UNIVERSITY OF LONDON

Database systems: Volume 2

D. Lewis

CO2209

2016

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

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

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

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

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

london.ac.uk

Published by: University of London

Contents

Notes

Chapter 1: Introduction to the subject guide

Introduction to Volume 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 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 subject 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 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

- Antoniou, G. and F. van Harmelen *A Semantic Web Primer.* (MIT Press, 2012) [ISBN 9780262018289].
- Bertino, E., B. Catania and G.P. Zarri *Intelligent database systems*. (Addison-Wesley, 2000) [ISBN 9780201877366 (pbk)].
- Chodorow, K. *MongoDB: the definitive guide.* (O'Reilly Media, 2014) 2nd edition [ISBN 9781449344689].

- Elmasri, R. and S.B. Navathe *Fundamentals of database systems*. (Pearson, 2016) 7th edition [ISBN 9781292097619 (pbk)].
- Harold, E.R. and W. Scott Means *XML in a nutshell*. (O'Reilly Media, 2004) [ISBN 9780596007645].
- Özsu, M.T. and P. Valduriez *Principles of distributed database systems*. (Springer, 2011) 3rd edition [ISBN 9781441988348 (pbk)].
- Stevens, P. and R. Pooley *Using UML: software engineering with objects and components*. (Addison-Wesley, 2006) 2nd edition [ISBN 9780321269676].
- White, T. *Hadoop: the definitive guide.* (O'Reilly Media, 2012) 3rd edition [ISBN 9781449311520].

Articles

- Chang et al. 'Bigtable: A Distributed Storage System for Structured Data', *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, 2006; http://research.google.com/archive/bigtable-osdi06.pdf
- Comer, D. 'The Ubiquitous B-Tree', *Computing Surveys*, 11(2) 1979, pp.121–37.
- Dean, J. and S. Ghemawat 'MapReduce: simplified data processing on large clusters', *Proceedings of the 6th Symposium on Operating Systems Design and Implementation – Volume 6 (OSDI'04), Vol. 6.* USENIX Association, Berkeley, CA, 2004; http://research.google.com/archive/mapreduce-osdi04.pdf
- Gray, J. 'The Transaction Concept: Virtues and Limitations', presented at the Seventh International Conference on Very Large Databases, 1981; http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf

Websites

- www.dbdebunk.com
- https://highlyscalable.wordpress.com

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 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 papers is probably the single best way to prepare in the weeks before the examination. You are strongly advised to use 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.

Chapter 2: Data preservation, security and database optimisation

2.1 Introduction

This chapter is concerned with some of the practical and theoretical issues that arise from using databases in real-life situations – issues that have not been discussed in the previous volume of the subject guide. We start by considering the effect of interactions with multiple users on managing a database, both in terms of the safety of the database information and in terms of security and privacy. After this, we will look briefly at questions of the speed of query execution, as well as methods for optimisation.

2.1.1 Aims of the chapter

The aims of this chapter are to give you an overview of the main issues in the use of real-life databases. Starting with data protection and recovery, we move on to consider concurrency control, data security, concluding with database optimisation.

2.1.2 Learning outcomes

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

- explain the concept of the transaction and the mechanisms (COMMIT and ROLLBACK) that implement it
- explain how transaction recovery is achieved in a DBMS and how it is implemented in at least one SQL dialect
- explain how database recovery can be achieved through the transaction mechanism
- explain the two-phase commit protocol and its use in a distributed database system
- discuss the problems arising from concurrent access and the concept of serialisability
- define and explain the locking mechanism, including the concept of deadlocks and how they may be resolved
- discuss and give examples of issues of data security, including the support SQL provides for security control
- discuss the concepts of indexing and denormalisation in the context of database optimisation.

2.1.3 Essential reading

- Date, C.J. *An introduction to database systems.* (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Part IV (same in earlier editions).

  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)], Part 4 (1999 edition), Chapters 16 and 17 or Part 5 (later editions), Chapters 20 and 22.

2.1.4 Further reading

- Comer, D. 'The Ubiquitous B-Tree', *Computing Surveys*, 11(2) 1979, pp.121–37.
- Connolly and Begg, Chapters 9, 18 and Appendix B (1999 edition) or Chapters 18, 19 and 23 (later editions).
- Date, Chapters 16 and 17 (1999 edition) or Chapters 17 and 18 (2004 edition).
- Elmasri, R. and S.B. Navathe *Fundamentals of database systems*. (Pearson, 2016) 7th edition [ISBN 9781292097619 (pbk)], Part 9.
- Gray, J. 'The Transaction Concept: Virtues and Limitations', presented at the Seventh International Conference on Very Large Databases, 1981; http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf

2.2 Databases and the mechanisms for data protection

A database sits at the core of an organisation's information system – it houses all the relevant information of that organisation. This has two consequences.

- Many users access and manipulate data stored in the database, whether directly through the DBMS or through a client application. Since data manipulation requests can happen at any time, there is a higher risk of data getting corrupted.
- Much of what the organisation does will rely on data from the database – the more parts of an organisation that use it, the more reliant the organisation as a whole will become on it. If the data gets corrupted, the consequences for an organisation can be disastrous.

Clearly, it is vital that the data of an organisation is safely stored, even if access to it is widespread and frequent. The DBMS must guarantee that data cannot be lost or damaged by accident or by bad intent – it must provide **mechanisms for data protection**.

To mitigate the risks posed to a DBMS and its data, we must first consider, in some detail, what those risks are. We can illustrate some of the dangers with a few scenarios.

- The DBMS (or a part of it) crashes while executing a set of operations. Because the operations were not successfully completed, the database is left in an unknown, unpredictable and possibly damaged state.
- Two applications, with different users, change the same part of the database at the same time. Because neither is aware of the other, some of the changes made by one application can be overwritten with old data by the other. If the two sets of updates are complicated, the database can easily end up in an incorrect, unpredictable or inconsistent state.
- An unauthorised user gets access to the database, either seeing private information or deliberately altering the database to leave it in a compromised state.

These three scenarios illustrate three important data protection issues.

1. Data recovery.
2. Concurrency control.
3. Security.

We will consider these separately in this chapter. These issues are generic – they affect any data management system that permits multiple users and simultaneous, concurrent interactions – and, as a result, the resolutions of them are also often applicable beyond relational database systems.

Some of the more generic issues – especially those of recovery and concurrency control – will be revisited when we look at alternative models to the relational database.

2.3 Data recovery

Most people have had a computer system fail at some point, whether because a piece of software crashed, or because power was lost to a machine due to loss of battery power, for example. At that point, it is possible that files that were being worked on at the time are lost and unrecoverable. It may not be a disaster, however – perhaps the user saved their work just before the trouble started. More helpfully, perhaps the operating system or the software that was being used at the time had automatically saved versions of all current files and could retrieve them when the software was restarted.

This situation illustrates some common aspects of data recovery.

- Data recovery requires duplication of data – data redundancy – so that there are backup copies of any given piece of information at all times. This redundant data must be stored on non-volatile storage – 'permanent support' – so that it can survive the host computer going through a power cycle.
- The more robust the data recovery features used, the less efficient the system becomes. For example, if a word processor automatically saves a document every few seconds or on each keypress, that requires processor and disk overhead. Depending on the capabilities of the host system, that may cause a noticeable, even annoying, lag.

A system crash that occurs during the use of a database system can lead to the loss of important data and to inconsistencies in the data that is left behind. The ability to recover a correct, consistent state is vital.

Data recovery. Data recovery of a database is the ability to restore its content and functionality following a system failure to an earlier and correct state.

Database systems must be able to handle large amounts of complex data and many concurrent interactions. That means that the data recovery mechanisms that they use are necessarily more elaborate and sophisticated than a simple automatic save function. Still, they are based on similar principles – most importantly the requirement for redundancy.

The database must maintain not only its current content, but also information about a state that is known to be correct that can be used in the case where a system crash results in damage to its main resource. This redundancy has to be on non-volatile storage, or it would be lost in the crash as well.

Ideally, the user should be unaware of the details of the process – if a system crash occurs, the DBMS should be able to restore its database to a correct state without an administrator having to be involved. Counterintuitively perhaps, this means that a DBMS and the logical structure of the data it stores should appear the same whether data recovery is supported or not. Redundancy occurs at the physical level only, and is not reflected at all at the logical level.

The mechanisms of data recovery are based on the concept of the transaction.

2.3.1 Transactions

Consider a database for a company that keeps information about the programmers they employ and the projects they work on. The information is stored in two tables, illustrated in Figure 2.1 below.

Programmers			
PID	**Name**	**SalaryBand**	**NumberOfProjects**
MB1	M. Baker	22	2
GK1	G. Kovacs	24	1
JL1	J. Lilly	28	3

Assignments		
PID	**Project**	**HoursPerWeek**
MB1	Mobile	24
MB1	UserTesting	11
GK1	UserTesting	30
JL1	Mobile	12
JL1	Parser	6
JL1	UIDesign	12

Figure 2.1. Two tables in a company's database.

Suppose that a new programmer, B. Eich, has been employed and has had three projects assigned to him. The database must be updated accordingly. The SQL insertion operations required to do that would be something like this:

```
INSERT INTO Programmers VALUES (“BE1”, “B. Eich”, 20, 3);
INSERT INTO Assignments VALUES (“BE1”, “Parser”, 12);
INSERT INTO Assignments VALUES (“BE1”, “UIDesign”, 12);
INSERT INTO Assignments VALUES (“BE1”, “Mobile”, 12);
```

If an error occurred after the first two insert operations were performed successfully, but before the others were even started, the database will end up both incorrect – it does not reflect the current state of the system being modelled – and inconsistent – it contains data in one part that contradicts the information in another part. Since the Programmers table indicates that B. Eich has three projects, but the Assignments table only shows one, there is an inconsistency between the tables.

Activity

What effect does the order of operations have on this problem? Is there a way of reordering these commands so that, if they are not all completed, the database is consistent? Is there a different set of insertions that would be more robust? Is there anything about the data model used that makes this example particularly susceptible to inconsistencies?

What this example illustrates is that there are situations where a set of operations only make sense if they are performed together – the performance of only a part of them would make the database incorrect, inconsistent or both. In other words, the operations represent a single logical unit of work.

The problem arises because many logical units of work cannot be expressed in just one SQL statement that is guaranteed either to run as a whole or not be executed at all. The example above can be expressed as four statements as above, or two statements:

```
   INSERT INTO Programmers VALUES (“BE1”, “B. Eich”, 20, 3);
   INSERT INTO Assignments VALUES (“BE1”, “Parser”, 12),
      (“BE1”, “UIDesign”, 12), (“BE1”, “Mobile”, 12);
```

It is impossible, however, to achieve the same result with a single statement, and yet they make no sense separately.

Transaction. A transaction is a sequence of database operations that represent a logical unit of work. The database should be in a consistent state both before they start and after they are completed.

Note that during the performance of a transaction – between its first operation and its last one – the database can, and often will, be in an inconsistent state. It is only between transactions that the database is guaranteed to be in a consistent state (provided the user gives consistent data).

If a system supports transactions, then it should guarantee that all the constituent operations of the transaction will be executed once the transaction is initiated. Of course, if that requirement were achievable – if we could guarantee that every operation that started would also finish – then we would have less need for transactions in the first place. Instead, there is a weaker requirement, the so-called principle of transaction processing support.

Principle of transaction processing support. If some, but not all, operations of a transaction are performed and a failure occurs before the planned termination of the transaction, then the DBMS must guarantee that those operations that have been carried out already will be undone.

This principle provides an all-or-nothing feature that ensures the **atomicity** of a transaction – a transaction cannot be split, at least in terms of its execution, into smaller components. Using transactions, the database can be seen to evolve from one correct state to another with intermediate states concealed, and transitions only carried out using successfully executed transactions.

Implementation. Most modern RDBMS implementations support transactions. However, there may be some restrictions. For example, to use transactions in MySQL, you will need to ensure that your tables use the InnoDB storage engine, not the older MyISAM.

The module of a DBMS that implements transaction support is called the transaction manager. The transaction manager supports two mechanisms:

- COMMIT TRANSACTION – guarantees the performance of all the corresponding operations of the transaction.
- ROLLBACK TRANSACTION – rolls back, or undoes, operations that have been performed within the transaction so far – for when an error occurs.

In pseudocode, a transaction might look something like this:

```
START TRANSACTION;
   INSERT ...
   UPDATE ...
   ...
```

```
IF no problems
   THEN COMMIT TRANSACTION;
   ELSE ROLLBACK TRANSACTION;
```

So the transaction has a single beginning, but two possible ends – either the whole sequence of operations is committed to the database, or everything is undone.

From the example we gave earlier, we could add the programmer and his new assignments more safely by placing the necessary operations inside a transaction (still in pseudocode):

```
START TRANSACTION;
   INSERT INTO Programmers VALUES (“BE1”, “B. Eich”, 20, 3);
   INSERT INTO Assignments VALUES (“BE1”, “Parser”, 12);
   INSERT INTO Assignments VALUES (“BE1”, “UIDesign”, 12);
   INSERT INTO Assignments VALUES (“BE1”, “Mobile”, 12);
IF no problems
   THEN COMMIT TRANSACTION;
   ELSE ROLLBACK TRANSACTION;
```

Crucially, the DBMS keeps a **log** or **journal** of each operation as it carries out these actions, allowing the system to keep track of what has and has not been carried out.

In general, a DBMS is likely to copy some or all of any database into a buffer in volatile memory and perform operations on those copies – largely for reasons of speed. At certain points, these copies are written to non-volatile storage. If a system crashes when only the buffers have been altered by an operation, the update will be lost. Because of this, it is vital that the log of operations is recorded to non-volatile memory until it can be written permanently to disk.

The information recorded in the log allows the DBMS to:

- undo any performed operation if an error occurs before its transaction is completed;
- perform all operations of a transaction on non-volatile storage even if all the operations were previously lost from volatile memory – as would be the case in a system crash.

As we noted earlier, if all the operations of a transaction are correctly performed, with no errors or interruptions, then a COMMIT statement can be issued. Completing a COMMIT statement guarantees that all the operation of the transaction can be performed on non-volatile storage – even if the data is not written directly, the log has all the information necessary.

The moment when a COMMIT statement is issued is called a commit point or sync point. Once a commit point is reached, it is certain that no update operation from the corresponding transaction will have to be undone – the changes made are permanent. Between a commit point and the beginning of the next transaction, the database is available for any retrieve operations. Between a START TRANSACTION statement and a COMMIT or ROLLBACK, the database is likely to be in an incorrect and inconsistent state and some or all operations from outside the transaction are likely to be blocked. Figure 2.2 illustrates the succession of states and the restriction of access during transactions.

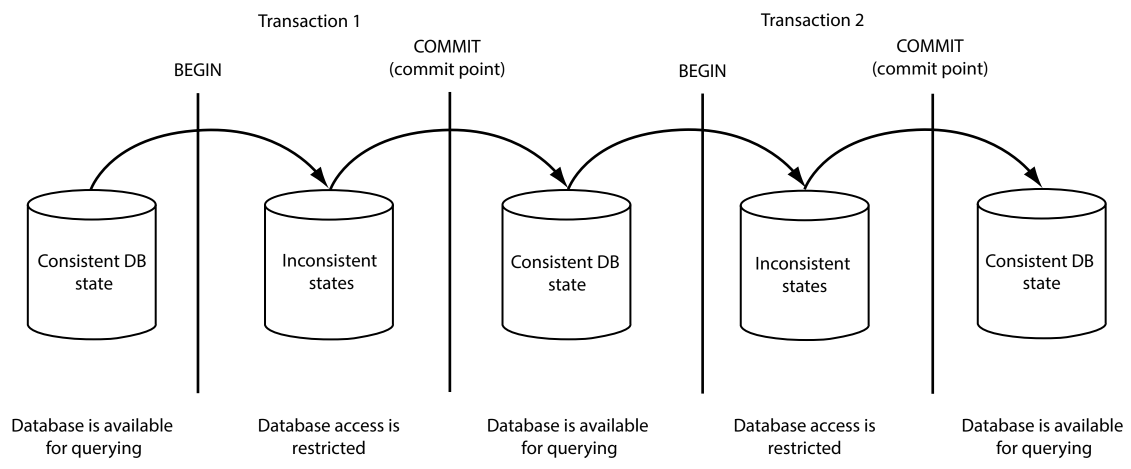


Figure 2.2. Since, during a transaction's execution, the database may be in an inconsistent state, access must be restricted until the transaction has been committed and its operations are complete.

If something goes wrong, the ROLLBACK statement is issued, and the database has to recover its previous state. This mechanism is called transaction recovery and is shown in Figure 2.3. Clearly, if full database access had been available to other users during the aborted transaction, wrong information could have propagated elsewhere.

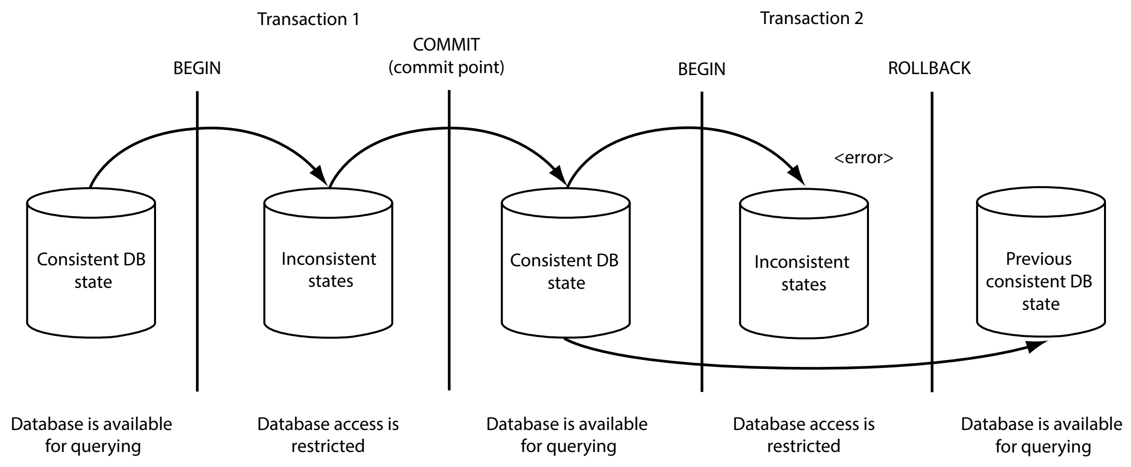

Transaction 1
Transaction 2
BEGIN
COMMIT (commit point)
BEGIN
ROLLBACK
<error>
Consistent DB state
Inconsistent states
Consistent DB state
Inconsistent states
Previous consistent DB state
Database is available for querying
Database access is restricted
Database is available for querying
Database access is restricted
Database is available for querying

Figure 2.3. When Transaction 2 fails for some reason, it triggers a ROLLBACK, and the previous state of the database is restored.

We have said that a transaction is the logical unit of work. From the behaviour described above, it can also be said that the transaction is the unit of **recovery**.

The log plays an important part in this process, but it can only be used in transaction processing if the relevant information about each operation of a transaction is completely entered into the log before the operation starts. This rule is known as the **write-ahead** rule.

We have assumed so far that the constituent operations of a transaction can be considered to be atomic; that is, that they cannot be divided, and that they either succeed or fail in entirety. This is an assumption that is guaranteed valid by lower level mechanisms of the DBMS, and is not regarded as a part of transaction management.

The use of transactions makes it possible for DBMS developers to make certain important guarantees of reliability. These guarantees were formalised over the course of the 1970s and 1980s. One important paper on the subject is Jim Gray's (1981) 'The Transaction Concept: Virtues and Limitations', presented at the Seventh International Conference on Very Large Databases (see Further reading for website address), which laid the groundwork for the form in which these guarantees are best known – the **ACID** properties. The ACID properties are four properties that are central to discussions of data recovery:

1. **Atomicity** – either all operations in a transaction are performed or none of them are.
2. **Consistency** – the transaction mechanism guarantees that the DB evolves from one consistent state to another.
3. **Isolation** – transactions are isolated from one another, so that when a transaction is performed on a database, no other transaction that might affect the same data can be performed at the same time.
4. **Durability** – once a transaction has been committed, the changes it makes are guaranteed to be performed physically (i.e. in non-volatile storage).

These properties inform not only important aspects of database recovery but, as we shall see later, they are a key issue in comparing database models that present alternatives to relational theory.

2.3.2 Database recovery

So far, we have considered a single transaction on a database. If an error occurs during a transaction, the DBMS recovers by undoing the operations that have been performed. If the database is only used by one person at a time, then this is straightforward, but a database is a shared resource. It is likely that a DBMS will have to deal with several transactions at once.

A transaction can fail because of an error, but it can also fail because of a system failure either of hardware or software. In the case of hardware failure, it is possible that the non-volatile memory is itself damaged, which brings additional complications, but for the present, we will consider the case of a 'soft crash', that results in the database system failing, but all storage written to disk remaining accessible afterwards.

If a DBMS is processing several transactions when a failure occurs, all the transactions that were in progress at the time of the crash must be dealt with appropriately. As in the case of a single transaction, it is the information kept in the log that makes this possible.

Much of the normal running of a database happens in in-memory buffers – the DBMS takes a copy of the data it needs and manipulates it in internal memory. At certain points in time (at the **checkpoint**), the system automatically writes ('force writes') all the data that has changed onto the permanent support. At the same time, a list of all the transactions that were in progress at the time of writing is recorded in a log called the checkpoint record. Then the system continues processing data until the next force writing point, and so on.

The implications of a crash on any given transaction depend on its status both at the time of the crash and at the time of the last force write. Figure 2.4 illustrates the five types of transaction states.

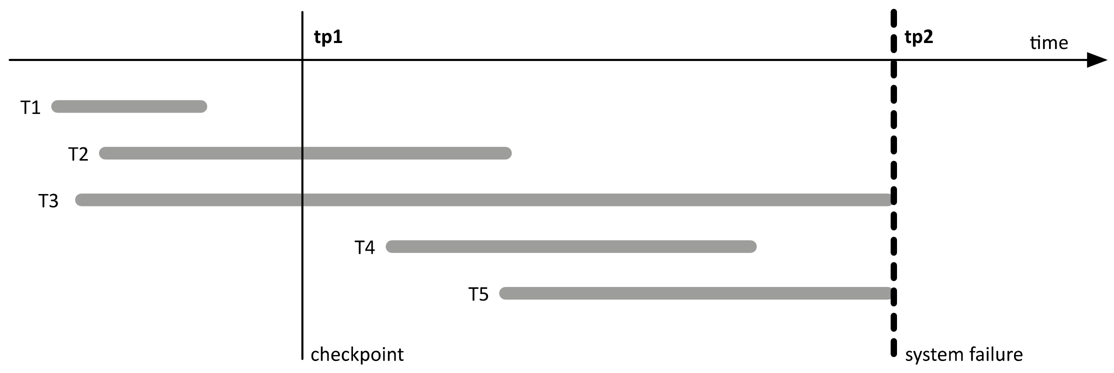


Figure 2.4. A system failure has occurred at time point tp2. The last save was at tp1. This diagram illustrates the different stages of operation of five transactions (T1-5) before the crash.

In this diagram, a failure has occurred at time point tp2, and the most recent checkpoint was at time tp1. The possible situations that a transaction can be in as a result are:

- T1 – Completely successful before tp1. It was written on permanent support at tp1 and its successful completion has been recorded in the log.
- T2 – Initiated before tp1 and completed successfully before tp2. However, it was completed after tp1, and so its successful completion was written to the log, but, we cannot guarantee that all operations performed after tp1 have been written to permanent support, so this operation transaction may be complete in the logs, but not on disk.
- T3 – Initiated before tp1, but not completed at the time of the failure. Only its initiation was written to the log, so the log will be incomplete for this transaction.
- T4 – Initiated after tp1, but completed before tp2, so its successful completion was recorded in the log, but we cannot guarantee that the data has been committed to permanent support.
- T5 – Initiated after tp1 and incomplete at the time of the crash. Only its initiation was written to the log, so the log record for this transaction will be incomplete.

After the system restarts, and before it begins fresh data processing, the DBMS must recover from the failure. The operations that it undertakes are:

- Transactions of type T1 can be disregarded. They have not only completed successfully, but their results have been written on recoverable, non-volatile media.
- Transactions of type T2 and T4 have to be repeated based on the information in the log. This is because they completed successfully, but we cannot guarantee that the results of their operation were saved to permanent support.
- Transactions of type T3 and T5 must be **undone**, because they were not successfully completed at the time of the crash. The log does not contain sufficient information in order for them to be re-initiated.

2.3.3 Transactions in SQL

SQL supports transaction-based recovery using `START TRANSACTION`, `COMMIT` and `ROLLBACK` commands, and these are generally reliably implemented in most implementations.

The standard also provides much more fine-grained control – the type of a transaction can be declared, for example – and extra functionality, such as the ability to create a checkpoint (`SAVEPOINT`) in the middle of a transaction. Support for these extra features may vary, and the best recommendation is to read the documentation for the DBMS that you are currently using.

In the SQL standard, statements such as `INSERT`, `UPDATE` and `SELECT` are called transaction-initiating statements, and if they follow a `COMMIT` or `ROLLBACK`, automatically start a new transaction. We would recommend that this approach is avoided – it is usually clearer to have explicit `START TRANSACTION` commands where the DBMS provides them.

2.3.4 Two-phase commit

We briefly discussed in Chapter 2 of Volume 1 of this subject guide that a database may be split into several resources. Although we will explore this subject in the next chapter in more detail, one aspect of it directly affects transactions, and so needs to be introduced here. Each resource in a distributed database will be managed by its own resource manager, each on a different machine.

If a transaction involves operations that need to be carried out on multiple resources, it may involve more than one resource manager. If, for example, a database is split, based on the departments within an organisation, all that is required is a transaction that modifies data from multiple departments. To keep the Atomicity requirement, the DBMS must ensure that all resource managers perform their updates locally or that none of them do. If one department updates its data and another does not – perhaps because that department's computer crashes – then we are left with an inconsistent database.

The presence of multiple resource managers distributed across a network adds a new dimension to the problem of recovery. The DBMS has to implement global `COMMIT` and `ROLLBACK` statements. When a global `COMMIT` is issued, the DBMS guarantees that all the operations are committed, even in the case of a crash somewhere in the network. If any single resource manager fails, then the global transaction must be rolled back. To achieve this global task, a special system component, the co-ordinator, is needed.

To show how this is carried out, we can first look at the case where a transaction's operations have completed successfully, and the co-ordinator is issued with a global `COMMIT` statement. The actions of the DBMS now are divided into two phases.

- **Phase 1**: The co-ordinator prepares each resource manager that is involved in the transaction by sending a 'get ready' message. On receiving this message, each resource manager must force write in their log the description of all the operations it has performed. Once this information is recorded on non-volatile storage, it will be possible to commit or roll back the operations even if a crash occurs. If the force writing process is completed successfully, the resource manager sends an `OK` to the co-ordinator, otherwise it sends a `NOT OK` message.
- **Phase 2**: Once it has received responses from all the resource managers involved in the transaction, the co-ordinator decides how to proceed. If all

the answers were `OK`, the co-ordinator will decide to commit; if even one response is `NOT OK`, it will choose to roll back. The co-ordinator records its decision in its own log (so that, if there is a crash, this decision will also be recovered) and then sends it to all the participant resource managers. Even if a failure occurs, each resource manager is guaranteed to execute the coordinator's decision, because all the relevant information has been written to permanent support.

The two-phase commit ensures that the atomicity of transactions is preserved across a distributed database, with each resource manager first preparing and then committing their share of the transaction. If an interruption occurs, the entire transaction is either committed or rolled back as a whole.

However, there is an added risk from this approach. If the co-ordinator itself fails while the resource managers are waiting for its final decision, the separate databases may remain in a locked state. Normally, this situation will be resolved when the co-ordinator restarts, but the entire system becomes dependent on restoring this component – something which acts against the aim of distributing the database in the first place.

2.4 Concurrency control

2.4.1 Concurrency problems

Because a database is a shared resource, it must allow multiple users or programmes to have **concurrent access** to data; that is, each must be able to access the database at the same time. When we listed the five possible stages for transactions before a failure, we allowed (in Figure 2.4) five transactions to access the database concurrently.

If we have many transactions occurring at once (that is, they are **interleaved**), there should be no problem if each accesses disjoint parts of the database – if there is no data in common between them. Problems occur when the same data is accessed by two or more transactions at the same time.

The problems caused by concurrent access to the same data can be classified into three particular types.

1. Inconsistent analysis problem.
2. Lost update problem.
3. Uncommitted dependency problem.

Inconsistent analysis. The inconsistent analysis problem is caused by a transaction querying data that another transaction is changing at the same time.

To illustrate this, consider a wholesaler selling party supplies. The wholesaler has three warehouses – A, B and C – that stock balloons, and three fields of the database represent the current stock each carries – BalloonsA, BalloonsB and BalloonsC.

Now, suppose there is a regular stock-checking transaction (Transaction 1) that queries the number of balloons held by each warehouse, keeping a running total as it goes. On its own, this would be an uncomplicated transaction, but suppose a second transaction is executed that records that 200 balloons have been transported from warehouse A to warehouse C. This information can be entered by first reducing the value of BalloonsA by 200 and then increasing it in BalloonsC (Transaction 2). Clearly, the total number of balloons stays the same, we have simply moved some from one place to another. Figure 2.6 illustrates what can happen.

Time	Transaction 1	Transaction 2	Balloons		
			A	B	C
0	START TRANSACTION		200	200	600
1	RETRIEVE BalloonsA [val=200, sum=200]				
2	–	START TRANSACTION			
3	–	UPDATE BalloonsA = BalloonsA – 200	0	200	600
4	–	UPDATE BalloonsC = BalloonsC + 200	0	200	800
5	–	COMMIT			
6	RETRIEVE BalloonsB [val=200, sum=400]				
7	RETRIEVE BalloonsC [val=800, sum=1200]		0	200	800

Figure 2.6. Two concurrent transactions resulting in an inconsistent analysis.

In the example above, there are 1,000 balloons distributed between the three warehouses; this is true both at the beginning and the end of the transactions. Even though, in our example, Transaction 1 does nothing while Transaction 2 is running, and so only operates on the database when it is both correct and consistent, it still produces an incorrect total of 1,200 balloons.

This is because Transaction 2 changes the values of BalloonsA and BalloonsC between Transaction 1 reading the one and the other. Although the state of the database has been kept consistent, the transaction has not, and it has values from two different states of the database.

Lost update. The lost update problem arises when two transactions update the same data, but each fails to take the action of the other into account.

To return to our balloon stocks, consider a transaction that checks the number currently available in a warehouse and, if they are above a certain level, and some other database checks are satisfied, processes an order for 200 and so reduces the stock level appropriately. Figure 2.7 shows what this might look like if two transactions like this run concurrently.

Time	Transaction 1	Transaction 2	BalloonsA
0	START TRANSACTION		300
1	RETRIEVE BalloonsA [val=300]		
2	<other operations>		
3		START TRANSACTION	
4		RETRIEVE BalloonsA [val=300]	
5		<other operations>	
6	UPDATE BalloonsA = 300 – 200		100
7		UPDATE BalloonsA = 300 – 200	100

Figure 2.7. Two transactions; each process orders for 200 balloons at the same time. Since there were only 300 balloons available, these orders cannot be fulfilled.

In the example above, orders totalling 400 balloons have been processed even though the warehouse only has 300 in stock. Furthermore, the database only shows the effect of one of the two transactions. Since the update of Transaction 2 is based on data from before Transaction 1 took place, the change Transaction 1 has made to BalloonsA is ignored and overwritten.

Uncommitted dependency. The uncommitted dependency problem occurs when as-yet-uncommitted changes from one transaction are relied on by another transaction. This causes no problems if the transaction is successfully committed, but if it is rolled back, then the values used by the second transaction may never have been part of a consistent database state.

Figure 2.8 illustrates how this might affect the stock-taking transaction for our balloon supplier when another transaction tries to withdraw stock and fails.

Time	Transaction 1	Transaction 2	Balloons		
			A	B	C
0	START TRANSACTION		200	200	600
1	–	START TRANSACTION			
2	–	UPDATE BalloonsA = BalloonsA – 200			
3	RETRIEVE BalloonsA [val=0, sum=0]	–	0	200	600
4	–	ROLLBACK	200	200	600
5	RETRIEVE BalloonsB [val=200, sum=200]				
6	RETRIEVE BalloonsC [val=600, sum=800]		200	200	600

Figure 2.8. A read-only transaction (Transaction 1) using a value resulting from a transaction that is later rolled back. The result is incorrect.

In the example above, the sum returned by Transaction 1 reflects a database state that was never committed, and so is incorrect. The same problem arises if instead of merely returning a value calculated from incorrect data, Transaction 1 stores it somewhere else in the database. If Transaction 1 is wholly successful and Transaction 2 fails and is rolled back, then the database will be inconsistent.

All these problems arise from concurrent access to the same resource. It is hard to avoid them as long as full access to all data is provided all the time to several interactions at the same time. To avoid the problems, we will need some sort of mechanism that restricts data access at any point where having unrestricted access could introduce inconsistencies.

Clearly, the simplest solution would be to permit one transaction only to be executed at a time. This would be guaranteed to be free of concurrency problems, but would remove an important part of what makes a DBMS useful. Nonetheless, there is one important aspect of this solution – it shows us that any method of carrying out a set of operations that has the same results as this concurrency-free solution is, by definition, free of concurrency problems. Such a method is called serialisable because the result is indistinguishable from a serial, non-concurrent, ordering of the transactions.

Serialisable. An interleaved execution of a set of transactions is serialisable if it produces the same result as some serial execution of the same set of transactions, one at a time.

With this definition established, we can say that **the interleaved execution of a set of transactions is guaranteed to be correct if it is serialisable**.

The way a set of transactions is executed is called a **schedule**. A schedule of a set of transactions specifies the order in which the separate operations in the transactions will be executed. As such, Figures 2.6–2.8 represent schedules, but they are not serialisable, and thus give rise to concurrency issues.

There are many ways to try to achieve serialisable schedules, but by far the most common method is called **locking**.

Activity

Before we move on to discussing locking, consider the problem of serialisation. What sorts of operations can be interleaved safely? What sorts will necessarily interfere with each other? What would a method for ensuring serialisability look like?

2.4.2 Locking

The locking mechanism provides a means for preventing multiple transactions from accessing the same data in a way that might give rise to inconsistencies.

Before a transaction uses the data it refers to, it must acquire permission to use it. Permission may be granted or refused depending on the type of access required and whether the data is currently being used. If permission is granted, then the data will have a **lock** placed on it that marks that it is being used and the type of use occurring. Put more simply, a transaction can use data if it can acquire a lock on it.

There are two basic types of lock – Exclusive or write locks, and Shared or read locks.

Exclusive lock. An exclusive lock is required for any transaction that needs to modify a certain tuple. Once a transaction has acquired an exclusive lock on a tuple, no other transaction can access it – even for reading.

Shared lock. A shared lock is required by any transaction that needs to retrieve data from a tuple, but does not need to modify it. There can be any number of shared locks on a tuple at any given time. However, if a transaction requests an exclusive lock, the request will be refused if there are any shared locks already on the tuples involved.

The table in Figure 2.9 gives a simple summary of how requests are treated given the current lock state of the required data.

		Current lock		
		Exclusive	**Shared**	**None**
Requested lock	**Exclusive**	×	×	✓
	Shared	×	✓	✓

Figure 2.9. Table showing the requirements for acquiring a lock. An exclusive lock is only granted if no other transaction is using the data, while a shared lock is granted provided no exclusive lock is already in place.

It is enough in this subject guide to consider the tuple as the only lockable object, but, in Chapter 16 on concurrency, **Date** (2003) includes a section ('Intent locking') that discusses other kinds of lockable object.

This locking mechanism is accompanied by a data access protocol that describes how it is used. This is summarised below.

- A transaction that only needs to read the value of a tuple must first acquire a shared lock on that tuple.
- A transaction that needs to modify or update a tuple must first acquire an exclusive lock on that tuple.

- If a transaction is refused a lock on a tuple because another transaction has a lock on it already, then it goes into a **wait state** until the lock is released. When more than one transaction is waiting for a lock on the same tuple, a sub-module of the concurrency control module called the scheduler decides on the order in which to serve them. The scheduler should ensure that no transaction waits forever for its lock to be granted.
- All locks are normally held until the transaction they were granted for is completed – a `COMMIT` or `ROLLBACK` statement releases the lock.

The use of locking solves the various concurrency problems listed earlier. For example, Figure 2.10 shows how the uncommitted dependency problem shown in Figure 2.8 would have been resolved.

Time	**Transaction 1**	**Transaction 2**	**Balloons**		
			A	**B**	**C**
0	START TRANSACTION <request shared lock on BalloonsA> <shared lock granted>		200	200	600
1	–	START TRANSACTION <request exclusive lock on BalloonsA> <request denied>			
2	RETRIEVE BalloonsA [val=200, sum=200]	<wait>			
3	RETRIEVE BalloonsB [val=200, sum=400]	<wait>			
4	RETRIEVE BalloonsC [val=600, sum=1000]	<wait>			
5	COMMIT <release Shared lock>				
6		<exclusive lock granted> UPDATE BalloonsA = BalloonsA - 200	0	200	600
7		ROLLBACK	200	200	600

Figure 2.10. Two transactions as in Figure 2.8. This time, through locking, the second transaction is not granted the ability to write to BalloonsA until after Transaction 1 has finished.

Activity

Construct similar tables to resolve the other two problems introduced in Figures 2.6 and 2.7. Does the use of locks like this fix these problems? Can you see any other potential issues?

The data access protocol for locking allows us to resolve some problems, but gives rise to some of its own. In the next section, we consider the problem of deadlocks and the issues that arise from waiting for a lock to be released.

2.4.3 Deadlocks

If we consider the lost update problem, the behaviour without locks was illustrated in Figure 2.7. Figure 2.11 shows the same transactions, only using locks.

Time	Transaction 1	Transaction 2	BalloonsA
0	START TRANSACTION		300
	<Request S lock on BalloonsA> <S lock granted>		
1	RETRIEVE BalloonsA [val=300]		
2	<other operations>		
3		START TRANSACTION	
		<Request S lock on BalloonsA> <S lock granted>	
4	<request X lock on BalloonsA> <X lock denied>	RETRIEVE BalloonsA [val=300]	
5	<wait>	<other operations>	
6	<wait>	<request X lock on BalloonsA> <X lock denied>	
7	<wait>	<wait>	
8	<wait>	<wait>	

Figure 2.11. Two transactions both acquire shared locks (S locks) on the same data. Later, each needs an exclusive lock (X lock), but is denied the request because of the current S lock. Neither will release its S lock until the new lock is granted, causing a deadlock.

The two transactions in this case each require, and obtain, a shared lock on BalloonsA. Each transaction then requests an exclusive lock so that they may modify its value, but the request is denied because, in each case, the other transaction still has a shared lock on the data. Neither transaction will release its shared lock until it is granted the new one (or, more commonly, until the end of the transaction). This results in a situation in which each transaction is blocking the other from being able to progress, causing an unending wait state. This situation is called **deadlock**.

A deadlock can also occur when each of two transactions holds two or more exclusive locks in common, as shown in Figure 2.12.

Time	Transaction 1	Transaction 2
0	<some operations>	<some operations>
1	<request X lock on r1> <X lock granted> <operation on r1>	
2		<request X lock on r2> <X lock granted> <operation on r2>
3	<request X lock on r2> <X lock denied>	
4	<wait>	<request X lock on r1> <X lock denied>
5	<wait>	<wait>

Figure 2.12. A deadlock caused by one transaction operating on r1 and then r2 and another r2 and then r1. Each has a lock that the other needs, but neither will be first to release it.

In summary, with the help of shared and exclusive locks and the data access protocol, we can solve many of the problems of concurrent transactions, but only by introducing a new problem in the form of deadlocks.

Although it is possible to have a deadlock involving more than two transactions, this case is rare in practice.

Activity

Try to construct a situation in which three transactions result in a deadlock.

Since a deadlock will never break on its own, the system must identify such situations and resolve them. The most common way to detect deadlocks involves each system maintaining a directed graph that represents which transaction is waiting for which other ones to complete.

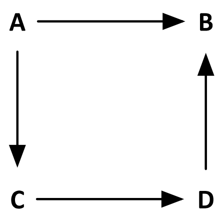


Figure 2.13. Example of a wait-for graph.

Figure 2.13 shows a simple wait-for graph. Transaction A depends on B and C to complete, C is waiting for D which in turn is waiting for B. In this case, the sequence B then D then C then A will complete without any need for intervention.

Figure 2.14 shows a larger wait-for graph. In this case, F is waiting for C, but is one of the two transactions that C is waiting for, causing a deadlock. The same is true for {BED}.

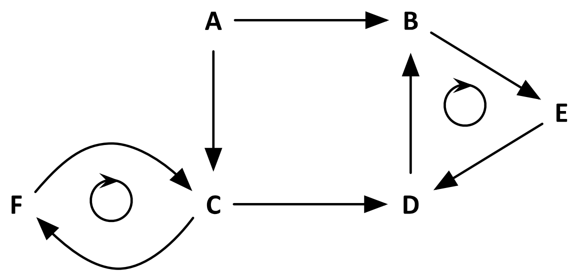


Figure 2.14. A wait-for graph showing two deadlocks (indicated by cyclical arrows).

By checking for cycles in the wait-for graph at regular intervals, the system can identify and resolve deadlocks. Once a deadlock is identified, the system terminates one of the transactions involved by rolling it back. The terminated transaction is called the **victim**. Rolling back the victim results in it releasing the locks that it holds, resolving the deadlock. Usually, the victim is automatically restarted after the deadlock has cleared. How the victim is chosen is outside the scope of this subject guide.

Despite the risks posed by deadlocks and the extra complications of dealing with them, the locking mechanism allows a DBMS to extend the guarantees of transaction execution to all concurrent transaction executions.

2.4.4 Serialisability

The approaches described in the previous section are a way to ensure that a schedule is serialisable, and thus correct.

Time:	t1	t2	t3	t4	t5	t6	t7	t8	t9
Transaction 1	op1-1	op1-2	op1-3	op1-4					
Transaction 2							op2-1	op2-2	op2-3
Transaction 3					op3-1	op3-2			

Figure 2.15. Schedule showing operations of three transactions executed serially: first Transaction 1, then Transaction 3 then Transaction 2.

Time:	t1	t2	t3	t4	t5	t6	t7	t8	t9
Transaction 1	op1-1			op1-2			op1-3		op1-4
Transaction 2		op2-1			op2-2	op2-3			
Transaction 3			op3-1					op3-2	

Figure 2.16. Three interleaved transactions. The important question: is this serialisable?

Figures 2.15 and 2.16 show a serial and an interleaved schedule for the same three transactions. It should be clear now that the outcome of the two will be the same if:

- the operations work on completely different data, making locking irrelevant; or
- all operations are `SELECT` operations, since none of the data will be changed by the concurrent transactions – this is why shared locks do not exclude other shared locks from being granted.

If one item writes a data item and the other reads it, the order does matter. An exclusive lock granted until the end of a transaction guarantees that the result is serialisable. For example, if op1-1 reads the same tuple that op1-3 writes, then requiring a lock on that triple for the remainder of Transaction 1 and Transaction 2 will guarantee that the schedule shown in Figure 2.16 is equivalent to running Transaction 1 first and then Transaction 2 (since that is the order in which the locks will be requested and granted, if these are the only transactions running). A different schedule might be equivalent to running Transaction 2 first, but will still guarantee serialisability.

In fact, a further definition is still needed to be sure that all possible interleaved schedules are correct. We have assumed certain restrictions on how locks are requested and released that we should now make explicit, since they are the final piece required to guarantee correctness of a schedule. These restrictions are codified as the **two-phase locking protocol**.

Two-phase locking protocol. The two-phase locking protocol requires that:

1. a transaction must acquire a lock on any object before accessing it; and
2. after releasing a lock, a transaction must not attempt to acquire other locks.

This is called two-phase locking because these restrictions mean that a transaction can be divided into two phases. During the first phase, as each operation is carried out, locks are either acquired or upgraded (from shared to exclusive). The second phase occurs only once all the locks that will be needed have been acquired, at which point locks may be released.

In almost all RDBMS implementations, the second phase is entered only when all transactions operations have carried out – when the COMMIT or ROLLBACK operation is reached – at which point all the locks acquired by the transaction can be released. This stricter form of the two-phase locking protocol is called **strong strict two-phase locking** (sometimes shortened to SS2PL).

It is important to remember that ensuring that a set of transactions are serialisable is only a way of guaranteeing the correctness of a schedule so that anomalies in the database will be avoided. What it does not do is avoid deadlocks.

2.5 Data security

Since a database is a shared resource where all the relevant information of an institution may be stored, users are usually only allowed to see or modify the parts of it that apply to them.

To use our recurring example, in a university's database, students are likely to be able to see information about the courses offered, about timetables and about themselves, but they are unlikely to be able to modify any of it or see information about other students or staff. Lecturers may be able to see everything that the student can, but also information about their own department and students. They may also be allowed to modify parts of their students' records, such as examination results.

The list will continue with different people – or rather the roles that they take on – receiving the ability to read or modify only the parts of the database that are required by those roles.

To generalise: every DBMS must provide mechanisms that guarantee users can only do the operations for which they have permission. The DBMS must provide data security.

There is no single reason or process to account for all security restrictions. They can come from various sources and have different motivations. For instance:

- **Legal**: the personal information about the employees of a company if stored must be kept confidential under data protection laws.
- **Ethical**: the examination results can legally be made public, but it is usually judged best to show them only to the students concerned, then the students can decide with whom to share them.
- **Strategic**: some universities may choose to share aggregated results or statistics about their staff that they feel reflects well on the organisation. Universities that do less well may share less.

Activity

Choose one of the database application examples that you worked with in Volume 1 of the **CO2209 subject guide**. Identify as many security restrictions that would apply to it. Are there any categories that we have missed out?

The problem of identifying security restrictions is an important one, but outside of the scope of this course. For now, we assume that such security restrictions have been identified and must now be expressed and enforced in a database system.

Since security is of concern well beyond the world of database administrators, the strategies for imposing it are subject to external constraints and specifications that make it accessible to managers and policy makers. Two common approaches to data security reflect this: mandatory and discretionary access control.

Mandatory access control. The access users have to data is defined externally by a policy, and in advance. Most commonly, this involves a simple list of classification levels for users and data, with rules specifying what access is granted to which levels of data by users having a given clearance level. Such systems are inflexible by design – access cannot be reassigned by whim, and control is locked down according to carefully specified policy. This approach is common for military and government use (the word mandatory here relates to legally-binding requirements for secrecy or privacy).

Discretionary access control. In discretionary access control, the access users have to individual data objects can be separately and dynamically controlled. Usually, a user with suitable access privileges can change the permissions of other users.

Discretionary access control is the main approach in DBMS applications; however, many do implement mandatory control, and in certain applications, their use will be a requirement.

Security requirements within mandatory access control systems can have surprising implications for the rules of the system. Imagine a typical system where each user is given a clearance level, with 0 being the lowest and 10 the highest. Documents are also classified with a number between 0 and 10. As one might expect, the standard rule for read access would be to allow users to view documents with a security level up to and including their own clearance level. It might seem appropriate to allow users similar rules for update access, but that gives rise to a potential problem.

An important aspect of a mandatory access system is that the rules are predefined. However, if a user can modify all data at their security level and below, this guarantee is broken. In our system described above, suppose two tables exist, VerySecret, with a security level of 8, and SlightlySecret, with a level of 6. If a user with level 8 clearance has modify access to both tables, then they can execute a query like this:

```
INSERT INTO SlightlySecret SELECT * FROM VerySecret;
```

Such queries would mean that any user with sufficient access could reclassify data by copying it somewhere else in the database. As a result, it is usual in such systems for a user only to be allowed to modify data at their highest security level. Thus the user could copy data from a level 8 table to a level 8 table, or from a level 6 to a level 8, but not from a high level down to a lower level.

In the discretionary approach, security restrictions are specified by rules that are part of the system itself, unlike in the mandatory system where they are external and cannot be modified. One example of a discretionary security system with which you may be familiar is the UNIX file permissions system that underlies both LINUX and Mac OS. In that case, a file on disk stores each user's name and ID. Each file has an owner user, whose ID will match one from the file, and privileges associated with the user, an owning group and all other users. The owner of a file or directory is able to change its access rights.

In an SQL DBMS, the rules are also stored as a table, as is user information. To use our university example as an illustration, suppose that a new Director of Undergraduate Studies is appointed. The role means that the staff member needs to be able to see and modify any and all data about the department's undergraduate students.

Before we consider how this is handled in SQL, it is helpful to use an informal language to show how we might want such a system to operate.

```
SECURITY RULE     undergrad-director
ALLOW             staff001
TO                INSERT, DELETE, UPDATE
```

```
   ON                 StudentInfo
                      WHERE Type=”Undergraduate”
   ON VIOLATION       REJECT;
```

This example shows the following aspects:

- **Name**. The rule has a name (undergrad-director). This makes it easier to describe, alter or delete.
- **User specification**. The rule specifies explicitly to which user or users it applies.
- **Operations enumeration**. The rule specifies which tasks the specified users are being granted permission to perform. Permission to perform an operation is usually referred to as a **privilege**.
- **Scope definition**. The rule specifies to which data objects the privileges apply. In our example above, the table and a relational expression have been used to limit the rows affected.
- **Violation response**. If an action is taken that violates this rule, we may wish to specify a behaviour, although the most obvious is simply to reject with an error.

2.5.1 Granting privileges in SQL

SQL supports data security through the granting (GRANT) and revoking (REVOKE) of privileges in a way similar to that described above. The syntax for these two commands is:

```
GRANT     @<privilege> | ALL PRIVILEGES
ON        <data object>
TO        @<user id> | PUBLIC
[WITH GRANT OPTION]
[GRANTED BY <user id>]

REVOKE    [GRANT OPTION FOR] @<privilege>
ON        <data object>
FROM      @<user id> | PUBLIC
[GRANTED BY <user id>]
[RESTRICT | CASCADE]

<privilege> ::= USAGE | <operation-with-optional-columns>
<operation-with-optional-columns> ::= <operation> ( @<column> )
<operation> ::= SELECT | INSERT | DELETE | UPDATE | REFERENCES
<data object> ::= DOMAIN <domain name> | [TABLE] <table name>
```

The `GRANT` command specifies a list of privileges. It is possible to specify privileges for domains, tables and views, using the ON clause. A USAGE privilege is the minimum access requirement, and is sufficient for the use of domains. `SELECT`, `INSERT`, `DELETE` and `UPDATE` correspond to their SQL command, and can be limited to specific columns of the table. The `REFERENCES` privilege allows a user to benefit from an integrity constraint that refers to the data object.

If `WITH GRANT OPTION` is specified, then the users specified can pass on any or all of the privileges granted in the statement to other users. They would do that by using their own `GRANT` statement.

The `GRANTED BY` clause is seldom used, since the default is to record the user executing the `GRANT` command as the one who gave the privileges. This clause provides a mechanism to support those cases in which it is useful to associate the rule with a different user account.

SQL security rules have no name, so the syntax for removing rules has to specify the rule to revoke by giving matching privileges and data objects. Most of the clauses of `REVOKE` are the same as in `GRANT`; the only exception is the final optional clause, which requires more explanation.

Consider a user, Alice, who was granted access to a table and had `WITH GRANT OPTION` specified. User Alice then grants the same access to user Bob, who in turn does the same for Charles. If Bob leaves the organisation, and is to have all privileges revoked, it may be sufficient not to specify either `RESTRICT` or `CASCADE`. In this case the `REVOKE` command applies to Bob only.

On the other hand, if Bob is found to have been dishonest, it may be that Alice wants to be warned about other users with privileges granted by Bob. In that case, the `RESTRICT` privilege can be used, and a `REVOKE` command will fail until all users that Bob has granted privileges to have their privileges removed or their `GRANTED BY` status changed. On the other hand, if Alice wishes to revoke the privileges of all these users in one command, then the `CASCADE` command can be used.

In SQL, the creator of a data object is, by default, granted all privileges for that object, with grant option.

Grant statements are often used together with views. Since it is not possible to specify privileges at the row level as in our informal example earlier, if a user should only have access to data where Type="Undergraduate", this restriction must first be used to make a view, and then access can be given to the view rather than to the base table.

Consider the two tables as illustrated in Figure 2.17, part of the database of a bank.

Accounts				
AccNo	**CustomerID**	**Type**	**Balance**	**OverdraftLimit**
01483331	ZYK015781475	Current	350.01	1500.00
01492125	ZYK015781475	Savings	9000.00	0.00
01596586	HXH515130551	Current	-223.44	500.00
01937584	NJV105137348	Current	5.23	150.00

Customers			
CustomerID	**Name**	**Address**	**Telephone**
ZYK015781475	Ronald Reynolds	33 Opal Road, Ilford	02084999121
HXH515130551	Douglas Goody	12 Mail St, Putney	02075392814
NJV105137348	Charles Wilson	15 Station Rd, Battersea	02074230067

Figure 2.17. Two tables from a bank's database.

Suppose that the sales department of the bank has two employees who are allowed to use the database to get contact information for customers. Since they may be promoting bank accounts, they should be able to see what type of products the customers have already, but not the account numbers, balance or overdraft limit. They are only allowed access for marketing purposes, so they should also have no power to modify any data.

```
DEFINE VIEW ContactDetails AS
 SELECT    Type, Name, Address, Telephone
 FROM      Accounts LEFT JOIN Customers USING (CustomerID);

GRANT      SELECT
ON         ContactDetails
TO         sales01, sales02;
```

We could have achieved this without using a view, by issuing two GRANT statements, one for each table, with limited columns. This method is not only clearer, but also allows the query executed by the sales staff to be simpler. Using a view is also more flexible – if sales staff are only to see users with positive bank balances, the view definition could have been changed to include a WHERE clause. Note that this allows the sales staff to have access to information that is sifted by a field that they cannot themselves read.

Similarly, a customer's access to their records might be defined by these three statements:

```
DEFINE VIEW ZYK015781475 AS
 SELECT    AccNo, Type, Balance, OverdraftLimit,
           Name, Address, Telephone
 FROM      Accounts LEFT JOIN Customers
 WHERE     CustomerID="ZYK015781475";

GRANT      SELECT
ON         ZYK015781475
TO         RonaldReynolds001;

GRANT      UPDATE (Address, Telephone)
ON         ZYK015781475
TO         RonaldReynolds001;
```

These security rules so far make no reference to the context in which the database system is accessed or used. There is no limitation by whether the connection is from a local machine or over a network, what the date or time of day should be, or any other contextual information. Because of this, such rules are called context independent. Conversely, if system-defined functions such as time(), date(), terminal() or user() are used in the definition of a security rule, that rule is said to be context dependent.

Activity

Check the documentation for your chosen DBMS implementation to see how security rules are implemented. Is there anything missing? Is there anything extra?

It is important to remember that sophisticated security in the database is useless if it can be bypassed. For example, the security of a DBMS can usually easily be compromised by administrator-level access to the server on which it runs. Even file-level access to the system may provide dangerous levels of access. Such risks should be part of any security plan.

There are many other security issues in deploying full database systems, but listing these and discussing strategies for dealing with them is outside the scope of this course. However, there is one very common attack method that anyone designing web interfaces should be aware of, since it is easily avoided, but otherwise potentially devastating.

SQL injection attacks

A common use of database queries is in web applications, where users interact with a DBMS indirectly by searching or browsing through dynamically-generated web pages. For example, if a user types 'Queen' into a search box on a music website, the server-side script that generates the results pages may execute a search like:

```
SELECT * FROM Songs
   WHERE Artist LIKE '%Queen%';
```

Now, suppose that, having discovered that the database contains results for this search, the user instead types 'Queen%' AND server_version_num LIKE '9%". Now, the query constructed on the server side looks more like this:

```
SELECT * FROM Songs
   WHERE Artist LIKE '%Queen%' AND server_version_num LIKE '9%';
```

If any results are returned, the user knows that the server is running PostgreSQL version 9.x.x. If none are returned, but no error message is given, then it is likely that a different version of PostgreSQL is present. If there is an error message, then either the server is protected or it is not running PostgreSQL. Similar queries could be constructed for other DBMS implementations or to probe the system platform, or to discover the structure of the database.

If the user is malicious, they might enter 'Queen%'; DROP TABLE users; SELECT". This time, the query constructed will remove the table on which all users are recorded by the DBMS, making the database unusable:

```
SELECT * FROM Songs WHERE Artist LIKE 'Queen%';
DROP TABLE users;
SELECT '';
```

The possible abuses of this security weakness are considerable. A malicious user can damage or even destroy the database, while a criminal one might access confidential information, place an order to be paid for by a different user or have their account given artificially high credit.

There are several options for mitigating this threat.

- The server side script will need at least one account for accessing the database. Limit the privileges of that account as tightly as possible. One approach might be to have a search account that has only read access – and even then only has access to the tables and columns that are absolutely required – and for those functions that need write access; for example, when an order is placed, to have their own accounts that also have specific privileges.
- Construct queries programmatically. If a form field requires a number, check that the input is appropriate. If a string is required, depending on the search, either remove or escape characters that might allow code to be injected – note that the examples above would not work without quote characters.
- Use prepared statements. Prepared statements are a feature of most SQL databases, and allow a query template to be defined and compiled in advance. Since inserting parameters into the template is carried out by the DBMS on clauses in a query, not into a string, injection is ineffective.
- Some libraries supporting the creation of web applications from databases have built-in mitigation procedures. Check the documentation of any that you use for what is available.

Although this may seem a simple issue to avoid, the long list of (known) past victims – which includes the British Royal Navy and the United Nations Internet Governance Forum – shows how poorly protected many systems are.

2.6 Database optimisation

Another practical consideration that you should be aware of is that of database optimisation. A full exploration of this is beyond the scope of this course, but there are two topics in optimisation that we shall briefly consider: indexes and denormalisation.

2.6.1 Indexes

An index (sometimes pluralised as indices) is a way of speeding up retrieval of data from a database. Without an index, searching a table for a particular value takes O(N) time[1], where N is the number of rows, since each row must be checked. Doing the same for a joined set of tables can make matters much worse. If two tables have N and M rows and no indexes, then checking the join criteria for them would take O(NM) time. The more tables, the worse this becomes.

[1] In this section and elsewhere in the subject guide, we use 'Big O notation' to indicate the way in which the size of the input to a problem relates to the time and memory needed to perform operations. A capital O, followed by a value in brackets indicates the 'order' of time or memory required. For example, an operation that takes O(n) time gets slower in proportion to the size of the input, n. If n doubles, the operation takes twice as long – the relationship is linear. O(1) time operations take the same amount of time no matter what the input, and are called constant-time operations. Since Log(n) < n for all positive values of n, O(Log(n)) operations are faster than O(n) and are described as sublinear.

An index is an extra data structure that allows a DBMS to find rows that match a query without looking at every one. The data structure chosen will have an impact on the performance of the index. For example, consider an algorithm that given any key value returns the location of the address stored based entirely on the key value – without having to look anything up. Such algorithms, called hashes, can be used to construct a hash table. An index that uses a hash table will have an approximately constant lookup time – O(1) – for searches that require an exact match. For non-exact searches, the hash table is of little use.

Where sorting or approximate matching are needed, a branching tree structure optimised for searching, such as a b-tree may be used. These give O(Log(n)) time retrieval – slower than a hash table – but they allow some approximate matches, ranged searches and support fast `ORDER BY` sorting. When searching string fields, a b-tree is useful when the search gives the beginning of the string (e.g. WHERE Name LIKE 'John %') but not where the beginning is unspecified (e.g. WHERE Name LIKE '% Smith'). For those interested in the b-tree as a family of data structures and some of its use in databases, Douglas Comer's 1979 article, 'The Ubiquitous B-Tree' is recommended. This is listed in Further reading.

Other data structures exist for indexing specific types of data or for optimising for known patterns of retrieval. Some focus on supporting full-text search; while others are optimised for geospatial, graphical or other specialised data types. It is important when implementing a database system to understand any requirements that the data and its use will place on it. In some cases, the choice of DBMS to use will be dictated by the indexing options available.

An index is automatically created for primary keys because of the uniqueness constraint. Without an index, if a new row is inserted into a table, the only way to check whether the primary key is unique is by checking every row in the table. Using an index, such a comparison is unnecessary, making insertion faster.

Many DBMS implementations also automatically index foreign keys, to speed up `JOIN` operations. It is important to know if the implementation you use does this – otherwise, you may need to create the indexes explicitly.

An index is part of the physical layer of the database and, because of that, it is not covered by the SQL standard. Nonetheless, all RDBMS implementations provide a `CREATE INDEX` command for explicitly creating an index.

Activity

How are indexes implemented in your chosen RDBMS? Find the command for explicitly creating indexes and try using it. What data structures are supported? Find out whether it is possible to add new index types.

When a query is executed, the DBMS will work out the best methods and order to use when carrying out the various elements of the query. This **query plan** will also involve a decision about which indexes to use, when a temporary table might be needed and other aspects of the physical processing of the data.

Let us return to the bank's Customers and Accounts tables used in Figure 2.17. For a large bank, the number of customers could easily be in the millions or tens of millions. For illustration, suppose that the Customers table has 3 million rows and the Accounts table 10 million.

For the query:

```
SELECT    Type, Name, Address, Telephone
  FROM    Accounts, Customers
 WHERE    Accounts.CustomerID=Customers.CustomerID
          AND Accounts.Balance>0;
```

There are several different ways of getting an answer. One option might be to make a cross join of the two tables, and then eliminate the rows that failed to match the `WHERE` criteria. The query plan might look like this:

1. Make a temporary table from the cross join of all rows of Accounts with all rows of Customers. Size of result: 30,000,000,000,000 rows ($3x10^{13}$).
2. Remove all rows where Accounts.CustomerID!=Customers.CustomerID or where Accounts.Balance>0. Indexes available: none; rows to check: $3x10^{13}$.

No indexes are available in the second stage, because the cross join operation has made a new table. Another possible plan would be to start with the Customers table, taking only rows from accounts that matched the CustomerID field.

1. For each row in Customers, find rows in Accounts for which CustomerID matches. Copy to a temporary table. Indexes available: Accounts.CustomerID, Customers.CustomerID; rows to check: 10^7; size of result 10^7.
2. Remove all rows where Accounts.Balance>0. Indexes available: none; rows to check 10^7.

This is already approximately a million times faster and uses much less space, but it still requires checking a large number of rows without the benefit of an index. Turning things around will make things even better.

1. For each row in Accounts, for which Balance>0, find the row in Customers for which CustomerID matches and return the result. Indexes available: Accounts.Balance, Accounts.CustomerID, Customers.CustomerId; rows to check $< 10^7$; size of result $< 10^7$.

This will be faster because all the fields have indexes at the point at which they are checked, because no data needs to be copied before the final result is returned, and because slightly less data is being used for the join. How much difference that would make will depend on the system, the nature of the indexes, and so on.

Although these numbers may seem outlandishly large, even a small retail outlet can reach millions of records easily, given a few hundred customers who, perhaps over a few years, make several hundred transactions, each containing tens of items per transaction. If queries start to execute slowly, it is useful to understand what is causing the speed reduction.

Most RDBMS provide an `EXPLAIN` command or something similar. Running it will describe the query plan, including the indexes that would be used and how many rows are expected at each stage. Understanding the query plan will be important in making decisions about when new indexes are needed or whether restructuring the database would help.

Activity

For your chosen DBMS implementations, find the EXPLAIN command and try it out on queries over the databases you have created so far. Make sure you understand the information it gives you. Experiment with adding indexes and see how the query plan changes.

2.6.2 Denormalisation

The physical level of a database system is where the details of storage and retrieval are handled, and it is at that level that we would expect to find ways to speed operations up without affecting the functionality of the system. Indexes, for example, are part of the physical level of the database as is the selection of the DBMS itself, along with the operating system. Adjustments at the physical level do not affect the queries themselves or any other of the higher levels of the database, and they can be carried out without changing any client code.

However, sometimes, it is impossible to keep the abstraction separated from questions of efficiency. The model that we implement is chosen for its match with the system that it models; it is then normalised to guarantee that certain update anomalies cannot happen. This process is, by intention, not based around concerns of speed; and normalisation, by separating the data into multiple tables, is likely to reduce the speed of queries and increase the complexity of queries.

An RDBMS is highly optimised for making joins efficiently, and in most use cases, the reduction in query speed resulting from normalising a database is irrelevant. Furthermore, the overhead involved in checking INSERT, UPDATE and DELETE statements in an unnormalised database to make sure that no anomalies are created can be large. Sometimes, then, there can be an argument for reversing the process of normalisation, combining tables to make single tables, with all the repetitions and redundancy that normalisation seeks to avoid. The process of reversing normalisation is called **denormalisation**. It is of use in certain, limited cases:

- where there is genuinely an important problem with speed that cannot be resolved in other ways; and
- where there are very many `SELECT` statements and few data modification statements.

In such a situation, the best option is usually a snapshot or materialised view (see section 4.5.2 'Retrieving data using views' of Volume 1 of this subject guide). These create quite a lot of data duplication but, where the RDBMS provides them, they include the mechanisms needed to ensure that the view is kept up to date as necessary, and can give protection against anomalies.

In Figure 2.1, we gave two tables, and introduced the idea of transactions by discussing the problems caused by updating one and not the other. In the activity that followed, you were given the opportunity to note that part of the problem arose from the fact that the tables would not have been created like that by someone following the procedure described in Volume 1 of the subject guide. Since the value of the `NumberOfProjects` field in the Programmers table was a calculated field derived from the `Assignments` table, it would have been excluded during the conversion from an E/R model to a relational one.

Had `NumberOfProjects` not been present in the tables, then the result of the failed set of operations would have been incorrect, but not inconsistent. Nonetheless, having this field turns a relatively complicated query into a simple one. If the speed gain is important enough for the programmer to be willing to sacrifice the guarantees that the RDBMS would otherwise make about consistency, then this sort of calculated field can be used. In effect, adding this field turns an operation that would be carried out at every query into one that is carried out on every update. If queries vastly outnumber updates, then the change may be beneficial.

The benefit can be seen more easily in large databases, where there is a requirement for calculated fields based on summaries of information in several tables, with multiple rows in each table being aggregated. However, in some cases, simply undoing a normalisation step – placing information from a separate table back into the parent table – is the best way to get the desired performance.

Denormalisation, beyond the use of snapshots, is generally to be avoided, and if it is necessary, it should be carried out with care – it is not an excuse for failing to normalise properly in the first place. The starting point for denormalisation should always be a fully normalised database, and as little should be denormalised as is necessary to meet the performance requirements.

For a fuller discussion of denormalisation and its risks see 'A note on denormalisation' section in Date's (2003) Chapter 13 'Further normalization II: Higher Normal Forms'.

2.7 Overview of the chapter

In this chapter, we introduced the notion of data recovery, describing the use of transactions, locking and logging to allow a DBMS to maintain a consistent state and to recover gracefully from system failure. We also considered how to manage security risks through user management and care when making aspects of database search available through client applications. The final section of the chapter touched on the issue of performance in a database and briefly introduced a few methods for optimising the speed of your database.

2.8 Reminder of learning outcomes – concepts

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

- explain the concept of the transaction and the mechanisms (`COMMIT` and `ROLLBACK`) that implement it
- explain how transaction recovery is achieved in a DBMS and how it is implemented in at least one SQL dialect

- explain how database recovery can be achieved through the transaction mechanism
- explain the two-phase commit protocol and its use in a distributed database system
- discuss the problems arising from concurrent access and the concept of serialisability
- define and explain the locking mechanism, including the concept of deadlocks and how they may be resolved
- discuss and give examples of issues of data security, including the support SQL provides for security control
- discuss the concepts of indexing and denormalisation in the context of database optimisation.

2.9 Reminder of learning outcomes – key terms

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

- ACID: Atomicity, Consistency, Isolation, Durability
- Concurrent access
- Database log/journal
- Data recovery
- Deadlock
- Denormalisation
- Index
- Locking (shared and exclusive); two-phase locking
- Mandatory/discretionary access control
- Privilege
- Serialisable schedule
- Two-phase commit (for distributed databases)
- Transaction
- Write-ahead rule.

2.10 Test your knowledge and understanding

2.10.1 Sample examination questions

a.
   i. What is a serialisable schedule? [4]
   ii. Name and briefly explain each of the ACID properties. [8]
   iii. What is a transaction? How does it relate to serialisation and ACID? [3]

b. User Bastian01 is to be given limited access to parts of a database.
   i. What SQL command would give him the ability to read the data in a table called Auryn? [2]
   ii. What command would give him the ability to change data in the Auryn table, in the column called Name? [1]

c. What security precautions would you recommend to a developer who is using text from a web form as input to a database query? [2]

d. ‘Physical data independence is achieved by the creation of a data model.’

  i. What is an index? [2]

  ii. Under what circumstances is an index automatically created by the DBMS? [1]

  iii. What factors would you consider when deciding what columns to index? [2]

Chapter 3: Distributed architectures for database systems

3.1 Introduction

In this chapter we consider the objectives of distributed architectures, exploring the problems they may cause. We describe some of the ways these objectives are achieved and how problems may be resolved, including necessary compromises when faced with competing objectives.

3.1.1 Aims of the chapter

The aims of this chapter are to introduce distributed database systems, which are driven by a set of objectives, discuss some of the barriers to achieving these objectives, before concluding with an introduction to some new models.

3.1.2 Learning outcomes

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

- define the concept of distributed database systems
- list and explain the objectives of the development of distributed database systems
- list and explain the problems that arise in the development of distributed database systems.

3.1.3 Essential reading

- Date, C.J. *An introduction to database systems.* (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], Chapter 21 (1999 edition: Chapter 20).

  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 24 (1999 edition: Chapter 19).

3.1.4 Further reading

- Connolly and Begg: Chapter 25 (1999 edition: Chapter 20).
- Özsu, M.T. and P. Valduriez *Principles of distributed database systems*. (Springer, 2011) 3rd edition [ISBN 9781441988348 (pbk)]. Sections 1.4–1.6 give good supporting reading for this chapter. The rest of the book provides in-depth reading for those interested in the topic.

3.2 Distributed database systems: an introduction

Sometimes, a company will decide that there should not be just one centralised database for all its data running on a single machine attached somewhere on the network. Suppose a company has a Sales department in one city, Development in a second and Personnel and Finance in a third. If each maintains a fully-separate, independent database, with no interaction between them, then the situation is not far from the file-based system described in Volume 1 of this subject guide. In short, this solution is less effective if many individuals require data from departments other than their own. It is particularly bad if there is a need to combine data from the different separate data resources.

An alternative approach is to ensure that the databases are connected to a network in a way that allows the DBMS at each site to communicate with the others. It then becomes possible for the DBMS to have access to the total data for the company and let the separation of the databases become an aspect of the operation of the database system as a whole. Individual users simply interact with the DBMS at their site, while the DBMS identifies the true location of the requested data and handles the queries.

Distributed database system. A distributed database system (DDBMS) is a collection of local database systems, or sites, where each site can be used independently of the others and where the sites are interconnected. Sites communicate with each other so that:

- each local database can access the other system's data; and
- from the point of view of the user, this access is made transparently – the user perceives all data as if it were stored locally.

This definition applies to distributed systems that follow the model for database systems that we have already encountered, and will provide the basis for the discussion that follows. There are certain use cases for distributing data sources for which this model is considered less appropriate, and other models are favoured. We shall consider some of those later in this chapter.

From the definition above, it follows that each site must have a DBMS that is capable of managing its data independently of the others. It is also a necessary requirement that an extra subsystem must exist at each site that caters for the interaction with the other sites. We have already encountered this in the discussion of transactions in the previous chapter.

We should state at the outset that that there is no requirement that sites are geographically distributed – they could co-exist on the same machine. The issues are generally the same however they are distributed, although some problems become more pressing in situations where communication between sites is slow or where some sites are less stable than others.

In our discussion of a RDBMS-based DDBMS, we will not explore implementations in any detail. There is too little standardisation or homogeneity between implementations for such details to be generally helpful. This lack of standardisation means that it is likely to be easier to create and maintain a homogeneous DDBMS – one with only the same DBMS implementation (and operating system) on all sites – than a heterogeneous one.

3.3 Objectives

The development of distributed database systems is usually considered to be driven by a set of objectives for what is to be achieved.

- Logical independence – a user should not see a difference between a distributed and a non-distributed system.

Date calls this objective the 'fundamental principle' of a DDBMS. However, as stated it is largely impossible, since DDL operations such as CREATE TABLE are likely to require the user to specify where the table is to be stored. If we assume that it refers only to data manipulation, however, it becomes more practical.

Beyond this principle, Date identifies a further 12 objectives. They are not all the objectives of a DDBMS, and they are not independent of one another, but they do provide a good example of the philosophy underlying this form of DDBMS.

1. **Local autonomy**. Each site of the DDBMS should be autonomous. Specifically, if A and B are any two sites in a distributed database system:
   - B should not be able to prevent the execution of any operation on A nor should it be allowed to influence its result, if the operation only needs access to the local resources of A.
   - The success of a local operation on A should not depend on B.
   - If A is unavailable, B should be able to carry out local operations as normal. There are limits to the practicality of this requirement, as we shall see. A more accurate statement of this objective might therefore be: local autonomy should be maximised.
2. **No reliance on a central site**. Firstly, if a central site controls all transactions, it will constitute a bottleneck and undermine many of the advantages of a distributed system. Secondly, if the central site becomes unavailable, the whole system becomes unavailable. This objective is closely related to the first objective, but it is achievable. Although each site may at certain points in time control a transaction, there is no permanent controller.
3. **Continuous operation**. No matter how many sites go down, some functionality should remain in the system, provided one site remains up. This is also a result of Objective 1 above; however, it is an important goal in itself and reflects the importance of keeping the following high:
   - availability – the system is running most of the time; and
   - reliability – the system fails infrequently and, when it does fail, is restored quickly.
4. **Location independence**, or **location transparency**. Users should not need to be aware of the location of the data that they are using. This also means that data can be moved from one site to another, perhaps to improve efficiency, without application programs needing to be changed.
5. **Fragmentation** or **partition independence**. Before we can explain this objective, we first need to explain **data fragmentation**. Simply put, data fragmentation occurs when a relation is divided into **partitions** or **fragments**.

We have already encountered one very constrained type of fragmentation in the form of normalisation, in which a table is split, by its columns, into multiple tables. The important difference is that normalisation is carried out for reasons of database logic, whereas proper fragmentation is motivated by performance improvements.

Consider a table containing all the customers of a large bank. Assuming that the table is indexed, searches for exact matches should be fast, no matter how large the table gets. Nonetheless, searches for partial or approximate matches may require the table to be searched row by row, without reference to the index. At this point, we might consider dividing the table into 10 smaller tables. If the original table had millions of rows, there is no computational time saving in searching all 10 smaller tables separately.

However, there are some situations in which there might be a performance gain. Firstly, in a distributed system, each table could be hosted on a different site, allowing for searching to happen in parallel. Although the computation time might be the same or greater, it could be divided among multiple computers and so a 5-second search might take 0.5 seconds on each machine. Secondly, if there is a logic behind the partitioning into tables, it might be possible to predict which table is most likely to hold the row that the user needs. For example, if the world is divided into regions, with each region's customers having a table, then the user generating the query might already know which table would contain the data.

In the case of a company located at multiple sites each of which houses a different part of the organisation, the table of employees could be split such that each division of the company has its own table, stored on a DBMS that is on the same site as that division.

For these partitions to be manageable, the fragments should be disjoint – so that no duplication is introduced – and result in no loss of information. There are two basic types of data fragmentation.

- **Horizontal fragmentation** – performed with a restriction operator. This is the sort that we have been discussing here. A horizontal partition of a table is sometimes called a shard.
- **Vertical fragmentation** – performed with a projection operator. Note that, as with normalisation, a vertical partition must duplicate enough fields to allow a join to reconstruct the original table. Vertical fragmentation is particularly useful for performance where either a subset of a table's columns has much heavier use than some others or where particular groups of columns tend to be used separately for different operations.

The fragmentation independence objective means that users should not be able to tell the difference between fragmented data and the original whole tables. This should mean that the database fragmentation can be changed for efficiency without changing application programs (a form of program-data independence). Since the users cannot see the fragmentation, it is the responsibility of the system (in practice, the system's optimiser).

For example, a company has horizontally partitioned its Employees table into two tables Emp1 and Emp2. Suppose the optimiser encounters the following query:

```
SELECT * FROM Employees WHERE Dept=”Development” and Age<40;
```

This might be rewritten as:

```
(SELECT * FROM Emp1 WHERE Dept=”Development” and Age<40)
UNION
(SELECT * FROM Emp2 WHERE Dept=”Development” and Age<40);
```

This could then be distributed to the two sites. However, suppose the optimiser has access to one other piece of information – Emp1 consists of members of the sales and payroll departments, while Emp2 consists only of Development department members. The tables would be defined such that Emp1 is specified as representing Employees `WHERE` Dept=”Sales” OR Dept=”payroll”, and Emp2 as Employees `WHERE` Dept=”Development”. With access to this extra information, the optimiser only needs to query Emp2, and can do so with a simpler query:

```
SELECT * FROM Emp2 WHERE Age<40;
```

6. **Replication independence**. Users should not need to be aware if data is replicated. Although we have specified that fragmentation should not

introduce duplication of data, there may be other reasons to have copies of tables or fragments in more than one site. Replication can mean:

- better performance, since multiple copies can be consulted at once, and also network-related overheads may be reduced by having a local copy of data; and
- better availability, since the data will still be accessible even if one of the copies is not.

Again, this independence allows copies to be created or destroyed without affecting application programs. Ensuring replication independence necessarily complicates many operations, since any operation that changes replicated data must be propagated to all copies before they are next accessed to avoid inconsistencies.

7. **Distributed query processing**. Databases tend to be large enough that the time taken to transfer data between sites in a network can dwarf the processing time. This means that it is normally desirable for the site holding a given set of data to process it, if doing so reduces the amount of information to send. In other words, a query should be processed to minimise the data transfer over the network and, ideally, to spread the load so that no bottlenecks form.

   This places a particular emphasis on the importance of the optimiser. A good evaluation of the best way to distribute and execute a query can make a substantial difference to execution time.

   Another effect of this objective is that set-level operators tend to be preferred over tuple-level operators, since transferring data in one go is usually preferable to transferring individual elements one at a time.

8. **Distributed transaction management**. As we noted when we described transactions in the previous chapter, a distributed database needs special mechanisms to ensure the ACID properties are preserved.

   If a site needs access to data not available locally, the optimiser on that site must start processes on the sites with the required information. These processes executed on behalf of a transaction on a different site are called agents. A transaction in a distributed database system consists of a set of agents executed at different sites. An agent can also create further agents if necessary. It is the responsibility of the DDBMS system as a whole to ensure that:

   - The whole transaction is atomic – either all the agents carry out their operations successfully or they all roll them back. This is ensured using the two-phase commit protocol introduced in the previous chapter of the subject guide.
   - Concurrent access to data is successfully resolved as for a non-distributed database. In particular, deadlocks must be recognised and resolved successfully.

   Since the use of agents requires a single controlling system, even if that controller changes depending on the site that needs the transaction carried out, this objective is in conflict with the objective of local autonomy. However, this is inevitable to ensure the ACID properties are preserved.

9. **Hardware independence**. The type of hardware being used should make no difference to the functioning of the DDBMS, provided the hardware can support the processing required of it. This is the first of several objectives that describe the desire to support heterogeneous systems.

Since homogeneous systems do not use different hardware, software or implementation, they have no need for this objective or those that follow.

10. **Operating system independence**. The operating system on each site should not affect the functioning of program applications. It should be possible to replace a site with one running on different hardware and a different operating system without having to alter the user interaction.
11. **Network independence**. The nature of the network implementation, hardware, addressing, etc. should not affect the function of the DDBMS.
12. **DBMS independence**. The strongest of the heterogeneity principles, this requires that the DDBMS should present the same functionality to users even if some sites change their DBMS implementation. DBMS independence can be achieved by means of gateways. These are beyond the scope of this subject guide, but some details can be found in **Date** (2003) in his section on DBMS independence, in Chapter 21 'Distributed Databases'.

3.4 Problems

We have considered the objectives of those who need a DDBMS to reproduce the characteristics and reliability of a DBMS. In this section, we consider the barriers to achieving those objectives.

One area that has evolved greatly in the last few decades is the speed of data transfer over wide area networks. Nonetheless, passing data around a network remains slower than accessing locally, especially when it is accessed from solid state memory. This is the case for the bandwidth of data transfer, but especially for the latency of the connection. While the imbalance between performance locally and over a network has reduced over time and will probably continue to do so, it is likely to remain significant, especially for applications that push data processing requirements to the limits of current hardware.

The remedy for this problem is usually to reduce the volume of data transfer over the network. This remedy applies both for traditional DDBMS approaches and for more recent methods for handling web-scale data resources.

3.4.1 Query processing

The optimiser of any DBMS can make a highly significant difference to the speed of execution of a query. The decision of when to combine tables and when to use indexes can make the difference for an operation between being polynomial and linear or even constant, in terms of time, memory and disk use, so the effects even on a single site can be substantial. When the system is distributed, the importance of the optimiser becomes even greater, but its task becomes proportionally more complex.

Suppose that a company's factories require various component parts to make its products. Each part can come from a variety of suppliers, so the company DDBMS has three tables:

- Parts: {PartID, Colour}
- Supplier: {SupplierID, City}
- Contract: {Supplier ID, Part ID}

We can assume that there would be other tables and that these tables would have other fields, but these are the only ones that concern us here.

Suppliers and Contracts tables are stored at site A and Parts at site B. A user at a third site, site C wishes to find suppliers from London who supply parts in red. The query might look like this:

```
SELECT Supplier ID
FROM Suppliers
   INNER JOIN Contracts USING (SupplierID)
   INNER JOIN Parts USING (PartID)
WHERE City='London' AND Colour="Red";
```

To understand how to optimise the query execution, we need more information than this. For this example, let us assume the following.

- Each row of each relation is 1,000 bits long.
- Suppliers has 10,000 rows, Parts has 1,000,000 and Contracts has 100,000,000.
- The estimated number of red parts is 100.
- The estimated number of contracts with suppliers from London is 10,000,000.
- The data transfer rate is 10,000,000 bits per second.
- The access time is 0.05 seconds.
- Local data transfer, access and computation times will be taken as insignificant compared with the times for network use.

There are several strategies the optimiser might consider.

a. Move the Parts relation to site A, and perform the joins and evaluate the result at A. This implies the following steps.
   1. Contact site A from site C and communicate evaluation method (0.05 s access time; communication and computation time are negligible).
   2. Contact site B from site A (0.05 s).
   3. Transfer all rows from Parts table to site A (1 m rows, each 1,000 bits = 10^9 bits at 10^7 bits per second = $10^9/10^7$ = 100 s).
   4. Perform the join and restrict according to the WHERE clause (assume 0 s).
   5. Contact C from A and transfer the results to C. (0.05 s access time; c.1000 rows of result, each 1,000 bits = 10^6 bits, needing 0.1 s to transfer).

   The time required for steps 1 and 5 will be the same for all strategies. We denote it by τ:

   τ=0.05 + (0.5 + 0.1) = 0.65 s

   The time taken to complete this strategy as a whole will be:

   τ + 0.05 + 100 = 100.7 s (2 m 40.7 s)

b. Move the Suppliers and Contracts relations to site B and evaluate the result at B. This involves:
   1. Contact B from C and communicate evaluation method (included in τ).
   2. Contact A from B to request Suppliers (0.05 s).
   3. Transfer Suppliers (10 k rows = 107 bits, takes 1 s).
   4. Contact A from B to request Contracts (0.05 s).
   5. Transfer Contracts (100 m rows = 1011 bits, takes 10,000 s).
   6. Perform the join and restriction (0 s).
   7. Contact C and deliver results (included in τ).

So, the time taken for this strategy is:

τ + 0.05 + 1 + 0.05 + 10,000 = 10,001.75 s (2 h 46 m 41.75 s)

c. At site A: Restrict the Suppliers table to only those in London, join the result with the Contracts table and project over {SupplierID, PartI}, to keep the row size the same. Then transfer the result to site B and join and restrict with the Parts table at B.

   1. Contact A from C and communicate evaluation method (included in τ).
   2. Perform join and restriction (assume 0 s).
   3. Contact B from A to communicate intermediate results and evaluation method (0.05 s).
   4. Transfer intermediate results to B (10 m rows = 1010 bits, takes 1,000 s).
   5. Perform join and restriction (0 s).
   6. Contact C and deliver results (included in τ).

The time taken for this strategy is:

τ + 0.05 + 1,000 = 1,000.7 s (16 m 40.7 s)

d. At site B: Restrict the Parts table to only red parts, transfer the result to A and finish the processing there.

   1. Contact B from C and communicate evaluation method (included in τ).
   2. Perform join and restriction (0 s).
   3. Contact A from B to communicate intermediate results and evaluation method (0.05 s).
   4. Transfer intermediate results to A (100 rows = 100,000 bits, takes 0.01 s).
   5. Perform join and restriction (0 s).
   6. Contact C and deliver results (included in τ).

The time taken for the query using this strategy is:

τ + 0.05 + 0.01 = 0.71 s

Activity

Can you think of other strategies? Write them out and estimate how long they would take. Find an example from earlier in this subject guide that queries on multiple tables and experiment with the effects of hosting different tables at different sites. Experiment with changing the size of tables and the data transfer and access rate. How much does that change the results?

Even for this simple example, the execution time for the same query varies from a fraction of a second to several hours – the fastest of these strategies is over 14,000 times faster than the slowest!

In the context of a distributed database, there are two types of optimisation:

- global optimisation – concerned with deciding which sites will execute which aspects of the query and how data should move between sites; and
- local optimisation – concerned with how to efficiently execute an operation that is to be executed locally.

A good optimiser for a general DDBMS is hard to develop, it relies on estimates of result sizes, system and network speeds and system load.

3.4.2 Catalogue management

The catalogue in a DDBMS records not only information about the base tables, views and users, but also the locations of data objects and details of fragmentation and replication. Such information is needed in order to provide the independence objectives discussed earlier.

In a non-distributed database, the catalogue is stored, naturally, locally, as part of the database itself. For a DDBMS, however, there are more options.

- Store the catalogue centrally. Such a system would fail Objective 2 above: the system would be dependent on a single, central point, and loss of that site would result in complete failure of the system.
- Store a copy of the catalogue in each site's DBMS. This carries with it the extra difficulty of propagating any changes around the distributed system every time a local catalogue change occurs. This compromises local autonomy, but is little worse than transactions in that respect. It does place both complexity and additional load on the system.
- Keep the information for each part of the system local, with its own DBMS. This makes any non-local operation expensive, since no site has information about the data on any other site. Repeated querying of the network for the location of a given table or fragment would either waste bandwidth or encourage local caching that would start to look like each DBMS holding the full catalogue of the DDBMS.

Catalogue management is a difficult problem for which the best answer will depend on the application, since each solution has benefits in one area, but adds cost in another. Different DDBMS implementations may provide a single approach or multiple options, but these are beyond the scope of this subject guide.

3.4.3 Update propagation

DDBMSs provide replication, because duplication of information across sites increases the speed and availability of operations, but replication carries risks as well.

Every time that an update is performed on a data object that has been replicated, the update must be propagated to all existing copies. If any part of that replication fails, then the transaction containing the update must be rolled back.

This gives rise to a contradiction – if a key advantage of replication is that sites continue to operate if the site holding one copy of the data is down, then the transaction model cannot require all replicas to be available before a transaction can be committed, without the advantage being lost. The restriction would also conflict with the objective of local autonomy.

One way around this is the so-called primary copy scheme. In the primary copy scheme, each data object has one primary copy and any other copies of that data are called secondary copies. If the database is an update that alters the data, the operation is only considered successful when the primary copy is updated. Once the primary copy is updated, the transaction can be completed, even if the secondary copies are yet to receive the update. It becomes the responsibility of the site holding the primary copy to ensure that the change is propagated. There are two possible scenarios.

- The initial update is requested on the primary copy. Once the update is carried out, the transaction can be considered complete, but the primary copy must now pass the changes on to the secondary copies.

- The initial update is requested on a secondary copy. The secondary copy must communicate with the primary copy and perform a coordinated update (or, if that fails, roll back). Again, the primary copy then ensures that the changes are propagated to other copies.

This solution allows greater local autonomy, but does not guarantee that the database will always be in a consistent state.

3.4.4 Recovery control

We dealt briefly with the issue of transactions in a distributed system in the previous chapter, when we introduced the two-phase protocol. Some extra details can be added, now that DDBMSs have been introduced.

- One site must play the role of the coordinator in any given transaction, although a different site may perform that role each time. Usually, it is the site where the transaction was initiated that takes the role, while all other sites obey the instructions of the coordinator. This means a certain loss of local autonomy, but does not necessarily breach the aim of avoiding a central control point, since the coordination role of any given site is temporary only.
- The two-phase protocol requires that the coordinator communicate at key points with the other sites; this creates some communication overhead. For situations where this represents a significant problem, there exist variations of the protocol that minimise communication.
- The two-phase protocol cannot resolve all problems that might occur as a result of a failure. A system where all agents commit or roll back together for any case of failure cannot be guaranteed.

3.4.5 Concurrency control

Any request for a lock requires extra messages travelling on the network – a problem mitigated somewhat by the primary copy approach described above. More concerning, however, is the difficulty in identifying deadlocks. In a distributed system, there is a new danger of a deadlock occurring between sites, with each waiting for the other to release locks. Such a deadlock is called a **global deadlock**. The only reliable way to resolve the problem is to maintain a global wait-for graph just like the local graph we described for local deadlocks. Note, however, that such a graph either requires centralisation, which we are trying to avoid, or replication, which makes it impossible to be sure that it is authoritative.

3.5 Concluding remarks and new models

As you will have realised by this point, where the single-instance, local DBMS is a provably predictable, correct and consistent solution to a large family of information-handling tasks, a DDBMS is always, to some extent, a product of compromises. Just as various aspects of normalisation and concurrency control on a DBMS represent a trade-off between reliability and processing speed and immediacy, a DDBMS must balance the objectives listed earlier with the restrictions required for ensuring as much correctness as possible. A decision about where compromises could or should be made is one that must be made with awareness of all the relevant issues.

The dramatic changes in networked computing technologies and the ways in which users interact with networks – particularly with the web – have pushed distributed data processing into a prominent position in many more applications than would have been imagined when the theory behind DDBMS was first being explored. In a world where many databases or similar applications are directly accessed online, many companies that deploy these tools decide to value continuous availability over some of the rigours we traditionally demand.

The most famous expression of these tensions is Brewer's Conjecture, sometimes called CAP theorem. In an invited paper at a conference in 2000, Brewer enumerated three goals for a DDBMS.

- **Consistency**: This is similar to the notion of consistency in the ACID properties, but weaker. For Brewer, it only means that all parts of the DDBMS should (eventually) agree on a consistent form of all data.
- **Availability**: Every request received by a non-failing node should result in a response. Ideally, the response should happen in a sensible amount of time, but that is not an explicit requirement.
- **Partition tolerance**: This refers to partition of a network rather than the partitioning of data that we considered earlier. Network partition occurs when some failure of infrastructure severs some of the lines of communication, meaning that some sites or sets of sites are incapable of communicating with the others. For example, if one company has sites in New York and London, serving requests from the Americas and Europe respectively, a problem with the transatlantic communications cables could mean that the two are maintaining different versions of the database. They may still serve requests, but messages sent from any sites in New York to any in London will be lost.

Brewer argued that these three goals are in conflict. Gilbert and Lynch later proved, given certain assumptions, that it is impossible for all three goals to be satisfied. This led to the rule of thumb that you can have any two of the three, but never all of them.

Activity

Thinking of modern uses of database systems, try to evaluate which of these goals might be considered important in each case. For example, which of the following might favour consistency over availability: a national stock exchange system; a dating website; a library catalogue?

The increasing weight placed by developers on high availability, especially for internet applications, has produced various attempts to create less strong forms of consistency. One such is eventual consistency, where an emphasis is placed on ways of resolving inconsistencies rather than avoiding them. This is sometimes codified as BASE (Basically Available, Soft state, Eventual consistency). BASE systems guarantee eventual consistency, although how long it will take before they will become consistent may be unclear, and until they do become consistent, they may give out results that relational theory would regard as incorrect or inconsistent.

In the next chapter of the subject guide, we will explore some new models for data handling, many of which also have to grapple with these challenges.

3.6 Overview of the chapter

In this chapter, we have described the objectives of distributed architectures, exploring the tensions they may cause; whether with the normal requirements of a conventional relational system; with each other; and/or with performance requirements. We introduced some of the ways that these objectives are achieved and problems resolved, but also the compromises that are inevitable when objectives compete with one another.

3.7 Reminder of learning outcomes – concepts

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

- define the concept of distributed database systems
- list and explain the objectives of the development of distributed database systems
- list and explain the problems that arise in the development of distributed database systems.

3.8 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 fragmentation/partition: horizontal and vertical fragmentation, fragmentation independence
- Distributed database system
- Global deadlock
- Local autonomy
- Location independence
- Replication/OS/Network/DBMS independence.

3.9 Test your knowledge and understanding

3.9.1 Sample examination questions

a. A company is considering migrating from a centralised database to a distributed system.
    i. Briefly explain what a distributed database system is. [3]
    ii. What factors should the company consider to help them make the decision? [1]
    iii. What is a homogeneous distributed database system, and why might it be preferred? [3]

b.
    i. What is a deadlock? [4]
    ii. What is a wait-for graph? Draw an example, featuring a deadlock. [4]
    iii. What issues in distributed systems affect deadlocks? Can they be resolved? If so, how? [3]

c. Briefly explain Brewer’s conjecture. [7]

Chapter 4: Advanced database systems

4.1 Introduction

This chapter starts with a review of three different implementations: the relational model, SQL and RDBMS, along with some criticisms of these approaches, before moving on to more recent advances.

4.1.1 Aims of the chapter

The aims of this chapter are to consider older and newer implementations of database systems and how the two relate. We move from pre-web to web-era paradigms before concluding with a list of major considerations for advanced database systems.

4.1.2 Learning outcomes

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

- discuss and evaluate critically the real and perceived shortcomings of the relational model and SQL
- discuss the deductive database approach
- discuss the OO database approach
- discuss alternative approaches to database systems, including:
  - deductive databases
  - object-oriented databases
  - key-value databases
  - document-oriented databases
  - the Semantic Web
- give at least one example of each approach
- discuss at least one alternative approach in greater detail
- explain how to decide between the various approaches and models for a given use case.

4.1.3 Essential reading

Criticism of the relational model:

- 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)], beginning of Chapter 27 (1999 edition: Chapter 21).
- Date, C.J. *An introduction to database systems*. (Pearson/Addison Wesley, 2003) 8th edition (international edition) [ISBN 9780321197849 (pbk)], beginning of Chapter 25 (1999 edition: beginning of Chapter 24).

Other models:

- Connolly and Begg, Part 7 (1999 edition: Chapters 21–23). Note: this is quite an enthusiastic promotion of object-oriented databases. Date is more critical:
- Date, Chapters 25–26 (1999 edition: Chapters 24–25).

- A brief introduction to object orientation may be found in Stevens, P. and R. Pooley *Using UML*. (Addison-Wesley; any edition, 1st edition was 1999), Chapter 2.

Big Data and MapReduce:

- White, T. *Hadoop: the definitive guide.* (O'Reilly Media, 2012) 3rd edition, Chapters 1 and 2 are highly recommended. Much of this book is generally useful, but Chapters 6, 7, 8 and 13 are particularly relevant to those interested in MapReduce and Big Data databases.

4.1.4 Further reading

Deductive database systems:

- Bertino, E., B. Catania and G.P. Zarri *Intelligent database systems*. (Addison-Wesley, 2000) [ISBN 9780201877366 (pbk)]. This gives an excellent overview of the motivations, issues and approaches, but is now somewhat out of date in terms of the software and technologies.

Big Data and MapReduce:

- Dean, J. and S. Ghemawat 'MapReduce: simplified data processing on large clusters', *Proceedings of the 6th Symposium on Operating Systems Design and Implementation – Volume 6 (OSDI'04), Vol. 6.* USENIX Association, Berkeley, CA, 2004 (http://research.google.com/archive/mapreduce-osdi04.pdf).

XML Databases:

- Connolly and Begg, later editions only, Chapter 31.
- Date, 2004 edition only, Chapter 27.
- For more detail: Harold, E.R. and W. Scott Means *XML in a nutshell*. (O'Reilly Media, 2004) [ISBN 9780596007645].

MongoDB:

- Chodorow, K. *MongoDB: the definitive guide.* (O'Reilly Media, 2014) 2nd edition [ISBN 9781449344689], especially Chapter 1. Otherwise, this is useful as an introduction to the tool, but has little evaluation or comparison.

Column-based databases:

- Chang et al. 'Bigtable: A Distributed Storage System for Structured Data', *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, 2006 (http://research.google.com/archive/bigtable-osdi06.pdf). (See also *Hadoop* book listed above.)

Semantic Web:

- Antoniou, G. and F. van Harmelen *A Semantic Web Primer.* (MIT Press, 2004) 2nd edition [ISBN 9780262012423]. A fairly short book that introduces most of the important concepts.
- Segaran, T., C. Evans and J. Taylor *Programming the Semantic Web.* (O'Reilly Media, 2009) [ISBN 9780596153816]. A more practical introduction to the technologies, including guidance for turning a relational schema into a graph schema.

Discussion of the relational model and NoSQL:

- www.dbdebunk.com; a blog by Fabian Pascal exploring the relevance of relational theory in a world full of alternative approaches.
- https://highlyscalable.wordpress.com; a blog by Ilya Katsov with useful practical discussions of Big Data and NoSQL.

4.2 Introduction to alternative implementations

Over the course of this subject guide, three distinct concepts have been introduced – the relational model, SQL and RDBMS implementations. The relational model is a theoretical model for organising data that has provable properties and a formal algebra. The model provides the foundations on which SQL was constructed. SQL itself is a standard that describes the way that a user or administrator should be able to interact with a conformant database, including some rules for how information should be processed. Finally, RDBMS implementations are products that implement the SQL standard, to a greater or lesser extent. This carries with it some relationship to the principles of relational theory, although there are differences between pure relational theory and RDBMS implications.

While it was never the case that all database systems were relational or SQL systems, either in theory or practice, the dominance of the SQL standard has meant that the DBMS and RDBMS have often been treated as interchangeable terms; while courses and books on database systems could focus exclusively on the relational model and SQL. As a result of this, the increase in the diversity of models that arose in the late 1990s and early 2000s were often defined as a reaction against SQL. These models can be very different from one another, but are often grouped together under the category **NoSQL**.

NoSQL databases can be defined as systems for which the logical modelling of real-world systems is not carried out in terms of the relational or E/R models. The name is unhelpful – it is not the SQL language that is rejected in most cases, but the relational model. Some NoSQL systems are implemented using SQL databases at the physical level. Although this is irrelevant for modelling, it has led some commentators to describe NoSQL as 'not only SQL'. This expansion of NoSQL also makes the occasional use of an SQL-like language, for querying NoSQL databases, look like less of a contradiction.

NoSQL approaches to information management and retrieval are often prompted by frustrations with some aspect of relational database systems. There are practical and theoretical shortcomings in each of the three elements – the relational model, SQL and RDBMS implementations – and there are certainly cases where the tools are not the most appropriate ones for a specific family of tasks.

In this chapter, we will introduce some of the types of alternative approaches being used, and briefly discuss some of the products that implement them. We have aimed to cover a wide range of approaches, and some of them are still undergoing development. We would recommend that, for any areas where you want to find out more and in greater depth, you consult reliable online resources as well as the Further reading that we have listed.

However, we shall first consider some common criticisms of SQL and the relational model, before considering the merits of the alternative approaches.

4.3 Criticisms of relational database systems

4.3.1 Generic, homogeneous data structure

Relational theory and SQL predate the rise of object-oriented (OO) programming. In an OO paradigm, the nature of a data object – its behaviour and its interactions with other objects – is available to the system either as part of its declared class or as attributes of the object itself. In SQL, on the other hand, all data objects are rows in tables, each value of which is a simple scalar.

The result of this simplicity of model is that many applications that need to apply reasoning that depends on the type of object being stored will perform those operations outside of the database, using it primarily as a data store. This tends to mean that more data is transferred than should be and less data management happens within the DBMS than might be desirable.

Features of the object-oriented approach are available in most RDBMS through domains, although how much this is the case depends on the implementation. Since domains can be defined with arbitrary complexity, they can easily stand in for classes, given a sufficiently powerful implementation.

Functions can be defined to operate on particular domains in a way that resembles method definition. This sometimes gives rise to some relational database management systems being referred to as Object Relational Database Management Systems (ORDBMS), but this seems unnecessary, since the technologies required for this level of object-oriented processing are already part of the relational model.

Nonetheless, the alignment between the relational model and SQL on the one hand and OO as it is most commonly encountered in programming languages is not a perfect one, and some would prefer a closer match.

4.3.2 Difficulty with recursive queries

Suppose that a student has been trying out their new DBMS with a simple family tree application for their personal history research. There is just one table – People. Each person entry has an ID, some biographical information and then the IDs of each of their parents.

Figure 4.1 illustrates a family tree and Figure 4.2 shows how it might look as a table.

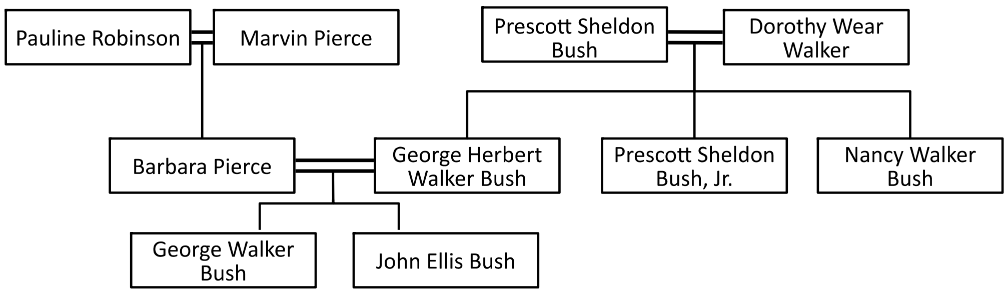


Figure 4.1. The Bush family tree.

ID	Name (at birth)	Mother	Father	Year of birth
1	Pauline Robinson	NULL	NULL	1896
2	Marvin Pierce	NULL	NULL	1893
3	Prescott Sheldon Bush	NULL	NULL	1895
4	Dorothy Wear Walker	NULL	NULL	1901
5	Barbara Pierce	1	2	1925
6	George Herbert Walker Bush	4	3	1924
7	Prescott Sheldon Bush	4	3	1922
8	Nancy Walker Bush	4	3	1926
9	George Walker Bush	5	6	1946
10	John Ellis Bush	5	6	1953

Figure 4.2. Some information about the family tree as a table.

Now, suppose that this project becomes very large – our student has now added very many people to the table. While she is showing the application off at a party two people who are in her database ask if she can check whether they have a common ancestor and, if they do, who it was and how far back in their history you would have to go to find them.

This is a fairly easy task in most programming languages – a tree is an easy structure to traverse, and iteration or recursion are usually provided as standard. However, none of the basic relational operators will help.

Since trees and graphs are very common structures in real-life applications, it is a serious shortcoming to be incapable of handling them effectively. If the application must retrieve the entire table in order to solve the problem, or must send queries to the database repeatedly, to retrieve the tree one node at a time, then the DBMS has failed to justify its use.

One very simple solution to the family tree problem – introduced in Chapter 2 in Volume 1 of the **CO2209 Database systems** subject guide – is to populate an Ancestor table. This table would simply list every ancestor at every level for every entry in People. Figure 4.3 illustrates this as might be applied to the table in Figure 4.2. As well as adding redundant data to the system, which presents a challenge for the integrity of the database, this option is very space intensive and only works for tree structures like a family tree, in which every node has a finite, non-cyclical list of parents. For a graph, this would be more problematic.

Ancestor	
DescID	**AncestorID**
10	5
10	6
10	1
10	2
10	3
10	4
9	5
9	6
9	1
9	2
9	3
9	4
8	3
8	4
7	3
7	4
6	3
6	4
5	1
5	2

Figure 4.3. An Ancestor table, derived from the main table, lists all descendents and their ancestors. For the 10 people listed above, we have 20 rows, because the tree is quite small. Adding just one of George Walker Bush's grandchildren would add 30 rows to the table.

In fact, SQL has had the ability to perform recursive queries since 1999, and most DBMS implementations (excluding MySQL, at the time of writing) have implemented these. The syntax, called common table expressions, allows a recursive query using the keyword `WITH` (or `WITH RECURSIVE` in PostgreSQL). Whether systems are well optimised for this and whether these sorts of queries feel idiomatic within SQL is arguably a matter for investigating at the point where a decision is being made about the system to use.

4.3.3 Mismatches between application programs and DBMS

A database will usually be encountered via at least one piece of software written in some other language. Often there will be a chain of such software, culminating in either a web page or a standalone application. Many programming languages interact well with each other because they share programming paradigms, data structures and interface elements. There are fewer commonalities between such languages and the relational model implemented in SQL-based databases. This usually means extra program layers to translate between the concepts and operations of the database and those of the application itself.

These are very practical concerns. Database systems are very different to most other programming environments and large organisations will usually hire database developers and programmers separately.

On the one hand, a company will need to consider whether to hire a separate database developer or whether they will do better using an option that integrates better with their other programming skills. On the other hand, a risk of developing a database application that looks like any other application and is written by a programmer without database expertise is that it may perform poorly, lose data or pose a security risk. In other words, the relational database paradigm is different partly because the problems it is designed to tackle are often different.

4.3.4 SQL does not support...

Commonly, specific types of application are raised as presenting situations that are impossible or difficult within the RDBMS paradigm. In many cases, these are true, and we will consider some in more detail separately in this chapter; however, some cases are actually observations about a lack of tools rather than a failure of the model. For example, Connolly and Begg describe geographical information systems (GIS) as one such application. They argue that a query for all shops within a two-mile radius of a given location is problematic for an RDBMS.

Position and navigational distance are functions of multi-dimensional information. Location, for example, must be specified in at least two dimensions and, for real accuracy, those dimensions must be mapped onto a globe, since they describe places on an approximately spherical world. The dimensions taken separately will not give sufficient information to solve the distance problem helpfully – a clause along the lines of `WHERE` Latitude<y+5 `AND` Latitude>y-5 `AND` Longitude<x+5 `AND` Longitude>x-5 will not give a particularly useful answer.

What this means, is that for a database application to support GIS, it must have spatial data types, so that there is no need for the queries to handle the complexities of the domain. For a given DBMS implementation, the question is whether they support or can be made to support such types; for example, through domains and user-defined functions. PostgreSQL, for example, is extensible in such a way that GIS extensions can and have been made. Since the problem was not a fundamental one about the model itself, they can be remedied by choice of DBMS, provided that the tools or extensions required have already been developed.

4.3.5 No ad hoc structures

Databases are based on tables that can be created at any time, with any combination of columns, but if every data object being stored is unique in structure and hard to predict in advance, then there would be a risk of an overly complicated model evolving, with very many tables, often containing only one row or with most columns being NULL, and querying would be proportionately complicated.

In more general terms, the relational model is based around highly-structured information with a good degree of consistency and overlap in the documents being stored. Web documents in HTML are good examples of information that has some structure and organisation, but little that is guaranteed, nor a very clear pre-defined form. Many other documents have such issues.

Such documents can be stored in relational databases, but either there is a requirement for a complex and evolving data model, or one may simply place documents into a field and process them using external routines, which reduces their usefulness.

4.3.6 Problems with scaling

Tables in RDBMS implementations are usually indexed in ways that are very fast, no matter how large the dataset is. Despite this, there are other aspects of the RDBMS approach that do not perform so well at scale. Often these aspects are those concerned with ensuring that the database remains correct and consistent. In some cases, alternative strategies can be devised that are optimised for specific requirements sets. For example, applications that are either less concerned about the continuous correctness of the data involved or where data will be updated only very rarely can be treated in a different way.

In some cases, the reason for using a different paradigm is that, given sufficiently good programmers, and enough of them, a specialised solution will perform better than a general-purpose database system. This is especially likely to be the case for industrial users with very large amounts of data and tight performance requirements.

In other cases, if no current RDBMS system supports a particular set of requirements for data processing or for distribution, then it may be considered easier for a company to either use a different approach where it has been implemented or to hire programmers to write a library in their programming language of choice. Often tools evolve from relatively local, specialised projects to larger, more data-intensive ones, in which case, transferring the functionality from an existing framework to one involving a DBMS may be considered too costly.

4.4 Other models for data handling systems

Some alternative database models originate in an academic environment, where the concern may be either the search for alternative theoretical groundings for database systems or better alignment with other programming paradigms.

Other models may originate in industry, and are often more closely associated with a product or the particular type of data that they are designed to store. Performance, behaviour and robustness will often change as the product becomes better understood, and as the range of users becomes greater. However, there are often connections between the ideas of these two communities – and each will usually know the work of the other.

For the quick survey that follows, we start with a brief introduction to two paradigms that were particularly favoured by academic communities in the 1990s and then, moving into more current, usually industry-led, approaches. We shall see, in the process, how these newer models relate to the older ones.

4.5 Pre-web paradigms

4.5.1 Deductive database systems

In the relational approach, the database consists of a set of extensionally-defined base relations. All tuples for each base relation are explicitly provided.

We have seen one of the effects of this in the family tree example above. The notion of an ancestor (as opposed to a direct parent) was not in a tuple, and was thus not available to the system for querying. One of the solutions for addressing that shortcoming was to make an explicit base relation for representing the concept of ancestor, even if that duplicated information.

Now consider a system where we could define what we mean by ancestor. It might look something like this:

```
ancestor(c, a) ← parent(c, a)
ancestor(c, a) ← parent(c, a') AND ancestor(a', a)
```

What these lines say is that a is an ancestor of c if either a is the parent of c or if a is the ancestor of the parent of c. In the second of these statements, ancestor(a', a) will expand out to find either a parent or a more remote ancestor.

In such a system, a query would be a case of asking the database to confirm whether two individuals were related according to predefined logic definitions of relatedness.

These systems are called deductive, because they deduce the truth of statements by reasoning using logical definitions applied to 'axioms' (the equivalent of base relations), rather than needing to be provided with all the explicit information they need and the steps required to turn that information into an answer. The system of logic used here is called **first-order logic**, which is only one class of formal logic. We will encounter another class, description logic, later in this chapter.

Data definition

To continue with the family tree example above. Information can be added to the database by adding **ground axioms**.

Let us say that we add people to the database by specifying name, date of birth and sex:

```
person('Leia Organa', 21/10/56, f)
person('Luke Skywalker', 21/10/56, m)
person('Han Solo', 13/7/42, m)
person('Anakin Skywalker', 14/7/34, m)
person('Kylo Ren', 3/10/82, m)
...
parent('Anakin Skywalker', 'Leia Organa')
parent('Anakin Skywalker', 'Luke Skywalker')
parent('Leia Organa', 'Kylo Ren')
parent('Han Solo', 'Kylo Ren')
```

We can also add some more **deductive axioms** about the concepts underlying the system:

```
father(X, Y) ← parent(X,Y) AND person(X, _, m)
mother(X, Y) ← parent(X, Y) AND person(X, _, f)
sibling(X, Y) ← parent(Z, X) AND parent(Z, Y) AND NOT equal(X, Y)
```

```
brother(X, Y) ← person(X, _, m) AND sibling(X, Y)
sister(X, Y) ← person(X, _, f) AND sibling(X, Y)
```

Note the use of a wildcard here – since we are not concerned about the date of birth of individuals concerned, we leave the second argument blank when specifying person in these axioms. This is a form of **template**, something that we will see again when we look at Semantic Web technologies.

Querying

There are two types of queries in logic-based systems – those that retrieve Boolean True/False statements about the query, and those that return values or sets of values.

The first type of query might be used to answer questions such as:

- Is Anakin Skywalker Luke's father?

  ```
  ← father('Anakin Skywalker', 'Luke Skywalker')
  ```

- Is Anakin Skywalker an ancestor of Kylo Ren?

  ```
  ← ancestor('Anakin Skywalker', 'Kylo Ren')
  ```

Note that in neither case was a tuple provided for the predicate being queried – we have stated only parent relationships and the sex of the people involved. The facts that they are a father in the first case and an ancestor in the other has been deduced.

For the second type of queries, where we only wish to see a particular value, the query is like those above, but substituting a variable for the information we want to see:

- When was Leia Organa born?

  ```
  ← person('Leia Organa', DoB, _)
  ```

- Who was Leia Organa's brother?

  ```
  ← brother(Bro, 'Leia Organa')
  ```

Where we require a set of results, deductive databases provide a forall predicate that finds all values that satisfy the formula:

- Who are the parents of Kylo Ren?

  ```
  ← forall(parent(Parents, 'Kylo Ren'))
  ```

- What is the name and date of birth of all parents of at least one male child?

  ```
  ← forall(person(Parent,DoB,_) AND
     parent(Parent, Son) AND
     person(Son,_,m))
  ```

Updating operations

The equivalent to the `INSERT` operator in SQL is assert, and works in a similar way:

```
assert(person('Padmé Amidala', 1/3/29, f))
assert(parent('Padmé Amidala', 'Leia Organa'))
assert(parent('Padmé Amidala', 'Luke Skywalker'))
```

To delete an axiom from the database, the retract operator is used:

```
retract(parent('Luke Skywalker', 'Kylo Ren'))
```

We have not touched on integrity constraints here, but the axiomatic approach continues here as well – statements about how data objects relate to one another can be made as general rules and the database can then police the applicability of those rules as data changes.

It should be clear by now that deductive databases have a more unified mode of interaction than relational databases, with broadly the same formalism operating for data definition, manipulation and access. Rather than relying on calculated fields or information to be rendered accessible through explicit queries or views, a deductive database is capable of generating this information from general axioms on demand.

Many find this a more intuitive way of interacting with a database.

These advantages do come with difficulties. Firstly, the expectation is for the database engine to perform a lot of the conceptual modelling that is carried out by the developer in the relational model. This translates to a more complex system, both less efficient in practice and harder to develop. New challenges, such as avoiding logical loops and optimising deduction speed, present barriers to the development of commercial models – as does the comparative rarity of logic programmers trained with languages such as Prolog.

Although the full deductive database model is rarely implemented for production systems, ideas developed for them have been taken up elsewhere, particularly for Semantic Web systems.

4.5.2 Object-oriented database systems

Object orientation (OO) is an approach to software development which brings aspects of good programming practice to the fore and allows programmers to make explicit particular models of the data being handled. Advantages of an OO approach include the following.

- The use of modelling concepts that have closer correspondence with real-life systems. Objects have attributes that refer to the attributes of the real-life 'thing' being modelled.
- Class-based object orientation, which is the most common form, groups and generalises those objects in a way that provides rules for their behaviour in a way that can also match how real-life systems are understood.
- Objects provide a strong mechanism for modular programming.
- Libraries of objects provide an intuitive mechanism for reusable components.

While all of these advantages are aspects of programming that can be available without an OO approach, they are easier and are, to some extent, built in when object systems are used.

Now read

If you are new to object-oriented programming or need a reminder of the principles, most of the core concepts are very clearly introduced in Stevens, P. and R. Pooley *Using UML*. (Addison-Wesley; any edition; 1st edition 1999), Chapter 2.

There are several ways to bring the OO approach to database systems.

- Implement classes as domains, and user-defined functions as methods on those classes. This is available in most current DBMS, although how powerful this is depends on the implementation.
- Embed SQL into an OO programming language. This requires more integration than the usual database libraries, since to take advantage of the OO aspects of the language, the library must perform extra operations usually reserved for the database system:

    - Since the DBMS and the programming language provide different data types, this mismatch must be resolved by translation code.
    - Additional type checking must be carried out, since the database itself can no longer enforce all rules regarding data types. This may also mean that integrity constraints must be enforced in the embedding language. Restrictions may also be necessary to prevent database access through other applications, since they might occur without appropriate type checking.
    - Management of checkpoints and transactions may have to be moved to the programming language.
- Turn an OO programming language into an OODBMS. Use the object system of an existing programming language, but make variables persistent (saved to permanent storage), permit concurrent access and so on.
- Create an OO DBMS from scratch.

All of these approaches, and others, have been put into practice from the 1980s onward.

Now read

Connolly and Begg are very strong proponents of this model, and if you are interested in more detail you should consult their chapters on the topic.

To summarise, the OO DBMS approach can offer advantages such as:

- richer modelling capabilities
- easier extensibility
- closer integration into existing programming paradigms
- removal of the mismatch between data structures in code and database
- expressive query language.

Disadvantages depend on the implementation, but may include:

- limited standardisation
- tension between database management tasks such as query optimisation and OO goals such as encapsulation
- locking and transaction control may be more complex.

4.6 Web-era paradigms

During the 1990s, at the same time as the World Wide Web quickly spread to become a part of almost every aspect of human endeavour, advances in data storage, networking and miniaturisation technologies dramatically increased both the ability to move and store data and the diversity of ways and contexts in which it is collected. New companies appeared, whose core business was supporting the exploration of that data. It is in this context that the newer models have evolved.

At the same time, as the quantity of data increased, some technical limitations became more constraining, and these limitations provide the motivation for the search for alternative strategies to overcome them. In 1991, a 40MB hard disk drive might have had a data transfer rate of less than one megabit per second. By the mid-2010s, hard drives have increased in storage capacity by a factor of 100,000, but transfer rates are only a few hundred times faster.

This means that although we can store vastly more data, it takes far longer to read the contents of a full disk. Other transfer speeds have increased at rates far lower than the capacity of drives and the computation speeds of processors. For data-heavy applications, these limitations necessitate heavily-optimised, distributed approaches.

4.6.1 Key-value databases

The simplest form of data storage is probably the closest to that of a file system. Since it is simple, it can be fast and its behaviour and consistency easier to predict. At its most basic form, a key-value database is just a persistent associative array – the database simply stores a data object or a list of data objects, associated with a key by which that object or list can be retrieved.

If the object is a file path or a URI, it can appear several times in the database, each time associated with a different key. This works well for storing the results of indexing a set of documents, and underpins document search technologies such as Apache Solr. Each document to be indexed has keywords (or the roots of keywords) extracted, and each keyword is then added as a key to the database, with the document path added to the values list.

Performing a textual search on the database is then a simple case of looking up all entries with the combination of keywords that have been entered into the search field. Retrieving the set of documents for each keyword is likely to be a constant-time operation, and then the set operation to find the documents that contain all keywords is carried out on a much smaller number of documents.

In the coming sections, we shall explore some of the technologies and models that have been built around key-value concepts.

Big Data and MapReduce

Big Data is a much-used term, but has no precise, generally-agreed meaning. Usually, Big Data implies some or all of:

- data that is too large to hold in the volatile memory of a single machine
- data that necessitates distributed storage
- data that will be processed as a whole, rather than by sampling a subset.

We will generally focus on data generated by web-based companies in the discussion below, but Big Data is also important in scientific research, with significant contributions from fields such as astronomy, particle physics and molecular biology.

Big Data must be stored somewhere, and the first component of any system storing large amounts of data is the file storage system. We shall not consider how these work, but note that, while Google developed their own proprietary system (GFS) for use in-house, Yahoo! adopted and extended an existing open-source file system called HDFS, the Hadoop Distributed File System. HDFS is a distributed file system that uses TCP/IP for communication between nodes. It is optimised for data that is written once, or at least relatively rarely, but read very often. Data is broken down into 'chunks' which will each be replicated several times across the network. HDFS is extremely fast for its main usage patterns, but this speed is partly achieved by not implementing some features that a desktop file system would implement as standard.

One useful aspect of key-value databases is that there is no significant interaction between entries in the database. At its core, this is a model with no joins. Not only does that make the database fast and simple, but it also makes distribution a lot easier. Since no 'row' in the database is dependent on any other, horizontal partitions of the data set have no effect on functionality. This approach is particularly suitable for data where normalisation and joins are less useful in the first place, such as the hundreds of millions of hours of video on YouTube.

There are other assumptions that can be made if the database is acting as the basis for search functionality in large, online databases. In those cases, there is rarely an absolute need for the real-time update of indexes whenever a new video is uploaded or website created.

We have seen how the optimiser of a DDBMS can improve the execution speed of a query by minimising the size of data transfers, performing restrictions at the site where the data is located. This is a specific example of a general rule – where stored data must be read, it is better to carry out the computation near the data than to move the data elsewhere first.

Very many queries, including SQL queries, can be thought of as having two phases. In the first phase, data is gathered from base tables, and projected and transformed as necessary. In the second phase, the results are processed and summarised before being presented to the user. Let us take as an example, one query from Volume 1 of this subject guide:

```
SELECT      Degree, Age,
            COUNT(*) AS Number_enrolled
FROM        Students
GROUP BY    Degree, Age;
```

In the first phase, the query gathers all the rows from Students, removes all columns except Degree and Age and then groups them together by these two fields. In the second phase, the result is summarised – instead of returning all the rows, the DBMS counts the number of results in each group.

Google engineers observed this two-phase pattern in many of the indexing operations that they were carrying out for their various different search engine and street-mapping products. They also noted that only the first of these steps needs access to the base relations, and that, in many of their use cases, the second phase could be easily parallelised, since each group in the aggregate function can be treated separately. Their solution was to develop a programming model that they called **MapReduce**.

Each MapReduce operation starts with a Map phase. In this phase, a Map operation provided by the user is carried out on all data. The Map operation takes the raw data as input and returns a collection of (key, value) pairs. The keys in the output of the Map phase need not be the same as the keys in the database. The output of the Map phase is then redistributed to reduce worker nodes. Reduce functions summarise all the values that share the same key, so data can be assigned to reduce workers based on the value of each key. The results are then gathered and returned to the user.

To illustrate this approach in practice, consider the task of counting the number of occurrences of each word in a corpus of literature. This sort of basic statistic can be interesting on its own, but can also be the first step towards more advanced style analysis. Our input documents would each be a work of literature – the key of each (key, value) pair would be the title and the value would be the textual content of the book. Figure 4.4 gives an idea of how part of the dataset might look.

Key	Value
ΙΛΙΑΔΑ	Μούσα, τραγούδα το θυμό του ξακουστού Αχιλέα, τον έρμο! π' όλους πότισε τους Αχαιούς φαρμάκια,…
Great Expectations	My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip. So, I called myself Pip, and came to be called Pip…
Der Verwandlung	Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fan der sich in seinem Bett zu einem ungeheuren Ungeziefer verwandelt…
The Color Purple	You better not never tell nobody but God. It'd kill your mammy. Dear God, I am fourteen years old…

Figure 4.4. The key and the beginning of the value for a literary dataset.

Now, the user provides a Map function like this:

```
map(String key, String value):
   while(String word = nextWord(value)):
         EmitIntermediate(word, 1);
```

For each text, this pseudocode function takes each word, and sends it to the MapReduce coordinator. The result of applying this over the corpus might begin like this:

```
[[“Μούσα”, 1][“τραγούδα”, 1], [“My”, 1], [“father’s”, 1] ,
[“family”, 1] , [“name”, 1] , [“you”, 1] , [“better”, 1] ,
[“Als”, 1]…]
```

Note that the intermediate keys here are individual words. These are entities that had no separate existence in the raw data structure. We have mixed the texts together here to illustrate the point that these would be executed in parallel, but there is no requirement in MapReduce to gather all the results together in one array before sending parts out to the reduce workers.

The reduce function is also simple. Given an iterator that loops over the values that have the same key, its pseudocode might look like this:

```
reduce(String key, Iterator values):
   int result=0;
   for each v in values:
      result += v;
   Emit(result);
```

As a result of applying this, each word will be associated with a count of the number of occurrences within the corpus.

Now read

This example is based on one described in a 2004 paper by Google engineers Jeffrey Dean and Sanjay Ghemawat. The paper is listed in the Further reading section at the beginning of this chapter and is a good introduction to the basics of MapReduce.

On the basis of this description of Google's MapReduce, an open-source implementation was created by Apache developers as part of the Nutch project. This gave rise to Hadoop, which adds HDFS as a file system, and formed a core component of the search infrastructure at Yahoo! from 2006.

The strengths of MapReduce are its simplicity, the ease with which it can support partitioned data and processing, and its ability to handle failures of individual nodes gracefully. MapReduce implementations usually have a coordinating node that can cause a whole job to fail if it crashes. There are recovery strategies for that, and implementations may also replicate this node, but this remains a point of weakness.

MapReduce on its own requires relatively simple tasks on very static data, and, although it is fast for the volume of data and the amount of hardware used, it is not a real-time solution for interactive search. MapReduce has proved useful for Big Data processing because the analysis usually involves running batch jobs once, or at regular intervals, on data that has been captured all at once.

Many tools have been built on MapReduce systems which provide useful abstractions and can make tasks more intuitive to create. One notable example of such a tool is Apache Hive, which runs on top of a Hadoop system and provides an interface very similar to SQL. Each query is interpreted into Map and reduce functions as necessary without the user needing to be aware of the underlying technology.

Google have since abandoned MapReduce and GFS in favour of a more responsive approach under the heading of Google Cloud Dataflow. An open-source release of the underlying tools is available as Apache Beam.

Activity

Read Chapter 2 of Tom White, *Hadoop: the definitive guide.* (This is listed above in Essential reading.) Try installing Hadoop and running the exercises in that chapter.

4.6.2 Document-oriented databases

We have noted that a relational database relies, on the whole, on having prior knowledge of the full formal structure of the data for which it is used. Now, consider an HTML document:

```
<!DOCTYPE html>
<html lang=”en”>
   <head>
      <meta charset=”utf-8”/>
      <title>Welcome page</title>
   </head>
   <body>
      <nav>
         <ul>
            <li>Welcome page</li>
            <li><a href=”more-details.html”>More details</a></li>
         </ul>
      </nav>
      <main>
         <h1>Welcome</h1>
         <p>Welcome to my website. It contains many exciting things.
            Including a picture of me:
            <img src=”my-picture.jpg”
            alt=”Photo of the webpage author standing in front of an
            amusingly-shaped tree” />
         </p>
      <main>
   </body>
</html>
```

This document is rich with information, and it does have a clear structure. It could be represented in a relational database, but the complexity of such a structure is likely to be excessive for most uses. XML and HTML are traditionally parsed into a document object model (DOM), which is a tree structure. Queries and update modifications are carried out on the parsed structure itself. Elements can be located in terms of their identity (using a unique ID), their properties (e.g. all the <img> elements that are contained in a <p>), or their position in the graph (e.g. the third <p> in the document).

The sort of document where there is a structure, but little is known about it in advance, or where there are few restrictions on how the elements of the structure are compiled is often called **semi-structured data**. Another common source of semi-structured data is JSON, a format for transmitting commonly-used data structures such as objects, arrays, strings numbers and so on. This format is often used as an interchange format between applications written using different programming languages, and it provides an easy way for an application to record the current state of its variables.

This sort of data processing and retrieval is called document-based, or document-oriented, because often all that is known at the outset is that there will be documents, and the format in which they will come. This is also the most common unit of interest. A document in a document-oriented database system is the equivalent of a tuple in the relational model, with many of the same implications. In the relational model, it is likely to be easier to query for information within a tuple than to explore a complex network of joins. Similarly, it is often easier in a document-oriented system to retrieve a piece of information per document or to restrict the set of documents based on one of their attributes than to perform the equivalent of a join of documents and then query those.

It is difficult to generalise about document-oriented databases, since the approaches can vary considerably, and a detailed exploration would certainly be outside of the scope of this subject guide. Instead, we shall briefly consider two examples: the first is the use of XML and related technologies; the second, which is extremely popular at the time of writing, is the very different document-oriented approach of MongoDB.

XML databases

Although XML is a language rather than a database technology, it is part of a family of standards that have been used, since the late 1990s, to construct certain sorts of database-like services. The family includes technologies to describe data (XML); to specify a data dictionary (schemas and DTDs); to query the data (XPath); and to transform it (XSLT).

XML is a mark-up language similar to HTML. To be more precise, HTML (before HTML5) and XML are both simple forms of a parent language called SGML. HTML structures can also be written using XML, in a format called XHTML, which is easily handled by all web browsers.

As a mark-up language, XML consists of annotations to a textual document. The annotations generally take the form of tags – textual code placed between angle brackets < >. A tag can represent an element on its own:

```
<graphic url=”f23v.png” />
```

An element represents a logical component of a document – somewhat analogous to an object. In the case above, the element is a graphic element, with an attribute key-value that assigns the value f23v.png to the key url. ‘Empty’ elements like this have a slash before the close bracket.

An element can also contain text and other elements. In such cases, the text to be contained is introduced by an opening tag, which can provide attributes, and then it is followed by a closing tag, which is just the name of the element preceded by a slash:

```
The man in question was
   <name type="person"
      ref="http://dbpedia.org/page/Tim_Berners-Lee">Tim Berners-Lee</name>,
      who was at the time employed by
      <orgName ref="http://dbpedia.org/page/CERN">CERN</orgname>.
```

Note that whitespace is unimportant in the XML tags, but the importance of spaces and new lines in the main text will depend on the uses to which the text is put. It is common, however, to regard all whitespace as a single space, wherever it occurs. Tabs and new lines are then either generated automatically or indicated by empty elements (such as <br/> in HTML and XHTML).

Activity

Using the 'View Page Source' command on your browser, look at the HTML or XHTML of some of the websites you visit. Look at the sort of information that comes in the form of element names and the sort that is in attributes. Is there a pattern? How much of the useful information in the site is represented in machine-readable form (i.e. in tags) and how much is only in the textual content that is only easily comprehensible for humans?

Just these simple structures are enough to allow XML, like JSON, to be a useful way of passing information or data structures from one application to another, or to be used as a document file format – many office applications now save user documents as XML. However, unlike most interchange formats, and much more like a database, XML supports schemas.

The XML standard has two forms of correctness – if it has only legal characters and tags, and has all the correct header information, then it is called **well formed**. If a document includes a reference to a schema of some sort, then it can be tested for its **validity**. This latter is a way of ensuring that, even in an environment where a variety of users and software are editing the files directly, a structure is maintained that conforms strictly to a set of rules. Schemas are usually written in XML, and dictate the structures that must be in a document for it to make sense, along with the order in which they may appear, and so on.

These rules can be shared, by making them accessible online. They support interchange, but they also provide guarantees that make processing XML files more reliable.

XML-structured documents can be queried using a language called XPath. Since XML documents describe a tree structure, in which nodes are sorted by order of appearance in the file, they are easy to traverse. XPath can describe the structure as if it were a directory tree or URI, using forward slashes to separate nodes in the tree. A single slash at the beginning specifies the document root, while a double slash means that the query can be attached anywhere in the tree.

XPath is too complex a language, and has too different a paradigm from SQL, for us to consider it in depth, but for a simple example, consider an XML fragment, listing chemical elements:

```
<chemical-element>
   <symbol>H</symbol>
   <name>Hydrogen</name>
   <atomic-weight>1.008</atomic-weight>
   <isotopes>
      <isotope>H-1</isotope>
      <isotope>H-2</isotope>
   </isotopes>
</chemical-element>
```

...

To retrieve the chemical element with atomic weight less than 2, we could use the following XPath expression:

```
//chemical-element[atomic-weight<2]
```

To retrieve chemical elements with more than one isotope, the expression could be as follows:

```
//chemical-element[count(isotopes/isotope)>1]
```

The core XML technologies are designed primarily to support read-only interactions. If data is to be transformed, a new XML file is made in the process. The new file can still be used to replace the old one, simulating an update operation, but this is not the most common mode of usage. The language usually used for carrying out this process of transforming one document into another is called XSLT.

An XSLT file is essentially an XML template for the resulting file, along with transformation rules built around XPATH queries. For example, to create a simple XHTML list of all elements, we would normally put two components into the XSLT file. The first is the XHTML structure into which the generated content would fit and the call to add automatically generated content. For the sake of this example, we include only the surrounding list element:

```
<ul>
   <xsl:apply-templates select=”elements/*”>
</ul>
```

The definition for what goes between the list tags is a template. A template is a rule that is used if a particular criterion is matched in the source XML. In this case, we want to generate a list item for each chemical element:

```
<xsl:template match=”chemical-element”>
   <li><xsl:value-of select=”name/text()” /></li>
</xsl:template>
```

Combining XML, XPath and XSLT, it is possible to build database applications optimised for generating web pages. However, XML is a very verbose file format – a small amount of information can use a lot of disk space. Performance is also a problem, since software operating directly on data files will not have any of the optimisations of a fully-fledged DBMS.

To rectify these problems, numerous XML database systems have been developed. These use XML as an input format, structure data as trees and can be queried using XPath and XSLT. Such databases can be particularly popular for organisations that share data files frequently, and so need file copies that are easily taken and put into other systems.

Our first two examples of XML above use the schema defined by the Text Encoding Initiative (TEI), who were early adopters of the format. This group of, primarily, academics use the TEI and XML technologies to allow them to share their tools and data effectively even when the applications are quite disparate. TEI has been used for Welsh poetry, Chinese Buddhist texts and Italian music treatises, in each case with some specialised elements added, and some tools re-used.

Clearly, XML does not provide generic database technologies; however, in certain specific circumstances it can offer real benefits.

Activity

What do you think the * and text() in the examples of XSLT mean? Using the Further reading or online, check your answer to see if you were right.

MongoDB

MongoDB is an open-source, cross-platform database system, and uses sharding (horizontal partitioning), replication and load balancing to allow it to handle large data collections across multiple computers. Drivers are provided for multiple languages, and it has libraries for integration with many programming languages.

MongoDB provides a JavaScript shell for interacting with the database, and documents are provided using a simple textual format based on JSON. This makes developing MongoDB applications easier for those most comfortable with web technologies. In MongoDB terminology, documents are stored in collections, and the syntax for insertion is simple:

```
collection = [{"Symbol": "H", "Name": "Hydrogen",
           "Atomic weight": 1.008
           "Isotopes": ["H-1", "H-2"]},
           {"Symbol": "He", "Name": "Helium",
           "Atomic weight": 4.003,
           "Isotopes": ["He-3", "He-4"]},
           {"Symbol": "Li", "Name": "Lithium",
           "Atomic weight": 6.94,
           "Isotopes": ["Li-6"]}]
db.elements.insert(collection)
```

Querying the collection for exactly-matching values is relatively straightforward:

```
db.elements.find({"Symbol": "He"})
```

Approximate matching is similar, using JavaScript's regular expression object:

```
db.elements.find({"Symbol": /H*/})
```

Querying using operators is a little more clumsy – for example, greater than and less than signs must be spelt out:

```
db.elements.find({"Atomic weight": {"$lt": 5, "$gt": 1}})
```

This example retrieves all the documents describing elements with an atomic weight between 1 and 5. Documents can be modified using a similar syntax:

```
db.elements.update({"Symbol": "He"},
           {"$set": {"Atomic weight": 4.0026}})
```

In SQL, this update would look something like this:

```
UPDATE Elements SET AtomicWeight=4.0026 WHERE Symbol="He";
```

However, because the document-based model allows arrays and objects to be stored within a document, some structures and operations that would normally require joins in a relational model are much simpler. For example, to add an isotope to an element, we simply write:

```
db.elements.update({"Symbol": "Li"},
          {"$push": {"Isotopes": "Li-7"}})
```

In SQL, this would be likely to be implemented as:

```
INSERT INTO Isotopes (Symbol, IsotopeName) VALUES ("Li", "Li-7");
```

This is still fairly simple because Symbol is a unique identifier, and so no separate ID would be necessary. MongoDB creates its own unique identifiers, so the update command works even where the field is not necessarily unique. A more general SQL equivalent of the command would be more complex:

```
INSERT INTO Isotopes (ID, IsotopeName)
  SELECT ID, "Li-7" FROM Elements WHERE Symbol="Li";
```

MongoDB is fast, and supports systems with a need for high availability. These two attributes are available partly – as Brewer's conjecture would suggest – at the expense of some consistency.

For example, data replication uses a primary copy scheme (see Chapter 3 of the subject guide) in which applications that require guarantees of consistency use only the primary copy, but more speed-sensitive applications can use secondary copies as well, on the understanding that the data held there may no longer be correct. If the primary copy fails, one of the secondary copies will be promoted, potentially promoting a stale database state.

Another sacrifice made by MongoDB is in the power of its queries. Suppose that our elements collection contained other information about each element, including the melting and boiling point. We might want to search for the elements that sublime – that is, they go straight from solid to gaseous state without becoming a liquid. In SQL, this is easy:

```
SELECT * FROM Elements WHERE BoilingPoint<MeltingPoint;
```

However, in MongoDB, there is no way of directly comparing a value from the database with another until all values have been retrieved. The only options would be either to add a property to every document that stored the difference between the melting and boiling point (and then use that as a search criterion); or to retrieve all melting and boiling points and iterate over them in JavaScript.

We have also seen how MongoDB minimises the need for join-like structures by including one-to-many structures and even documents inside each document. This approach works well where the substructures represent dependent information, like the isotopes of an element. However, consider the case of a social-networking site. Here each user has a list of 'friends', each of whom is themselves a user:

```
users = [{"username": "john123", "joined": 12-04-09,
        "friends": ["paulMMMM", "RichardStarkey922"]},
        {"username": "paulMMMM", "joined": 12-03-08,
        "friends": ["George", "john123", "RichardStarkey922"]},
        …]
```

If we wanted to find out which of john123's friends was the first to join, the SQL would be a single command:

```
SELECT joined
   FROM users JOIN userFriends
   WHERE users.username = userFriends.friendname
      AND userFriends.username = "john123"
   ORDER BY joined ASC
   LIMIT 0,1;
```

In MongoDB, since there is no full join equivalent, we would need to retrieve the list of friends, then query for their earliest 'joined' date separately. However, MongoDB is still changing quite rapidly and as of version 3.2 it does have some support for joins. This support may well increase with time.

Of course, there is much more to MongoDB than we have covered here, it is a powerful piece of software with a large user base and is under very active development.

Activity

Download and install MongoDB. Create an Elements database and explore what you can do with it. Make a list of which aspects of it you find more intuitive than SQL databases and which you find less intuitive.

Column-oriented databases

Another Google development was a type of storage and retrieval system called BigTable. BigTable takes the opposite approach to MongoDB and XML, reducing both the complexity of the data and the functionality of the system in favour of speed and ease of distributed implementation. BigTable is designed for systems essentially represented as a single, large table, having so many columns, that most RDBMS implementations would not be able to cope. BigTable supports version information for each cell, so that the contents can change without losing previous data. This helps the system guarantee consistency.

One important aspect of the optimisation is that RDBMS implementations will tend to be designed on the expectation that each cell will have something in it. If some columns are usually empty or `NULL`, a database engineer would probably put those in separate tables. In BigTable, the table can be very sparse – many columns can be empty for many rows – without affecting performance or storage.

Each cell is retrieved by a row name, a column name and a timestamp. Columns are grouped into column families, which help the system organise the data. Although the number of columns can be huge and potentially unbounded, column families are expected to be more limited.

After Google published their description of this model, an open-source implementation was created for Hadoop called HBase. Apache also provided another layer called Phoenix which allows SQL operations to be run on HBase instances.

This sort of database only shows its advantages for very large, not too interconnected datasets requiring fast, live retrieval. At the time of writing, users of HBase include Facebook, LinkedIn, Netflix and Spotify.

4.6.3 Graph databases, the Semantic Web and Linked Data

The internet (or 'web') is a virtual space in which a vast amount of information is published. However, using that information can be tricky. Tim Berners-Lee, the inventor of HTML, always intended it to convey semantics – meaning – as well as just structure. The need for HTML to be very generic meant that the standard provided very little of that. Semantic web technologies are intended to address that shortcoming in as simple and generic a way possible.

At the core of the Semantic Web is the triple, a part of the Resource Description Framework (RDF). The triple communicates a single piece of information with (up to) three elements. Consider the statement that 'Hydrogen has an atomic weight of 1.008', or that 'Tim Berners-Lee was born on the 8th of June 1955'. In each case, we can say that two things or values are related by a property:

```
<Hydrogen>       <has an atomic weight of> <1.008>
<Tim Berners-Lee> <was born on>       <1955-06-08>
```

It is these elements that make up a triple – the thing that has the property, the property and the value of the property, called, respectively, **subject**, **predicate** and **object**. But there is still an important barrier to sharing this information usefully.

In order to use this information together with information from somewhere else on the web, we would need to know if any given one of these elements is the same as one from another triple. For example, given a triple from a different source:

```
<Tim Berners-Lee> <is the author of> <Weaving the Web>
```

How would we know that the same man who wrote this was born in 1955? He could just happen to have the same name. The solution to this problem – and the point at which this becomes Linked Data – is to use URIs (web addresses) rather than strings for each of the elements of the triple. For example, instead of Tim Berners-Lee, we could use the reference created by DBPedia (the Semantic Web version of Wikipedia): <http://dbpedia.org/resource/Tim_Berners-Lee>. If two statements about Tim Berners-Lee have the same URI, then they are referring to the same thing.

Similarly, *Weaving the Web* is a book, but there could be several books with that title. On the other hand, <http://dbpedia.org/resource/Weaving_The_Web> is very specific. If you follow that URI in your browser or using a linked data viewer, you will be able to check that it is indeed the book that is the subject of our triple above.

Perhaps less obviously, it is also useful to know that the property itself means the same thing. For example, if we wanted to find all books and their authors, we would need to retrieve all subjects and objects given a predicate that means <is the author of>. If the predicate use is a string chosen by the developer of a given database, then we have little hope of guessing it. If the predicate is a URI such as <http://dbpedia.org/ontology/author>, then we can combine our data more easily.

Since RDF, including the triple, is a data model, it is not tied to a specific file format. There exist several serialisations (ways to write it to a file or stream), of which the simplest is Notation 3, which is usually shortened to N3. In N3, we could state:

```
<http://dbpedia.org/resource/Tim_Berners-Lee>
   <http://dbpedia.org/ontology/author>
      <http://dbpedia.org/resource/Weaving_The_Web>.
```

N3 supports prefixes, so this can be made much easier to read. We can also add several statements with the same subject, at which point, the prefix notation makes the format more compact as well.

```
@prefix dbr: <http://dbpedia.org/resource/>
@prefix dbo: <http://dbpedia.org/ontology/>
<dbr:Tim_Berners-Lee>
   <dbo:author> <dbr:Weaving_The_Web>;
   <dbo:birthDate> "1955-06-08"^^xsd:date;
   <dbo:birthName> "Timothy John Berners-Lee" .

<dbr:Weaving_The_Web>
   <dbo:name> "Weaving the Web: The Original Design and Ultimate Destiny
   of the World Wide Web by its inventor".
```

Another way of sharing RDF triples is to place them directly into a webpage, using a form of RDF called RDFa:

```
<div prefix="dbr: http://dbpedia.org/resource/
         dbo: http://dbpedia.org/ontology/"
      resource="dbr:Tim_Berners-Lee">
   Born <span property="dbo:birthName">Timothy John
   Berners-Lee</span>, in 1999, Tim Berners-Lee wrote
   <span property="dbo:author">Weaving the Web</span>
</div>
```

Unlike XML or HTML, the structure created by RDF triples is not a tree but a directed graph. Figure 4.5 shows an example graph generated from RDF triples.

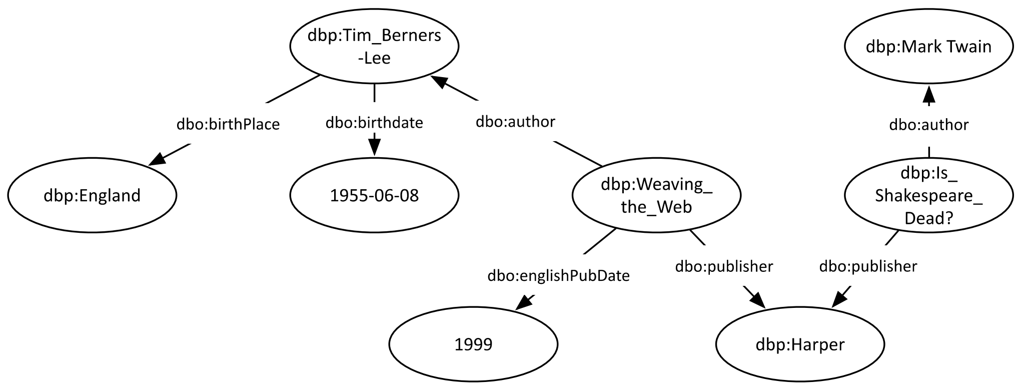


Figure 4.5. A graph representing a set of triples. Triples represented here are as published by dbpedia.org.

As with XML, there exist both query languages and databases specifically for the data structure. Databases for storing triples are, logically, called a **triplestore**. The dominant query language is called SPARQL. SPARQL queries are executed by a SPARQL **endpoint**. The endpoint may be an interface to an RDF database or it may query the web, or even, given appropriate mapping information, provide an interface to a relational database.

The querying syntax for SPARQL is based on SQL, and can be expressed in N3. For example, to retrieve all authors and titles of books published by Harper:

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?author ?title
WHERE {
   ?author dbo:author    ?x.
```

```
     ?x      dbo:name     ?title.
     ?x      dbo:publisher dbr:Harper_(publisher).
  }
```

This syntax lets the user specify the shape of the triples that are required, with gaps in the form of variables (specified with question marks). Any shape in the graph that fits the pattern will be returned as a match.

So far, we have introduced **Linked Data** – the name is justified by the use of URIs to connect triples – but we have not explained how the web that results might be a **Semantic Web**. The intention behind the creation of the Semantic Web was not only to create a method of publishing data in a way that could become part of a vast distributed database, but also to allow for the resulting database to be deductive, in the sense that a machine could reason about the information provided.

When we introduced the logic of deductive databases, we noted that they need deductive axioms as well as ground axioms – not just facts, but information about how those facts should be interpreted. In the Semantic Web, axioms are generally published in the form of **ontologies**. These ontologies, like XML schemas, define the types that a system can use. Just as XML schemas can be written in XML, Semantic Web ontologies are usually written in RDF, in a language called **OWL**.

Activity

Visit `http://dbpedia.org/ontology/` and explore the ontology there. Look at the concepts that are defined and what information is provided about them. Consider the examples we have given above – when does DBPedia use URIs in the `/resource` path, and when does it use ontology URIs?

Figure 4.6 shows the DBPedia ontology entry (at the time of writing) for `publisher` at the URI `http://dbpedia.org/ontology/publisher`. Although some of the content is repetitive or tautologous (it states what is obviously true, such as `<dbo:publisher> <owl:sameas> <dbo:publisher>`), other parts are more interesting – it is a property rather than a class, it indicates that the subject and object 'coparticipate' with one another, it is called *Heraugeber* in German, it can only have a company as its object, and so on.

Note: Linked data relies on URIs to remain the same or, if they change, for the old address to point to the new one. Despite this, there is nothing technological that stops pages being moved or removed from the URI referred to. Although all the addresses we include here were correct at the time of publication, they may have changed since. Making systems that are robust to missing or incorrect data is an important part of the linked data approach. The assumption that this may happen at any time is a particular instance of the 'open-world assumption'.

```
dbo:publisher   rdf:type             owl:ObjectProperty ,
                         rdf:Property ;
     rdfs:subPropertyOf     ns4:coparticipatesWith ;
     owl:equivalentProperty ns5:publisher ;
     owl:sameAs          dbo:publisher ;
     rdfs:label          "Herausgeber"@de ,
        "\u03B5\u03BA\u03B4\u03CC\u03C4\u03B7\u03C2"@el ,
                         "publisher"@en ,
                         "uitgever"@nl ;
```

```
    rdfs:range          dbo:Company ;
    rdfs:isDefinedBy    dbo: ;
    wdrs:describedby
  <http://dbpedia.org/ontology/data/definitions.ttl> ;
    ns7:wasDerivedFrom    ns8:publisher ;
    ns9:defines           dbo:publisher .
<http://dbpedia.org/ontology/data/definitions.ttl>   ns9:describes dbo:publisher .
```

Figure 4.6. Publisher in the DBPedia ontology.

Please note that in Figure 4.6 prefix declarations have been omitted to make the code more readable. They are:

rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>, **dbo:** <http://dbpedia.org/ontology/>, **owl:** <http://www.w3.org/2002/07/owl#>, **rdfs:** <http://www.w3.org/2000/01/rdf-schema#>, **ns4:** <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>, **ns5:** <http://schema.org/>, **wdrs:** <http://www.w3.org/2007/05/powder-s#>, **ns7**: <http://www.w3.org/ns/prov#>, **ns8:** <http://mappings.dbpedia.org/index.php/OntologyProperty:>, **ns9:** <http://open.vocab.org/terms/>

The ability to specify that a publisher is a specific case of co-participation is similar to the ability to specify that a parent is an ancestor – it paves the way for further deduction, and also allows rules declared for co-participants to be applied automatically for the publisher. The Semantic Web supports a range of powerful ways of specifying rules, including the full range of a knowledge representation logic called **description logic**.

The examples above are all from DBPedia, because it is not only a very large source of RDF, but also because many other publishers of triples use DBPedia URIs, or to put it another way, DBPedia is highly connected to other RDF sources. Some other important sources of RDF include:

- **http://MusicBrainz.org**. The information provided to accompany digital music, especially when it is taken from a CD, is generally provided through an online database service, most notably the CDDB. These services are generally commercial, and their data is not very rich. MusicBrainz started off as a free, open, user-contributed CD metadata source, but has expanded to a large resource of information about music and recordings. It is implemented as a PostrgreSQL database in the background, but data is made available in various ways, including as Linked Data, and provides a useful point of reference for other music-related projects. Another significant source of music-related RDF is: www.last.fm
- **http://greek-lod.math.auth.gr**. The Greek government has pushed strongly for transparency for its governance and the functioning of its emergency services and, as a result, they were quick to publish information in an open way. Every incident handled by the fire brigade and police is logged and published as linked open data. Although, at the time of writing, the government transparency programme, Diavgeia, was unavailable, and had changed URI a few times, it has published information about legislation and government spending for several years. A comparable source of governmental information for the UK is: https://data.gov.uk
- **http://viaf.org**. Libraries, particularly national libraries, are a good source of catalogue information about their books and other archives. Each country usually maintains a list – called an authority file – that standardises how to record a given entity – such as a person, place or book – in the catalogue. Since the rise of the Semantic Web, those authority files are published so that each entity has a URI. VIAF unites all the separate authorities, so that the different catalogues can be unified.

- **www.bbc.co.uk**. The British Broadcasting Corporation uses linked data to generate some of its web pages by collecting information from other sources to present to the user. They also provide information about the music and programmes that they broadcast on the radio and television, and also the results of sporting events. Since they benefit from the connections to other sources, they are particularly useful for joining up, for example, DBPedia with music information resources such as musicbrainz and last.fm.
- **http://bio2rdf.org**. Bio2RDF aggregates information from various public datasets in the life sciences, particularly medicine and biochemistry, and publishes it as RDF. It includes information about important organic molecules, medicines, and published research in the field.

What the Linked Data technologies provide for all these services is the ability not only to make their own data available but also to use the data of others or to incorporate it in their own. This must be the primary reason behind any decision to use these technologies rather than more conventional RDBMS approaches – they have information sharing as a fundamental part of how they work.

4.7 Conclusions

There are many different ways of organising and retrieving data but, although they may look very different in their approaches, there are a set of concepts that unite them all. For any given technology, the important considerations will still include the following.

- How does it behave with different amounts of data?
- How does it behave with different complexities of data?
- How is it affected by the balance of queries, inserts and updates?
- Does a query necessitate looking at every row of a table?
- How is concurrency handled? Are ACID properties preserved?
- If inconsistency is possible, how is it resolved?
- How is recovery handled?
- Is security robust?
- What tools are available to support its adoption?

Although the world of database systems changes quickly, especially for Big Data and web applications, the approach to evaluating them is still largely the same.

We hope that this subject guide has provided you with the skills and knowledge you need to make decisions about which technologies to try out, and which might be best for any projects you undertake in the future.

4.8 Overview of the chapter

In this chapter, we reviewed and evaluated criticisms of the relational model, SQL and RDBMS implementations. Following this, we introduced deductive and object-oriented database approaches, both of which have been held up as competing models to the relational model. More recent approaches to data systems were considered under three broad categories: key-value databases, document-oriented databases and graph databases. We provided example models and applications for each of these categories, including MapReduce implementations, XML databases, MongoDB, BigTable and the Resource Description Framework.

4.9 Reminder of learning outcomes – concepts

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

- discuss and evaluate critically the real and perceived shortcomings of the relational model and SQL
- discuss the deductive database approach
- discuss the OO database approach
- discuss alternative approaches to database systems, including:
    - deductive databases
    - object-oriented databases
    - key-value databases
    - document-oriented databases
    - the Semantic Web
- give at least one example of each approach
- discuss at least one alternative approach in greater detail
- explain how to decide between the various approaches and models for a given use case.

4.10 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:

- Description Logic
- MapReduce
- NoSQL
- Semantic Web
    - Subject/Object/Predicate
    - Linked Data
    - Triplestore
    - (Web) Ontology
- Semi-structured data.

4.11 Test your knowledge and understanding

4.11.1 Sample examination questions

Consider the following table:

PlanetaryBody	DayLength	YearLength	LiquidWater	Moons	
				Name	LiquidWater
Mercury	4,223	88	No	–	–
Venus	2,802	225	No	–	–
Earth	24	365	Yes	Moon	No
Mars	25	687	Maybe	Phobos Deimos	No No
Jupiter	10	4,331	No	Io Europa Ganymede Callisto …	Maybe Yes Yes Maybe
Saturn	11	10,747	No	Titan Calypso Hyperion Rhea Iapetus Enceladus …	No Maybe No Maybe No Yes
Uranus	17	30,589	No	Titania Oberon …	No No
Neptune	16	59,800	No	Triton …	No
Pluto	153	90,560	Maybe	Charon …	No

a. What would you change to put this into 1NF? [2]

b. Identify the functional dependencies and normalise the table. [3]

c. A friend says that this data is better suited to a document-oriented database system.

   i. What is a document-oriented database system? [2]

   ii. What are the arguments for and against her observation? [3]

   iii. Choose one other non-relational data management approach and give the arguments for and against using that for this data. [5]

   iv. The information is due to be expanded to include slightly more than 700,000 planets, minor planets and moons from our solar system, along with approximately 5,000 planets from other star systems. How does the scale of this information affect your evaluation of the options for the databases? Justify your answer. [6]

   v. What further information about the system or its use would affect your decision? What aspects of your database system design would the extra information change? [4]

Appendix 1: Sample answers/ Marking scheme

Chapter 2: Data preservation, security and database optimisation

a.

i. A schedule is an order of execution [1] for a series of concurrent operations [1]. It is considered to be serialisable if there exists a serial (i.e. non-concurrent) ordering of the transactions involved [1] that is guaranteed to have the same outcome [1].

ii. Atomicity [1] – logically-grouped operations (transactions) are executed either entirely or not at all [1]. Consistency [1] – The database never gives the appearance of an inconsistent state. Isolation [1] – Concurrent transactions should not affect each other's operation (i.e. they should behave as if they were not concurrent) [1]. Durability [1] – Once a transaction has been committed, that data is part of the database and remains there; until then, it is not considered part of the database at all. [1]

iii. A transaction is a group of database operations that form a single logical operational unit [1]. Without the concept of a transaction, the individual related operations could be separated, and result in various database anomalies and loss of data [1]. By grouping the operations together, the transaction allows them to be treated as one [1]. ACID properties and serialisation pertain to the transaction, not the separate operations, because that ensures that the logic of the user's interaction with a database is reliable [1]. [Up to 1 mark for the definition and up to 2 for the remainder]

b.

i. `GRANT SELECT ON` Auryn `TO` Bastian; [2 marks, 1 if roughly correct]

ii. `GRANT UPDATE` (Name) `ON` Auryn `TO` Bastian; [1 if correct or the part that differs from i. is correct, 0 otherwise]

c. Beware of SQL Injection attacks [1] where code is inserted as well as a query string [1]. [Max 1 mark from these]

Take precautions such as escaping all ways of terminating a quote [1] or carefully validating the input [1]. [Max 1 mark]

d.

i. An index is a data structure [1] that speeds up retrieval of data [1] from a database [1]. [Max 2]

ii. An index is always generated for the primary key [1] (it is usually automatically generated for foreign keys as well).

iii. Indexes are useful where a subset of the rows will usually be retrieved from the table [1]; and the columns to be indexed will be part of the `WHERE` clause] [1]; or where results will be sorted using the columns [1]; or where the columns are a candidate or foreign key [1]. [Max 2]

Chapter 3: Distributed architectures for database systems

a.

i. A distributed database system is a collection of local database systems [1]; where sites are interconnected [1]; but can be used independently [1]; and which, to a user, look like a single database [1]. [Max 3]

ii. Any reasonable answer. Might include network characteristics, scale of data, very partitioned data with little need for interrogating between departments, etc.

iii. In a homogeneous distributed database system, all the local DBMSs are the same DBMS software [1] of the same version [1] running on the same OS [1]. [Max 2 for these]

It is likely to be preferred because it is easier to build a DDBMS where all the parts have identical behaviour [1], and because maintenance only requires knowledge of one system [1]. [Max 1 for these]

b.

i. A deadlock is a situation in concurrency control. It occurs where each concurrent transaction in a set of transactions needs a lock on a table. Each transaction also holds a lock on a different table that it will not release until it completes. If transaction 1 depends on transaction 2 to release a lock, while transaction 2 is waiting for transaction 1, then the situation is in deadlock. More transactions can be involved, in which case the lock dependencies form a cyclical chain. It results in all transactions being held in a waiting state indefinitely. [4 marks for something that makes sense and covers the main points – this is a difficult topic to explain clearly]

ii. A wait-for graph is a directed graph [1] showing transactions dependent on others to release locks [1]. For an example, see Figure 2.13 in the subject guide. [2 marks for the diagram – 1 for something that looks like a directed graph; 1 for a correct deadlock, with appropriate arrow directions]

iii. A local DBMS can easily maintain a wait-for graph for its transactions, but to spot deadlocks on a DDBMS, there must be a global wait-for graph. This either requires a centralised controller that maintains the graph, which acts against the objective of having no single point of failure, or it requires some sort of replicated graph across the system, which increases network traffic and system complexity. [3 marks for a sensible answer that raises the issues and mentions options]

c. Brewer's conjecture discusses three desirable properties: Consistency [1]; that the DBMS should eventually reach a consistent state within itself [1]; Availability [1]; that any request received by a functioning node should result in a response; and Partition tolerance [1]; that if the network is disrupted, the system should still function and become consistent again when the network is repaired [1]. Brewer states that the three can never be achieved perfectly – that there is always some amount of compromise [1]. This is usually seen as tension between C and A [1]. [Max 7. Sensible points other than these should receive credit as appropriate]

Chapter 4: Advanced database systems

a. The moon's name and LiquidWater values are non-scalar [1]. In 1NF, they would have to be split (and the columns renamed Moon and MoonWater, to avoid duplication); so, for example, there would be two rows for Mars: {Mars, 25, 687, Maybe, Phobos, No} and {Mars, 25, 687, Maybe, Deimos, No} [1]

b. {PlanetaryBody} -> {DayLength, YearLength, LiquidWater}

   {PlanetaryBody} -> {MoonName}

   {MoonName} -> {MoonWater}

   So to normalise (to 5NF), all that is needed is to create two tables: PlanetaryBodies {PlanetaryBody, DayLength, YearLength, LiquidWater} and Moons {PlanetaryBody, MoonName, LiquidWater}. [This assumes that moons orbit around exactly one planet.] [3]

c.

   i. A document-oriented database system is one which stores the unstructured or semi-structured data of a set of documents with relatively little important connection between them. [2]

   ii. The structure of this data is appropriate, since there is no relationship between planets or moons, and the multiple moons to a planet would fit naturally within an object-type storage as an array. This would result in a simpler model. However, there are no obvious functional differences for this case, and the table is too small for any of the advantages of, say, MongoDB, for distributed use, to make a difference. [Max 3]

   iii. [Here is one example – many are possible] The data could be modelled easily using Linked Data either in a triplestore, or simply published as rdf files. The LiquidWater property in particular is clearly the same predicate, semantically, being applied to two different types of object – something that Linked Data would express more neatly. On the other hand, there is little in the way of network here, and many of the linking aspects of the technologies do not seem to be relevant here. [5]

   iv. Each row in PlanetaryBody is likely to be <20 bytes for the name, 2 bytes for DayLength, 3 bytes for YearLength and 2 bits for LiquidWater, giving <30 bytes in total. About 700,000 rows at that size would be about 20 or so megabytes. This is easily held in memory or passed down a network connection completely; although normally, you would expect only to send a subset. This means that the quantity of data **on its own** is unlikely to make much difference to the choice of system to use. [6 marks for anything well-reasoned and that uses the information given]

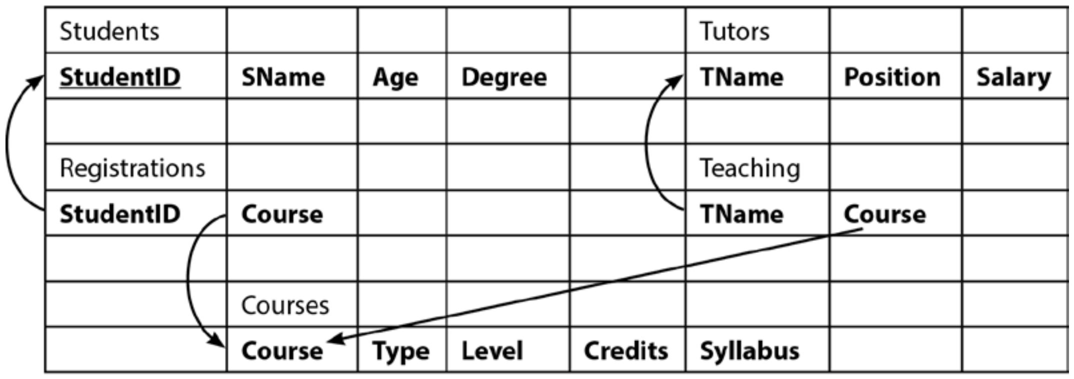
   v. Given the relatively small size of the data (and low likelihood of joins), a more important question is how it will be accessed. On the assumption that it will change relatively slowly, probably the issue would be how users access it and whether it relates to anything else on the system. Since the data is easily partitionable and rarely updated, any distributed system could be used for heavy demand, but if there is limited demand, then a simple system would be likely to be easier to maintain. [Up to 4 for any sensible answer that relates to use/queries rather than data size]

Notes

Appendix 2: Data tables

These tables are provided to act as a starting point for exploring databases in practice. They are also referred to by various examples, starting in Chapter 4 of Volume 1 of this guide. There are few enough rows that you could put them into your own DBMS manually. However, electronic versions of the dataset can be downloaded from the VLE for importing directly into your database system.

The figure below summarises the tables and their rows, along with their relations.



`Students` table

`Primary key: StudentID`

StudentID	SName	Age	Degree
14021	Vincente Tomi	29	Politics
14022	Yuan Tseh Lee	21	Chemistry
15196	Enayat Khan	28	Music
21260	Sim Wong Hoo	19	Computer Science
23123	Jeremy Bentham	19	Computer Science
24412	John Ferrabosco	18	Music
26711	Margaret Chan	18	Medicine
14993	Grace Hopper	20	Computer Science
15491	Frances Spence	19	Computer Science
39841	Ada Lovelace	19	Computer Science
16411	Maithili Thevar	20	Music
21311	Jane Smith	18	Physics
26921	Jane Smith	18	Biology

`Tutors` table

`Primary key: TName`

TName	Position	Salary
Soyun Park	Professor	45000
Yasmine Waugh	Reader	34000
Ani Rai	Senior Lecturer	32010
Mark Taylor	Lecturer	27823
Matthew Ridley	Lecturer	18050
Selene Maughan	Lecturer	22000
Andrea Mendoza	Senior Lecturer	32010
Mycroft Stuart	Lecturer	27823
Mohammed Saldin	Lecturer	18050

`Courses` table

`Primary key: Course`

Course	Type	Level	Credits	Syllabus
Programming1	2sem	1	30	CS/Programming1.pdf
Databases1	1sem	1	15	CS/Databases1.pdf
VertebrateDiversity	1sem	2	15	B/Vert.pdf
Databases2	1sem	2	15	CS/Databases2.pdf
MusicInformationRetrieval	proj	3	30	CS/MIR.pdf
DemocracyAndLabour	1sem	2	15	Politics/DandL.pdf
OrganicChemistry1	1sem	1	15	Chem/Organic1.pdf
MusicalPerformance1	2sem	1	15	Mus/Performance.pdf
MusicalComposition1	portfolio	1	15	Mus/Composition.pdf
Epidemiology1	1sem	2	15	Med/Epi.pdf
Anatomy2	2sem	2	30	Med/Anatomy.pdf
MusicAndDance	1sem	2	15	Mus/Music-and-dance.pdf
AdvancedMechanics	1sem	3	15	Phys/Mech2.pdf

`Registrations` table

`Primary key: (StudentID, Course)`

`Foreign key: StudentID REFERENCES Students`

`Foreign key: Course REFERENCES Courses`

StudentID	Course
21260	Programming1
23123	Databases1
26921	VertebrateDiversity
23123	Programming1
21260	Databases1
14993	Databases2
14993	MusicInformationRetrieval
14021	DemocracyAndLabour
14022	OrganicChemistry1
15196	MusicalPerformance1
24412	MusicalComposition1
26711	Epidemiology1
26711	Anatomy2
16411	MusicAndDance
16411	MusicalPerformance1
21311	AdvancedMechanics
39841	MusicInformationRetrieval

`Teaching` table

`Primary key: (TName, Course)`

`Foreign key: TName REFERENCES Tutors`

`Foreign key: Course REFERENCES Courses`

TName	Course
Mark Taylor	Programming1
Mohammed Saldin	Databases1
Matthew Ridley	VertebrateDiversity
Mark Taylow	Databases2
Mohammed Saldin	MusicInformationRetrieval
Selene Maughan	DemocracyAndLabour
Soyun Park	OrganicChemistry1
Ani Rai	MusicalPerformance1
Mycroft Stuart	MusicalComposition1
Yasmine Waugh	Epidemiology1
Yasmine Waugh	Anatomy2
Ani Rai	MusicAndDance
Andrea Mendoz	AdvancedMechanics

Notes