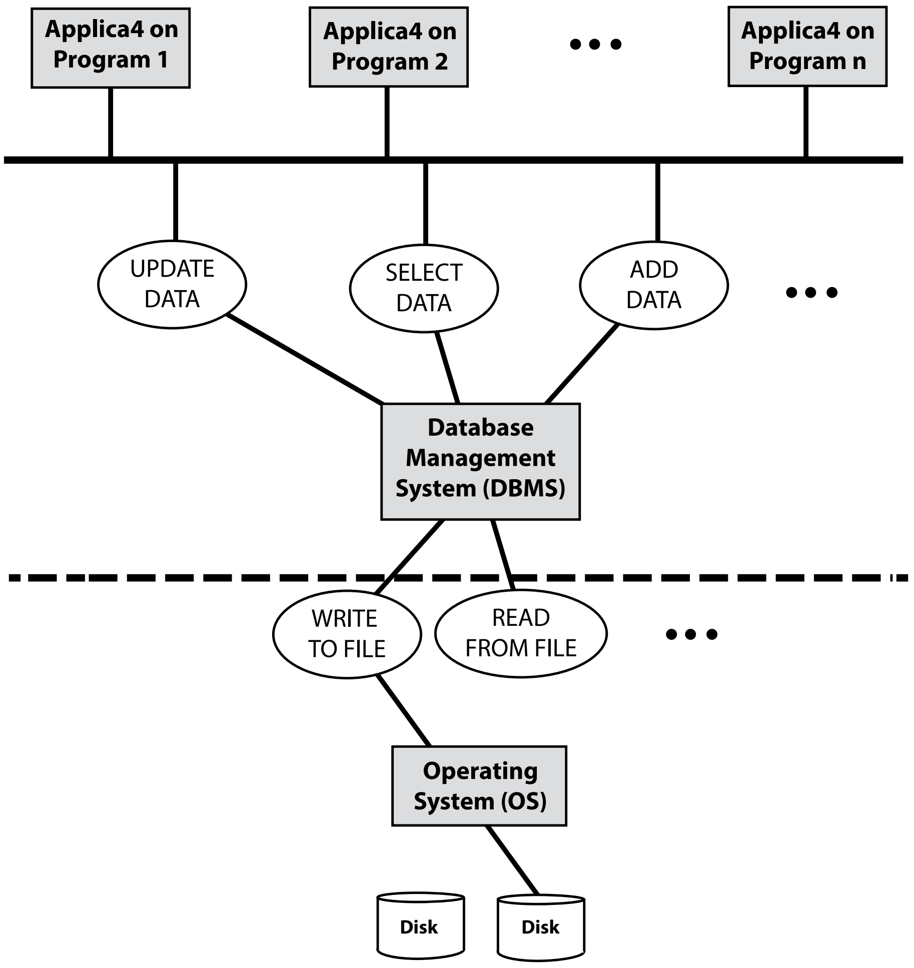


**Figure 2.11. The layered structure of the software component of a database system. Anything below the dashed grey line is platform-dependent, and so will not be discussed here in any detail.**

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.

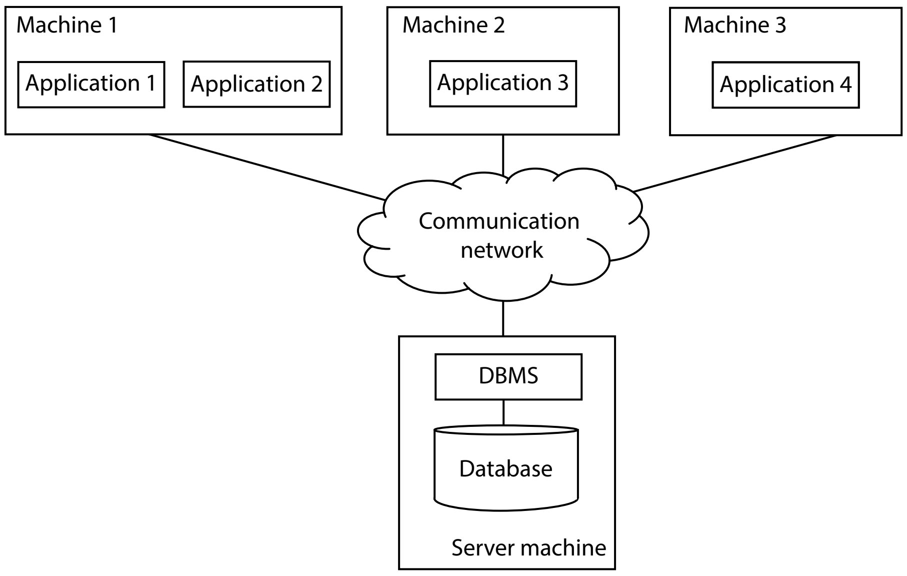


**Figure 2.12. The client-server architecture.**

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).

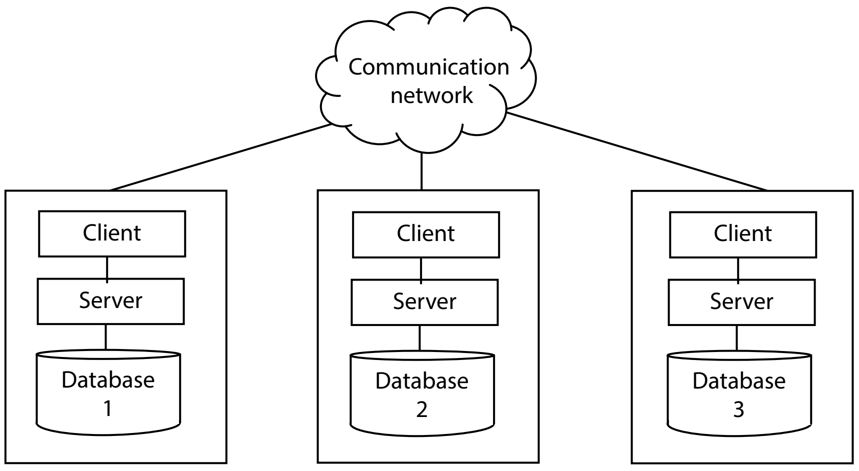


**Figure 2.13. A distributed database system architecture.**

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.

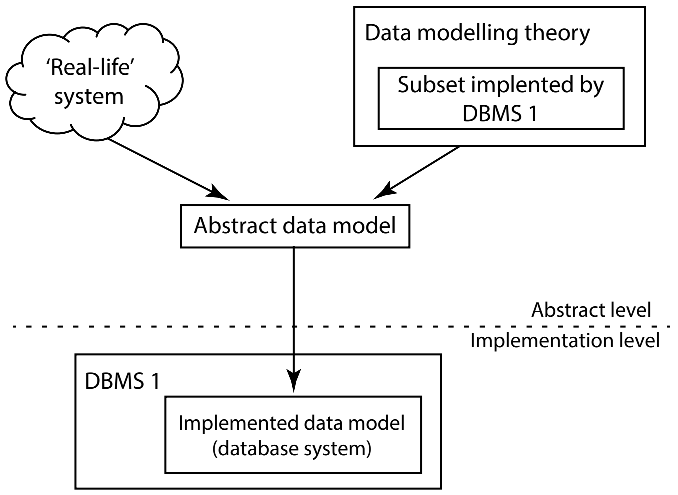


**Figure 2.14. Data modelling and data development.**

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.

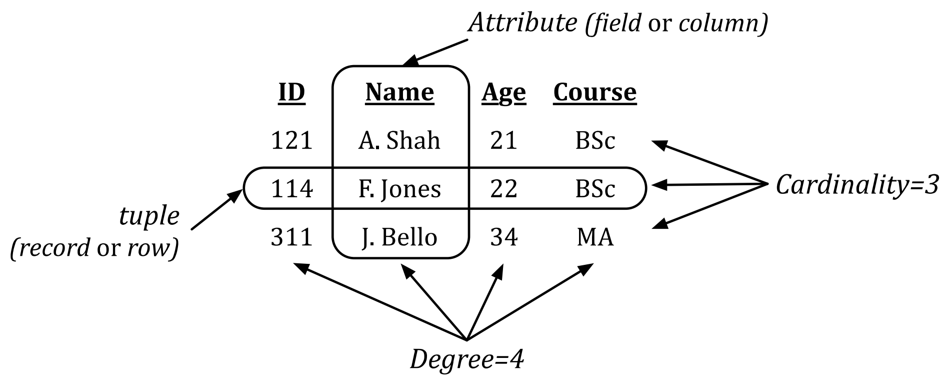


**Figure 3.5. Core terminology applied to the `Students` relation. The whole table represents a *relation*, with rows as tuples and columns as attributes.**

### 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 ith 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

| Student |
|---|
| BC24Z |
| AX007 |

Passed-Courses ÷ Honours-Requirements

| Student |
|---|
| 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

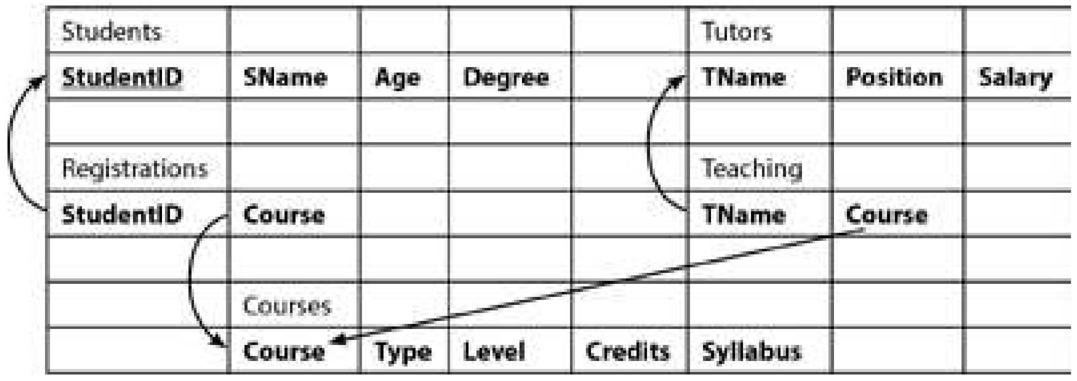


**Figure 3.28. The set of relations for the examples in this section.**

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.

*[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.*

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.

| 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.

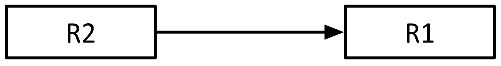


**Figure 3.32. A simple referential diagram. R2 references R1.**

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.

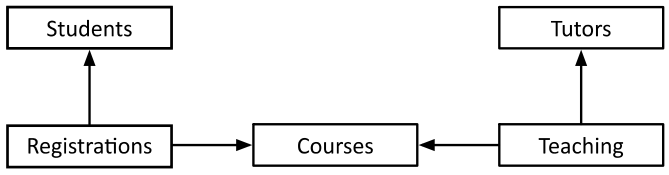


**Figure 3.33. A more complex referential diagram.**

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<br>Philbert II |
| Henry VIII, King of England | Catherine of Aragon<br>Anne Boleyn<br>… |

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