Chapter 1. Introduction to the Module

Table of contents

The purpose of this chapter is to introduce the fundamental concepts of database systems. Like most areas of computing, database systems have a significant number of terms and concepts that are likely to be new to you. We encourage you to discuss these terms in tutorials and online with one another, and to share any previous experience of database systems that you may have. The module covers a wide range of issues associated with database systems, from the stages and techniques used in the development of database applications, through to the administration of complex database environments. The overall aim of the module is to equip you with the knowledge required to be a valuable

member of a team, or to work individually, in the areas of database application development or administration. In addition, some coverage of current research areas is provided, partly as a stimulus for possible future dissertation topics, and also to provide an awareness of possible future developments within the database arena.

Module objectives

At the end of this module you will have acquired practical and theoretical knowledge and skills relating to modern database systems. The module is designed so that this knowledge will be applicable across a wide variety of database environments. At the end of the module you will be able to:

- Understand and explain the key ideas underlying database systems and the database approach to information storage and manipulation.
- Design and implement database applications.
- Carry out actions to improve the performance of existing database applications.
- Understand the issues involved in providing multiple users concurrent access to database systems.
- Be able to design adequate backup, recovery and security measures for a database installation, and understand the facilities provided by typical database systems to support these tasks.
- Understand the types of tasks involved in database administration and the facilities provided in a typical database system to support these tasks.
- Be able to describe the issues and objectives in a range of areas of contemporary database research.

Chapter objectives

At the end of this chapter you should be able to:

- Explain the advantages of a database approach for information storage and retrieval.
- Explain the concepts of physical and logical data independence, and describe both technically and in business terms the advantages that these concepts provide in Information Systems development.
- Understand the basic terminology and constructs of the Relational approach to database systems.

Introduction

In parallel with this chapter, you should read Chapter 1 and Chapter 2 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter sets the scene for all of the forthcoming chapters of the module. We begin by examining the approach to storing and processing data that was used before the arrival of database systems, and that is still appropriate today in certain situations (which will be explained). We then go on to examine the difference between this traditional, file-based approach to data storage, and that of the database approach. We do this first by examining inherent limitations of the file-based approach, and then discuss ways in which the database approach can be used to overcome these limitations.

A particular model of database systems, known as the Relational model, has been the dominant approach in the database industry since the early '80s. There are now important rivals and extensions to the Relational model, which will be examined in later chapters, but the Relational model remains the core technology on which the database industry worldwide is based, and for this reason this model will be central to the entire module.

Motivation for data storage

Day-to-day business processes executed by individuals and organisations require both present and historical data. Therefore, data storage is essential for organisations and individuals. Data supports business functions and aids in business decision-making. Below are some of the examples where data storage supports business functions.

Social media

Social media has become very popular in the 21st century. We access social media using our computers and mobile phones. Every time we access social media, we interact, collaborate and share content with other people. The owners of social media platforms store the data we produce.

Supermarket

A supermarket stores different types of information about its products, such as quantity, prices and type of product. Every time we buy anything from the supermarket, quantities must be reduced and the sales information must be stored.

Company

A company will need to hold details of its staff, customers, products, suppliers and financial transactions.

If there are a small number of records to be kept, and these do not need to be changed very often, a card index might be all that is required. However, where there is a high volume of data, and a need to manipulate this data on a regular basis, a computer-based solution will often be chosen. This might sound like a simple solution, but there are a number of different approaches that could be taken.

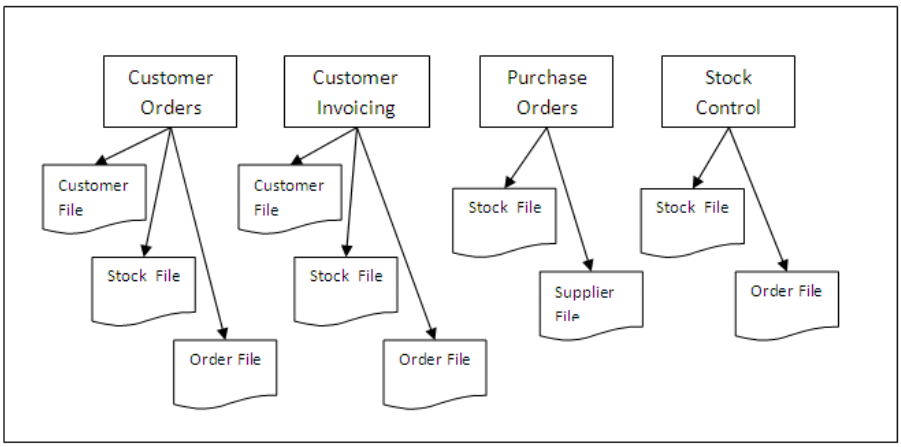
Traditional file-based approach

The term 'file-based approach' refers to the situation where data is stored in one or more separate computer files defined and managed by different application programs. Typically, for example, the details of customers may be stored in one file, orders in another, etc. Computer programs access the stored files to perform the various tasks required by the business. Each program, or sometimes a related set of programs, is called a computer application. For example, all of the programs associated with processing customers' orders are referred to as the order processing application. The file-based approach might have application programs that deal with purchase orders, invoices, sales and marketing, suppliers, customers, employees, and so on.

Limitations

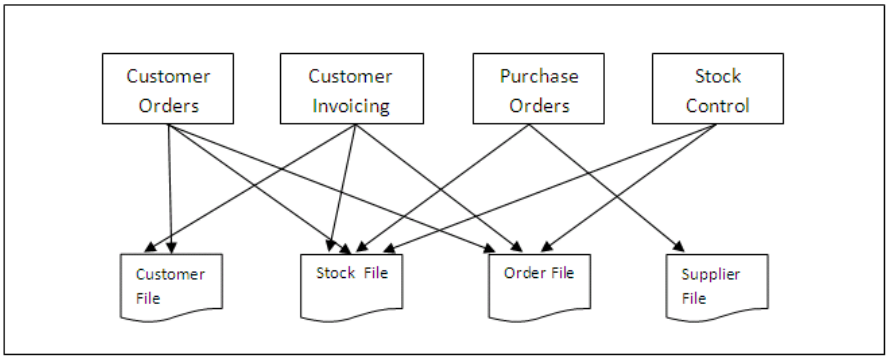
- Data duplication: Each program stores its own separate files. If the same data is to be accessed by different programs, then each program must store its own copy of the same data.
- Data inconsistency: If the data is kept in different files, there could be problems when an item of data needs updating, as it will need to be updated in all the relevant files; if this is not done, the data will be inconsistent, and this could lead to errors.
- Difficult to implement data security: Data is stored in different files by different application programs. This makes it difficult and expensive to implement organisation-wide security procedures on the data.

The following diagram shows how different applications will each have their own copy of the files they need in order to carry out the activities for which they are responsible:



The shared file approach

One approach to solving the problem of each application having its own set of files is to share files between different applications. This will alleviate the problem of duplication and inconsistent data between different applications, and is illustrated in the diagram below:



The introduction of shared files solves the problem of duplication and inconsistent data across different versions of the same file held by different departments, but other problems may emerge, including:

- File incompatibility: When each department had its own version of a file for processing, each department could ensure that the structure of the file suited their specific application. If departments have to share files, the file structure that suits one department might not suit another. For

example, data might need to be sorted in a different sequence for different applications (for instance, customer details could be stored in alphabetical order, or numerical order, or ascending or descending order of customer number).

- Difficult to control access: Some applications may require access to more data than others; for instance, a credit control application will need access to customer credit limit information, whereas a delivery note printing application will only need access to customer name and address details. The file will still need to contain the additional information to support the application that requires it.
- Physical data dependence: If the structure of the data file needs to be changed in some way (for example, to reflect a change in currency), this alteration will need to be reflected in all application programs that use that data file. This problem is known as physical data dependence, and will be examined in more detail later in the chapter.
- Difficult to implement concurrency: While a data file is being processed by one application, the file will not be available for other applications or for ad hoc queries. This is because, if more than one application is allowed to alter data in a file at one time, serious problems can arise in ensuring that the updates made by each application do not clash with one another. This issue of ensuring consistent, concurrent updating of information is an extremely important one, and is dealt with in detail for database systems in the chapter on concurrency control. File-based systems avoid these problems by not allowing more than one application to access a file at one time.

Review question 1

What is meant by the file-based approach to storing data? Describe some of the disadvantages of this approach.

Review question 2

How can some of the problems of the file-based approach to data storage be avoided?

Review question 3

What are the problems that remain with the shared file approach?

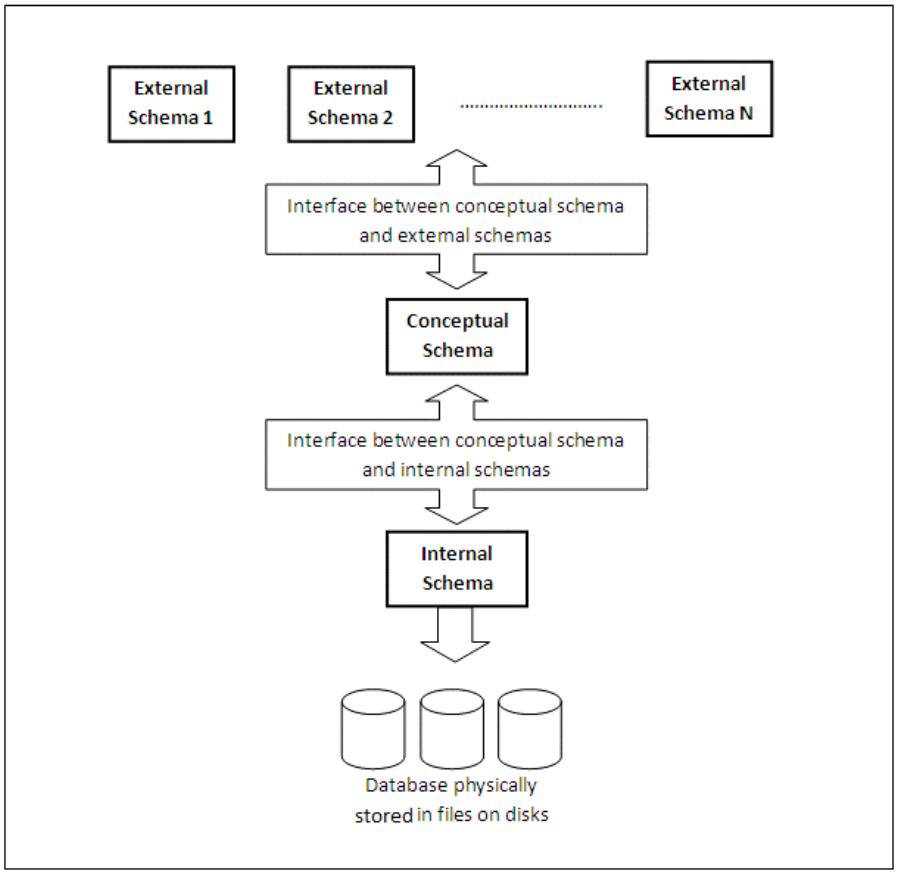
The database approach

The database approach is an improvement on the shared file solution as the use of a database management system (DBMS) provides facilities for querying, data security and integrity, and allows simultaneous access to data by a number of different users. At this point we should explain some important terminology:

- **Database:** A database is a collection of related data.
- **Database management system:** The term 'database management system', often abbreviated to DBMS, refers to a software system used to create and manage databases. The software of such systems is complex, consisting of a number of different components, which are described later in this chapter. The term database system is usually an alternative term for database management system.
- **System catalogue/Data dictionary:** The description of the data in the database management system.
- **Database application:** Database application refers to a program, or related set of programs, which use the database management system to perform the computer-related tasks of a particular business function, such as order processing.

One of the benefits of the database approach is that the problem of physical data dependence is resolved; this means that the underlying structure of a data file can be changed without the application programs needing amendment. This is achieved by a hierarchy of levels of data specification. Each such specification of data in a database system is called a schema. The different levels of schema provided in database systems are described below. Further details of what is included within each specific schema are discussed later in the chapter.

The Systems Planning and Requirements Committee of the American National Standards Institute encapsulated the concept of schema in its three-level database architecture model, known as the ANSI/SPARC architecture, which is shown in the diagram below:



ANSI/SPARC three-level architecture

ANSI = American National Standards Institute

ANSI/X3 = Committee on Computers and Information Processing

SPARC = Standards Planning and Requirements Committee

The ANSI/SPARC model is a three-level database architecture with a hierarchy of levels, from the users and their applications at the top, down to the physical storage of data at the bottom. The characteristics of each level, represented by a schema, are now described.

The external schema

The external schemas describe the database as it is seen by the user, and the

user applications. The external schema maps onto the conceptual schema, which is described below.

There may be many external schemas, each reflecting a simplified model of the world, as seen by particular applications. External schemas may be modified, or new ones created, without the need to make alterations to the physical storage of data. The interface between the external schema and the conceptual schema can be amended to accommodate any such changes.

The external schema allows the application programs to see as much of the data as they require, while excluding other items that are not relevant to that application. In this way, the external schema provides a view of the data that corresponds to the nature of each task.

The external schema is more than a subset of the conceptual schema. While items in the external schema must be derivable from the conceptual schema, this could be a complicated process, involving computation and other activities.

The conceptual schema

The conceptual schema describes the universe of interest to the users of the database system. For a company, for example, it would provide a description of all of the data required to be stored in a database system. From this organisation-wide description of the data, external schemas can be derived to provide the data for specific users or to support particular tasks.

At the level of the conceptual schema we are concerned with the data itself, rather than storage or the way data is physically accessed on disk. The definition of storage and access details is the preserve of the internal schema.

The internal schema

A database will have only one internal schema, which contains definitions of the way in which data is physically stored. The interface between the internal schema and the conceptual schema identifies how an element in the conceptual schema is stored, and how it may be accessed.

If the internal schema is changed, this will need to be addressed in the interface between the internal and the conceptual schemas, but the conceptual and external schemas will not need to change. This means that changes in physical storage devices such as disks, and changes in the way files are organised on storage devices, are transparent to users and application programs.

In distinguishing between 'logical' and 'physical' views of a system, it should be noted that the difference could depend on the nature of the user. While 'logical' describes the user angle, and 'physical' relates to the computer view, database designers may regard relations (for staff records) as logical and the database

itself as physical. This may contrast with the perspective of a systems programmer, who may consider data files as logical in concept, but their implementation on magnetic disks in cylinders, tracks and sectors as physical.

Physical data independence

In a database environment, if there is a requirement to change the structure of a particular file of data held on disk, this will be recorded in the internal schema. The interface between the internal schema and the conceptual schema will be amended to reflect this, but there will be no need to change the external schema. This means that any such change of physical data storage is not transparent to users and application programs. This approach removes the problem of physical data dependence.

Logical data independence

Any changes to the conceptual schema can be isolated from the external schema and the internal schema; such changes will be reflected in the interface between the conceptual schema and the other levels. This achieves logical data independence. What this means, effectively, is that changes can be made at the conceptual level, where the overall model of an organisation's data is specified, and these changes can be made independently of both the physical storage level, and the external level seen by individual users. The changes are handled by the interfaces between the conceptual, middle layer, and the physical and external layers.

Review question 4

What are some of the advantages of the database approach compared to the shared file approach of storing data?

Review question 5

Distinguish between the terms 'external schema', 'conceptual schema' and 'internal schema'.

Components of a DBMS

The major components of a DBMS are as follows:

DBMS engine

The engine is the central component of a DBMS. This component provides access to the database and coordinates all of the functional elements of the DBMS. An important source of data for the DBMS engine, and the database system as a whole, is known as metadata. Metadata means data about data. Metadata is

contained in a part of the DBMS called the data dictionary (described below), and is a key source of information to guide the processes of the DBMS engine. The DBMS engine receives logical requests for data (and metadata) from human users and from applications, determines the secondary storage location (i.e. the disk address of the requested data), and issues physical input/output requests to the computer operating system. The data requested is fetched from physical storage into computer main memory; it is contained in special data structures provided by the DBMS. While the data remains in memory, it is managed by the DBMS engine. Additional data structures are created by the database system itself, or by users of the system, in order to provide rapid access to data being processed by the system. These data structures include indexes to speed up access to the data, buffer areas into which particular types of data are retrieved, lists of free space, etc. The management of these additional data structures is also carried out by the DBMS engine.

User interface subsystem

The interface subsystem provides facilities for users and applications to access the various components of the DBMS. Most DBMS products provide a range of languages and other interfaces, since the system will be used both by programmers (or other technical persons) and by users with little or no programming experience. Some of the typical interfaces to a DBMS are the following:

- A data definition language (or data sublanguage), which is used to define, modify or remove database structures such as records, tables, files and views.
- A data manipulation language, which is used to display data extracted from the database and to perform simple updates and deletions.
- A data control language, which allows a database administrator to have overall control of the system, often including the administration of security, so that access to both the data and processes of the database system can be controlled.
- A graphical user interface, which may provide a visual means of browsing or querying the data, including a range of different display options such as bar charts, pie charts, etc. One particular example of such a system is Query-by-Example, in which the system displays a skeleton table (or tables), and users pose requests by suitable entry in the table.
- A forms-based user interface in which a screen-oriented form is presented to the user, who responds by filling in blanks on the form. Such forms-based systems are a popular means of providing a visual front-end to both developers and users of a database system. Typically, developers use the forms-based system in 'developer mode', where they design the forms or screens that will make up an application, and attach fragments of code

which will be triggered by the actions of users as they use the forms-based user interface.

- A DBMS procedural programming language, often based on standard third-generation programming languages such as C and COBOL, which allows programmers to develop sophisticated applications.
- Fourth-generation languages, such as Smalltalk, JavaScript, etc. These permit applications to be developed relatively quickly compared to the procedural languages mentioned above.
- A natural language user interface that allows users to present requests in free-form English statements.

Data dictionary subsystem

The data dictionary subsystem is used to store data about many aspects of how the DBMS works. The data contained in the dictionary subsystem varies from DBMS to DBMS, but in all systems it is a key component of the database. Typical data to be contained in the dictionary includes: definitions of the users of the system and the access rights they have, details of the data structures used to contain data in the DBMS, descriptions of business rules that are stored and enforced within the DBMS, and definitions of the additional data structures used to improve systems performance. It is important to understand that because of the important and sensitive nature of the data contained in the dictionary subsystem, most users will have no or little direct access to this information. However, the database administrator will need to have regular access to much of the dictionary system, and should have a detailed knowledge of the way in which the dictionary is organised.

Performance management subsystem

The performance management subsystem provides facilities to optimise (or at least improve) DBMS performance. This is necessary because the large and complex software in a DBMS requires attention to ensure it performs efficiently, i.e. it needs to allow retrieval and changes to data to be made without requiring users to wait for significant periods of time for the DBMS to carry out the requested action.

Two important functions of the performance management subsystem are:

- Query optimisation: Structuring SQL queries (or other forms of user queries) to minimise response times.
- DBMS reorganisation: Maintaining statistics on database usage, and taking (or recommending) actions such as database reorganisation, creating indexes and so on, to improve DBMS performance.

Data integrity management subsystem

The data integrity management subsystem provides facilities for managing the integrity of data in the database and the integrity of metadata in the dictionary. This subsystem is concerned with ensuring that data is, as far as software can ensure, correct and consistent. There are three important functions:

- Intra-record integrity: Enforcing constraints on data item values and types within each record in the database.
- Referential integrity: Enforcing the validity of references between records in the database.
- Concurrency control: Ensuring the validity of database updates when multiple users access the database (discussed in a later chapter).

Backup and recovery subsystem

The backup and recovery subsystem provides facilities for logging transactions and database changes, periodically making backup copies of the database, and recovering the database in the event of some type of failure. (We discuss backup and recovery in greater detail in a later chapter.) A good DBMS will provide comprehensive and flexible mechanisms for backing up and restoring copies of data, and it will be up to the database administrator, in consultation with users of the system, to decide precisely how these features should be used.

Application development subsystem

The application development subsystem is for programmers to develop complete database applications. It includes CASE tools (software to enable the modelling of applications), as well as facilities such as screen generators (for automatically creating the screens of an application when given details about the data to be input and/or output) and report generators.

In most commercial situations, there will in fact be a number of different database systems, operating within a number of different computer environments. By computer environment we mean a set of programs and data made available usually on a particular computer. One such set of database systems, used in a number of medium to large companies, involves the establishment of three different computer environments. The first of these is the development environment, where new applications are developed and new applications, whether written within the company or bought in from outside, are tested. The development environment usually contains relatively little data, just enough in fact to adequately test the logic of the applications being developed and tested. Security within the development environment is usually not an important issue, unless the actual logic of the applications being developed is, in its own right, of a sensitive nature.

The second of the three environments is often called pre-production. Applications that have been tested in the development environment will be moved into pre-production for volume testing; that is, testing with quantities of data that are typical of the application when it is in live operation.

The final environment is known as the production or live environment. Applications should only be moved into this environment when they have been fully tested in pre-production. Security is nearly always a very important issue in the production environment, as the data being used reflects important information in current use by the organisation.

Each of these separate environments will have at least one database system, and because of the widely varying activities and security measures required in each environment, the volume of data and degree of administration required will itself vary considerably between environments, with the production database(s) requiring by far the most support.

Given the need for the database administrator to migrate both programs and data between these environments, an important tool in performing this process will be a set of utilities or programs for migrating applications and their associated data both forwards and backwards between the environments in use.

Security management subsystem

The security management subsystem provides facilities to protect and control access to the database and data dictionary.

Benefits of the database approach

The benefits of the database approach are as follows:

- Ease of application development: The programmer is no longer burdened with designing, building and maintaining master files.
- Minimal data redundancy: All data files are integrated into a composite data structure. In practice, not all redundancy is eliminated, but at least the redundancy is controlled. Thus inconsistency is reduced.
- Enforcement of standards: The database administrator can define standards for names, etc.
- Data can be shared. New applications can use existing data definitions.
- Physical data independence: Data descriptions are independent of the application programs. This makes program development and maintenance an easier task. Data is stored independently of the program that uses it.
- Logical data independence: Data can be viewed in different ways by different users.

- Better modelling of real-world data: Databases are based on semantically rich data models that allow the accurate representation of real-world information.
- Uniform security and integrity controls: Security control ensures that applications can only access the data they are required to access. Integrity control ensures that the database represents what it purports to represent.
- Economy of scale: Concentration of processing, control personal and technical expertise.

Risks of the database approach

- New specialised personnel: Need to hire or train new personnel e.g. database administrators and application programmers.
- Need for explicit backup.
- Organisational conflict: Different departments have different information needs and data representation.
- Large size: Often needs alarmingly large amounts of processing power.
- Expensive: Software and hardware expenses.
- High impact of failure: Concentration of processing and resources makes an organisation vulnerable if the system fails for any length of time.

Review question 6

Distinguish between the terms 'database security' and 'data integrity'.

Data and database administration

Organisations need data to provide details of the current state of affairs; for example, the amount of product items in stock, customer orders, staff details, office and warehouse space, etc. Raw data can then be processed to enable decisions to be taken and actions to be made. Data is therefore an important resource that needs to be safeguarded. Organisations will therefore have rules, standards, policies and procedures for data handling to ensure that accuracy is maintained and that proper and appropriate use is made of the data. It is for this reason that organisations may employ data administrators and database administrators.

The role of the data administrator

It is important that the data administrator is aware of any issues that may affect the handling and use of data within the organisation. Data administration

includes the responsibility for determining and publicising policy and standards for data naming and data definition conventions, access permissions and restrictions for data and processing of data, and security issues.

The data administrator needs to be a skilled manager, able to implement policy and make strategic decisions concerning the organisation's data resource. It is not sufficient for the data administrator to propose a set of rules and regulations for the use of data within an organisation; the role also requires the investigation of ways in which the organisation can extract the maximum benefit from the available data.

One of the problems facing the data administrator is that data may exist in a range of different formats, such as plain text, formatted documents, tables, charts, photographs, spreadsheets, graphics, diagrams, multimedia (including video, animated graphics and audio), plans, etc. In cases where the data is available on computer-readable media, consideration needs to be given to whether the data is in the correct format.

The different formats in which data may appear is further complicated by the range of terms used to describe it within the organisation. One problem is the use of synonyms, where a single item of data may be known by a number of different names. An example of the use of synonyms would be the terms 'telephone number', 'telephone extension', 'direct line', 'contact number' or just 'number' to mean the organisation's internal telephone number for a particular member of staff. In an example such as this, it is easy to see that the terms refer to the same item of data, but it might not be so clear in other contexts.

A further complication is the existence of homonyms. A homonym is a term which may be used for several different items in different contexts; this can often happen when acronyms are used. One example is the use of the terms 'communication' and 'networking'; these terms are sometimes used to refer to interpersonal skills, but may also be employed in the context of data communication and computer networks.

When the items of data that are important to an organisation have been identified, it is important to ensure that there is a standard representation format. It might be acceptable to tell a colleague within the organisation that your telephone extension is 5264, but this would be insufficient information for someone outside the organisation. It may be necessary to include full details, such as international access code, national code, area code and local code as well as the telephone extension to ensure that the telephone contact details are usable worldwide.

Dates are a typical example of an item of data with a wide variety of formats. The ranges of date formats include: day-month-year, month-day-year, year-month-day, etc. The month may appear as a value in the range 1 to 12, as the name of the month in full, or a three-letter abbreviation. These formats can be varied by changing the separating character between fields from a hyphen (-) to a slash (/), full stop (.) or space ( ).

The use of standardised names and formats will assist an organisation in making good use of its data. The role of the data administrator involves the creation of these standards and their publication (including the reasons for them and guidelines for their use) across the organisation. Data administration provides a service to the organisation, and it is important that it is perceived as such, rather than the introduction of unnecessary rules and regulations.

The role of the database administrator

The role of the database administrator within an organisation focuses on a particular database or set of databases, and the associated computer applications, rather than the use of data throughout the organisation. A database administrator requires a blend of management skills together with technical expertise. In smaller organisations, the data administrator and database administrator roles may be merged into a single post, whereas larger companies may have groups of staff involved with each activity.

The activities of the database administrator take place in the context of the guidelines set out by the data administrator. This requires striking a balance between the security and protection of the database, which may be in conflict with the requirements of users to have access to the data. The database administrator has responsibility for the development, implementation, operation, maintenance and security of the database and the applications that use it. Another important function is the introduction of controls to ensure the quality and integrity of the data that is entered into the database. The database administrator is a manager of the data in the database, rather than a user. This role requires the development of the database structure and data dictionary (a catalogue of the data in the database), the provision of security measures to permit authorised access and prevent unauthorised access to data, and to guard against failures in hardware or software in order to offer reliability.

Exercise 1

Find out who is responsible for the tasks of data administration and database administration in the organisation where you are currently working or studying. Find out whether the two roles are combined into one in your organisation, or if not, how many people are allocated to each function, and what are their specific roles?

Introduction to the Relational model

A number of different approaches or models have been developed for the logical organisation of data within a database system. This 'logical' organisation must be distinguished from the 'physical' organisation of data, which describes how the data is stored on some suitable storage medium such as a disk. The physical

organisation of data will be dealt with in the chapter on physical storage. By far the most commonly used approach to the logical organisation of data is the Relational model. In this section we shall introduce the basic concepts of the Relational model, and give examples of its use. Later in the module, we shall make practical use of this knowledge in both using and developing examples of Relational database applications.

Entities, attributes and relationships

The first step in the development of a database application usually involves determining what the major elements of data to be stored are. These are referred to as entities. For example, a library database will typically contain entities such as Books, Borrowers, Librarians, Loans, Book Purchases, etc. Each of the entities identified will contain a number of properties, or attributes. For example, the entity Book will contain attributes such as Title, Author and ISBN; the entity Borrower will possess attributes such as Name, Address and Membership Number. When we have decided which entities are to be stored in a database, we also need to consider the way in which those entities are related to one another. Examples of such relationships might be, for the library system, that a Borrower can borrow a number of Books, and that a Librarian can make a number of Book Purchases. The correct identification of the entities and attributes to be stored, and the relationships between them, is an extremely important topic in database design, and will be covered in detail in the chapter on entity-relationship modelling. In introducing the Relational approach to database systems, we must consider how entities and their attributes, and the relationships between them, will be represented within a database system.

A relation is structured like a table. The rows of the structure (which are also sometimes referred to as tuples) correspond to individual instances of records stored in the relation. Each column of the relation corresponds to a particular attribute of those record instances. For example, in the relation containing details of stationery below, each row of the relation corresponds to a different item of stationery, and each column or attribute corresponds to a particular aspect of stationery, such as the colour or price.

Each tuple contains values for a fixed number of attributes. There is only one tuple for each different item represented in the database.

The set of permissible values for each attribute is called the domain for that attribute. It can be seen that the domain for the attribute Colour in the stationery relation below includes the values Red, Blue, Green, White, Yellow, and Black (other colours may be permitted but are not shown in the relation).

The sequence in which tuples appear within a relation is not important, and the order of attributes within a relation is of no significance. However, once the attributes of a particular relation have been identified, it is convenient to refer to them in the same order.

Very often it is required to be able to identify uniquely each of the different instances of entities in a database. In order to do this we use something called a primary key. We will discuss the nature of primary keys in detail in the next learning chapter, but for now we shall use examples where the primary key is the first of the attributes in each tuple of a relation.

Relation: Stationery

Item-code	Item-name	Colour	Price
19876	A4 folder	Red	0.25
19877	A4 folder	Blue	0.25
19878	A4 folder	Green	0.25
20216	A4 paper 250 sheets	Red	2.75
20217	A4 paper 250 sheets	Blue	2.75
20218	A4 paper 250 sheets	Green	2.75
20219	A4 paper 250 sheets	White	2.50
33006	A4 ring binder	Red	1.95
33007	A4 ring binder	Green	1.95
33008	A4 ring binder	Blue	1.95
33010	A4 ring binder	Yellow	1.95
33015	A4 ring binder	Black	1.50

Here, the attributes are item-code, item-name, colour and price. The values for each attribute for each item are shown as a single value in each column for a particular row. Thus for item-code 20217, the values are A4 paper 250 sheets for the item-name, Blue for the attribute colour, and <=2.75 is stored as the price.

Question: Which of the attributes in the stationery relation do you think would make a suitable key, and why?

The schema defines the 'shape' or structure of a relation. It defines the number of attributes, their names and domains. Column headings in a table represent the schema. The extension is the set of tuples that comprise the relation at any time. The extension (contents) of a relation may vary, but the schema (structure) generally does not.

From the example above, the schema is represented as:

Item-code	Item-name	Colour	Price

The extension from the above example is given as:

19876	A4 folder	Red	0.25
19877	A4 folder	Blue	0.25
19878	A4 folder	Green	0.25
20216	A4 paper 250 sheets	Red	2.75
20217	A4 paper 250 sheets	Blue	2.75
20218	A4 paper 250 sheets	Green	2.75
20219	A4 paper 250 sheets	White	2.50
33006	A4 ring binder	Red	1.95
33007	A4 ring binder	Green	1.95
33008	A4 ring binder	Blue	1.95
33010	A4 ring binder	Yellow	1.95
33015	A4 ring binder	Black	1.50

The extension will vary as rows are inserted or deleted from the table, or values of attributes (e.g. price) change. The number of attributes will not change, as this is determined by the schema. The number of rows in a relation is sometimes referred to as its cardinality. The number of attributes is sometimes referred to as the degree or grade of a relation.

Each relation needs to be declared, its attributes defined, a domain specified for each attribute, and a primary key identified.

Review question 7

Distinguish between the terms 'entity' and 'attribute'. Give some examples of entities and attributes that might be stored in a hospital database.

Review question 8

The range of values that a column in a relational table may be assigned is called the domain of that column. Many database systems provide the possibility of specifying limits or constraints upon these values, and this is a very effective way of screening out incorrect values from being stored in the system. It is useful, therefore, when identifying which attributes or columns we wish to store for an entity, to consider carefully what is the domain for each column, and which values are permissible for that domain.

Consider then for the following attributes, what the corresponding domains are,

and whether there are any restrictions we can identify which we might use to validate the correctness of data values entered into attributes with each domain:

- Attribute: EMPLOYEE_NAME
- Attribute: JOB (i.e. the job held by an individual in an organisation)
- Attribute: DATE_OF_BIRTH

Discussion topic

External schemas can be used to give individual users, or groups of users, access to a part of the data in a database. Many systems also allow the format of the data to be changed for presentation in the external schema, or for calculations to be carried out on it to make it more usable to the users of the external schema. Discuss the possible uses of external schemas, and the sorts of calculations and/or reformatting that might be used to make the data more usable to specific users or user groups.

External schemas might be used to provide a degree of security in the database, by making available to users only that part of the database that they require in order to perform their jobs. So for example, an Order Clerk may be given access to order information, while employees working in Human Resources may be given access to the details of employees.

In order to improve the usability of an external schema, the data in it may be summarised or organised into categories. For example, an external schema for a Sales Manager, rather than containing details of individual sales, might contain summarised details of sales over the last six months, perhaps organised into categories such as geographical region. Furthermore, some systems provide the ability to display data graphically, in which case it might be formatted as a bar, line or pie chart for easier viewing.

Additional content and activities

Database systems have become ubiquitous throughout computing. A great deal of information is written and published describing advances in database technology, from research papers through to tutorial information and evaluations of commercial products. Conduct a brief search on the Internet and related textbooks. You will likely find that there are many alternative definitions and explanations to the basic concepts introduced in this chapter, and these will be helpful in consolidating the material covered here.

Chapter 2. The Relational Model

Table of contents

Objectives

At the end of this chapter you should be able to:

- Describe the structure of the Relational model, and explain why it provides

a simple but well-founded approach to the storage and manipulation of data.

- Explain basic concepts of the Relational model, such as primary and foreign keys, domains, null values, and entity and referential integrity.
- Be able to discuss in terms of business applications, the value of the above concepts in helping to preserve the integrity of data across a range of applications running on a corporate database system.
- Explain the operators used in Relational Algebra.
- Use Relational Algebra to express queries on Relational databases.

Introduction

In parallel with this chapter, you should read Chapter 3 and Chapter 4 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

The aim of this chapter is to explain in detail the ideas underlying the Relational model of database systems. This model, developed through the '70s and '80s, has grown to be by far the most commonly used approach for the storing and manipulation of data. Currently all of the major suppliers of database systems, such as Oracle, IBM with DB2, Sybase, Informix, etc, base their products on the Relational model. Two of the key reasons for this are as follows.

Firstly, there is a widely understood set of concepts concerning what constitutes a Relational database system. Though some of the details of how these ideas should be implemented continue to vary between different database systems, there is sufficient consensus concerning what a Relational database should provide that a significant skill base has developed in the design and implementation of Relational systems. This means that organisations employing Relational technology are able to draw on this skill-base, as well as on the considerable literature and consultancy know-how available in Relational systems development.

The second reason for the widespread adoption of the Relational model is robustness. The core technology of most major Relational products has been tried and tested over the last 12 or so years. It is a major commitment for an organisation to entrust the integrity, availability and security of its data to software. The fact that Relational systems have proved themselves to be reliable and secure over a significant period of time reduces the risk an organisation faces in committing what is often its most valuable asset, its data, to a specific software environment.

Relational Algebra is a procedural language which is a part of the Relational model. It was originally developed by Dr E. F. Codd as a means of accessing data in Relational databases. It is independent of any specific Relational database product or vendor, and is therefore useful as an unbiased measure of the power

of Relational languages. We shall see in a later chapter the further value of Relational Algebra, in helping gain an understanding of how transactions are processed internally within the database system.

Context

The Relational model underpins most of the major database systems in commercial use today. As such, an understanding of the ideas described in this chapter is fundamental to these systems. Most of the remaining chapters of the module place a strong emphasis on the Relational approach, and even in those that examine research issues that use a different approach, such as the chapter on Object databases, an understanding of the Relational approach is required in order to draw comparisons and comprehend what is different about the new approach described. The material on Relational Algebra provides a vendor-independent and standard approach to the manipulation of Relational data. This information will have particular value when we move on to learn the Structured Query Language (SQL), and also assist the understanding of how database systems can alter the ways queries were originally specified to reduce their execution time, a topic covered partially in the chapter called Database Administration and Tuning.

Structure of the Relational model

Theoretical foundations

Much of the theory underpinning the Relational model of data is derived from mathematical set theory. The seminal work on the theory of Relational database systems was developed by Dr E. F. Codd, (Codd 1971). The theoretical development of the model has continued to this day, but many of the core principles were described in the papers of Codd and Date in the '70s and '80s.

The application of set theory to database systems provides a robust foundation to both the structural and data-manipulation aspects of the Relational model. The relations, or tables, of Relational databases, are based on the concepts of mathematical sets. Sets in mathematics contain members, and these correspond to the rows in a relational table. The members of a set are unique, i.e. duplicates are not allowed. Also the members of a set are not considered to have any order; therefore, in Relational theory, the rows of a relation or table cannot be assumed to be stored in any specific order (note that some database systems allow this restriction to be overridden at the physical level, as in some situations, for example to improve the performance response of the database, it can be desirable to ensure the ordering of records in physical storage).

Uniform representation of data

We saw in the previous chapter, that the Relational model uses one simple data structure, the table, to represent information. The rows in tables correspond to specific instances of records, for example, a row of a customer table contains information about a particular customer. Columns in a table contain information about a particular aspect of a record, for example, a column in a customer record might contain a customer's contact telephone number.

Much of the power and robustness of the Relational approach derives from the use of simple tabular structures to represent data. To illustrate this, consider the information that might typically be contained in part of the data dictionary of a database. In a data dictionary, we will store the details of logical table structures, the physical allocations of disk space to tables, security information, etc. This data dictionary information will be stored in tables, in just the same way that, for example, customer and other information relevant to the end users of the system will be stored.

This consistency of data representation means that the same approach for data querying and manipulation can be applied throughout the system.

Relation

A relation is a table with columns and tuples. A database can contain as many tables as the designer wants. Each table is an implementation of a real-world entity. For example, the university keeps information about students. A student is represented as an entity during database design stage. When the design is implemented, a student is represented as a table. You will learn about database design is later modules.

Attribute

An attribute is a named column in the table. A table can contain as many attributes as the designer wants. Entities identified during database design may contain attributes/characteristics that describe the entity. For example, a student has a student identification number and a name. Student identification and name will be implemented as columns in the student table.

Domain

The domain is the allowable values for a column in the table. For example, a name of a student can be made of a maximum of 30 lower and upper case characters. Any combination of lower and upper case characters less or equal to 30 is the domain for the name column.

Tuple

A tuple is the row of the table. Each tuple represents an instance of an entity. For example, a student table can contain a row holding information about Moses. Moses is an instance of the student entity.

Degree

The degree of a relation/table is the number of columns it contains.

Cardinality

The cardinality of a relation/table is the number of rows it contains.

Primary key

In the previous chapter, we described the use of primary keys to identify each of the rows of a table. The essential point to bear in mind when choosing a primary key is that it must be guaranteed to be unique for each different row of the table, and so the question you should always ask yourself is whether there is any possibility that there could be duplicate values of the primary key under consideration. If there is no natural candidate from the data items in the table that may be used as the primary key, there is usually the option of using a system-generated primary key. This will usually take the form of an ascending sequence of numbers, a new number being allocated to each new instance of a record as it is created. System-generated primary keys, such as this, are known as surrogate keys. A drawback to the use of surrogate keys is that the unique number generated by the system has no other meaning within the application, other than serving as a unique identifier for a row in a table. Whenever the option exists therefore, it is better to choose a primary key from the available data items in the rows of a table, rather than opting for an automatically generated surrogate key.

Primary keys may consist of a single column, or a combination of columns. An example of a single table column would be the use of a unique employee number in a table containing information about employees.

As an example of using two columns to form a primary key, imagine a table in which we wish to store details of project tasks. Typical data items we might store in the columns of such a task table might be: the name of the task, date the task was started, expected completion date, actual completion date, and the employee number of the person responsible for ensuring the completion of the task. There is a convenient, shorthand representation for the description of a table as given above: we write the name of the table, followed by the name of

the columns of the table in brackets, each column being separated by a comma. For example:

TASK (TASK_NAME, START_DATE, EXPECTED_COMP_DATE, COMP_DATE, EMPNO)

We shall use this convention for expressing the details of the columns of a table in examples in this and later chapters. Choosing a primary key for the task table is straightforward while we can assume that task names are unique. If that is the case, then we may simply use task name as the primary key. However, if we decide to store in the task table, the details of tasks for a number of different projects, it is less likely that we can still be sure that task names will be unique. For example, supposing we are storing the details of two projects, the first to buy a new database system, and the second to move a business to new premises. For each of these projects, we might have a task called 'Evaluate alternatives'. If we wish to store the details of both of these tasks in the same task table, we can now no longer use TASK_NAME as a unique primary key, as it is duplicated across these two tasks.

As a solution to this problem, we can combine the TASK_NAME column with something further to add the additional context required to provide a unique identifier for each task. In this case, the most sensible choice is the project name. So we will use the combination of the PROJECT_NAME and TASK_NAME data items in our task table in order to identify uniquely each of the tasks in the table. The task table becomes:

TASK (PROJECT_NAME, TASK_NAME, START_DATE, EXPECTED_COMP_DATE, COMP_DATE, EMPNO)

We may, on occasions, choose to employ more than two columns as a primary key in a table, though where possible this should be avoided as it is both unwieldy to describe, and leads to relatively complicated expressions when it comes to querying or updating data in the database. Notice also that we might have used a system-generated surrogate key as the solution to the problem of providing a primary key for tasks, but the combination of PROJECT_NAME and TASK_NAME is a much more meaningful key to users of the application, and is therefore to be preferred.

Foreign keys

Very often we wish to relate information stored in different tables. For example, we may wish to link together the tasks stored in the task table described above, with the details of the projects to which those tasks are related. The simplicity by which this is achieved within the Relational model, is one of the model's major strengths. Suppose the Task and Project tables contain the following attributes:

TASKS (TASK_NAME, START_DATE, EXP_COMP_DATE, COMP_DATE, EMPNO)

PROJECT (PROJECT_NAME, START_DATE, EXP_COMP_DATE, COMP_DATE, PROJECT_LEADER)

We assume that TASK_NAME is an appropriate primary key for the TASK table, and PROJECT_NAME is an appropriate primary key for the PROJECT table.

In order to relate a record of a task in a task table to a record of a corresponding project in a project table, we use a concept called a foreign key. A foreign key is simply a piece of data that allows us to link two tables together. In the case of the projects and tasks example, we will assume that each project is associated with a number of tasks. To form the link between the two tables, we place the primary key of the PROJECT table into the TASK table. The task table then becomes:

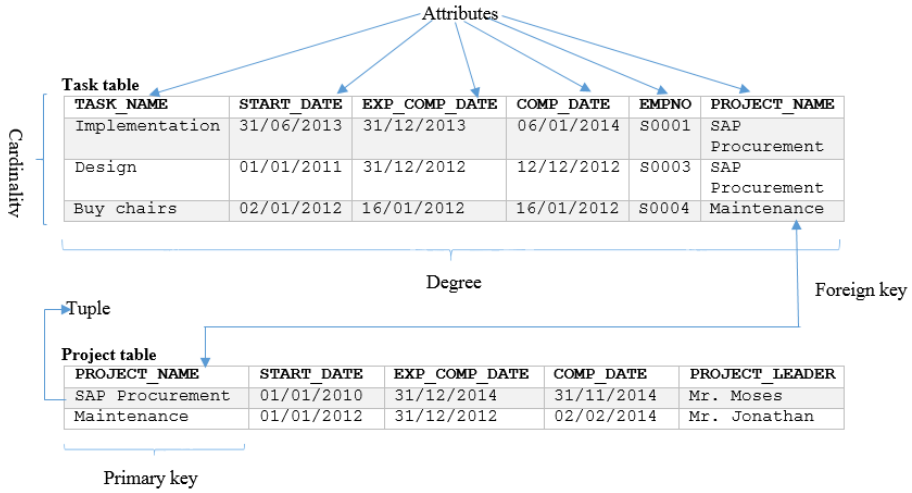
TASK (TASK_NAME, START_DATE, EXP_COMP_DATE, COMP_DATE, EMPNO, PROJECT_NAME)

Through the use of PROJECT_NAME as a foreign key, we are now able to see, for any given task, the project to which it belongs. Specifically, the tasks associated with a particular project can now be identified simply by virtue of the fact that they contain that project's name as a data item as one of their attributes. Thus, all tasks associated with a research project called GENOME RESEARCH, will contain the value of GENOME RESEARCH in their PROJECT_NAME attribute.

The beauty of this approach is that we are forming the link using a data item. We are still able to maintain the tabular structure of the data in the database, but can relate that data in whatever ways we choose. Prior to the development of Relational systems, and still in many non-Relational systems today, rather than using data items in this way to form the link between different entity types within a database, special link items are used, which have to be created, altered and removed. These activities are in addition to the natural insertions, updates and deletions of the data itself. By using a uniform representation of data both for the data values themselves, and the links between different entity types, we achieve uniformity of expression of queries and updates on the data.

The need to link entity types in this way is a requirement of all, other than the most trivial of, database applications. That it occurs so commonly, allied with the simplicity of the mechanism for achieving it in Relational systems, has been a major factor in the widespread adoption of Relational databases.

Below is the summary of the concepts we have covered so far:



Integrity constraints

Integrity constraints are restrictions that affect all the instances of the database.

Nulls

There is a standard means of representing information that is not currently known or unavailable within Relational database systems. We say that a column for which the value is not currently known, or for which a value is not applicable, is null. Null values have attracted a large amount of research within the database community, and indeed for the developers and users of database systems they can be an important consideration in the design and use of database applications (as we shall see in the chapters on the SQL language).

An important point to grasp about null values is that they are a very specific way of representing the fact that the data item in question literally is not currently set to any value at all. Prior to the use of null values, and still in some systems today, if it is desired to represent the fact that a data item is not currently set to some value, an alternative value such as 0, or a blank space, will be given to that data item. This is poor practice, as of course 0, or a blank space, are perfectly legitimate values in their own right. Use of null values overcomes this problem, in that null is a value whose meaning is simply that there is no value currently allocated to the data item in question.

There are a number of situations in which the use of null values is appropriate. In general we use it to indicate that a data item currently has no value allocated to it. Examples of when this might happen are:

- When the value of the data item is not yet known.

- When the value for that data item is yet to be entered into the system.
- When it is not appropriate that this particular instance of the data item is given a value.

An example of this last situation might be where we are recording the details of employees in a table, including their salary and commission. We would store the salaries of employees in one table column, and the details of commission in another. Supposing that only certain employees, for example sales staff, are paid commission. This would mean that all employees who are not sales staff would have the value of their commission column set to null, indicating that they are not paid commission. The use of null in this situation enables us to represent the fact that some commissions are not set to any specific value, because it is not appropriate to pay commission to these staff.

Another result of this characteristic of null values is that where two data items both contain null, if you compare them with one another in a query language, the system will not find them equal. Again, the logic behind this is that the fact that each data item is null does not mean they are equal, it simply means that they contain no value at all.

Entity integrity

As briefly discussed in Chapter 1, in Relational databases, we usually use each table to store the details of particular entity types in a system. Therefore, we may have a table for Customers, Orders, etc. We have also seen the importance of primary keys in enabling us to distinguish between different instances of entities that are stored in the different rows of a table.

Consider for the moment the possibility of having null values in primary keys. What would be the consequences for the system?

Null values denote the fact that the data item is not currently set to any real value. Imagine, however, that two rows in a table are the same, apart from the fact that part of their primary keys are set to null. An attempt to test whether these two entity instances are the same will find them not equal, but is this really the case? What is really going on here is that the two entity instances are the same, other than the fact that a part of their primary keys are as yet unknown. Therefore, the occurrence of nulls in primary keys would stop us being able to compare entity instances. For this reason, the column or columns used to form a primary key are not allowed to contain null values. This rule is known as the Entity Integrity Rule, and is a part of the Relational theory that underpins the Relational model of data. The rule does not ensure that primary keys will be unique, but by not allowing null values to be included in primary keys, it does avoid a major source of confusion and failure of primary keys.

Referential integrity

If a foreign key is present in a given table, it must either match some candidate value in the home table or be set to null. For example, in our project and task example above, every value of PROJECT_NAME in the task table must exist as a value in the PROJECT_NAME column of the project table, or else it must be set to null.

General constraints

These are additional rules specified by the users or database administrators of a database, which define or constrain some aspect of the enterprise. For example, a database administrator can contain the PROJECT_NAME column to have a maximum of 30 characters for each value inserted.

Data manipulation: The Relational Algebra

Restrict

Restrict (also known as 'select') is used on a single relation, producing a new relation by excluding (restricting) tuples in the original relation from the new relation if they do not satisfy a condition; thus only the tuples required are selected.

This operation has the effect of choosing certain tuples (rows) from the table, as illustrated in the diagram below.

The use of the term 'select' here is quite specific as an operation in the Relational Algebra. Note that in the database query language SQL, all queries are phrased using the term 'select'. The Relational Algebra 'select' means 'extract tuples which meet specific criteria'. The SQL 'select' is a command that means 'produce a table from existing relations using Relational Algebra operations'.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Using the Relational Algebra, extract from the relation Singers those individuals who are over 40 years of age, and create a new relation called Mature-Singers.

Relational Algebra operation: Restrict from Singers where age > 40 giving Mature-Singers

We can see which tuples are chosen from the relation Singers, and these are identified below:

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

The new relation, Mature-Singers, contains only those tuples for singers aged over 40 extracted from the relation Singers. These were shown highlighted in the relation above.

New Relation: Mature-Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0247	Desmond Venables	27 Long Lane	55
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Note that Anne Freeman (age 40) is not shown. The query explicitly stated those over 40, and therefore anyone aged exactly 40 is excluded.

If we wanted to include singers aged 40 and above, we could use either of the following operations which would have the same effect.

Either:

Relational Algebra operation: Restrict from Singers where age >= 40 giving Mature-Singers2

or

Relational Algebra operation: Restrict from Singers where age > 39 giving Mature-Singers2

The result of either of these operations would be as shown below:

New Relation: Mature-Singers2

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Project

Project is used on a single relation, and produces a new relation that includes only those attributes requested from the original relation. This operation has the effect of choosing columns from the table, with duplicate entries removed.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

The task here is to extract the names of the singers, without their ages or any other information as shown in the diagram below. Note that if there were two singers with the same name, they would be distinguished from one another in the relation Singers by having different singer-id numbers. If only the names are extracted, the information that there are singers with the same name will not be preserved in the new relation, as only one copy of the name would appear.

Relation singers:

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

The column for the attribute singer-names is shown highlighted in the relation above.

The following Relational Algebra operation creates a new relation with singer-name as the only attribute.

Relational Algebra operation: Project Singer-name over Singers giving Singer-names

New Relation: Singer-names

Singer-name
Helen Drummond
Katerina Christou
Desmond Venables
Anne Freeman
Alphonse Trieste
Tamanna Patel
Swee Hor Tan
Panos Constantinou

Union

Union (also called 'append') forms a new relation of all tuples from two or more existing relations, with any duplicate tuples deleted. The participating relations must be union compatible.

Relation : Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

It can be seen that the relations Singers and Actors have the same number of attributes, and these attributes are of the same data types (the identification numbers are numeric, the names are character fields, the address field is alphanumeric, and the ages are integer values). This means that the two relations are union compatible. Note that it is not important whether the relations have the same number of tuples.

The two relations Singers and Actors will be combined in order to produce a new relation Performers, which will include details of all singers and all actors. An individual who is a singer as well as an actor need only appear once in the new relation. The Relational Algebra operation union (or append) will be used

in order to generate this new relation.

Union Compatible Relations: Singers and Actors

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	4

Actor-id	**Actor-name**	**Address**	**Age**
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

The union operation will unite these two union-compatible relations to form a new relation containing details of all singers and actors.

Relational Algebra operation: Union Singers and Actors giving Performers

We could also express this activity in the following way:

Relational Algebra operation: Union Actors and Singers giving Performers

These two operations would generate the same result; the order in which the participating relations are given is unimportant. When an operation has this property it is known as commutative - other examples of this include addition and multiplication in arithmetic. Note that this does not apply to all Relational Algebra operations.

The new relation Performers contains one tuple for every tuple that was in the relation Singers or the relation Actors; if a tuple appeared in both Singers and

Actors, it will appear only once in the new relation Performers. This is why there is only one tuple in the relation Performers for each of the individuals who are both actors and singers (Helen Drummond and Desmond Venables), as they appear in both the original relations.

New Relation: Performers

Performer-id	Performer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49
0107	Basil Bond	14 Mutton Lane	45
0234	Arthur Simpson	35 Fox Corner	52
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

Intersection

Intersection creates a new relation containing tuples that are common to both the existing relations. The participating relations must be union compatible.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

As before, it can be seen that the relations Singers and Actors are union compatible as they have the same number of attributes, and corresponding attributes are of the same data type.

We can see that there are some tuples that are common to both relations, as illustrated in the diagram below. It is these tuples that will form the new relation as a result of the intersection operation.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

The Relational Algebra operation intersection will extract the tuples common to both relations, and use these tuples to create a new relation.

Relational Algebra operation: Actors Intersection Singers giving Actor-Singers

New Relation: Actor-Singers

Actor Singer-id	Name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0247	Desmond Venables	27 Long Lane	55

The intersection of Actors and Singers produces a new relation containing only those tuples that occur in both of the original relations. If there are no tuples that are common to both relations, the result of the intersection will be an empty relation (i.e. there will be no tuples in the new relation).

Note that an empty relation is not the same as an error; it simply means that there are no tuples in the relation. An error would occur if the two relations were found not to be union compatible.

Difference

Difference (sometimes referred to as 'remove') forms a new relation by excluding tuples from one relation that occur in another. The resulting relation contains tuples that were present in the first relation only, but not those that occur in both the first and the second relations, or those that occur in the second relation alone. This operation can be regarded as 'subtracting' tuples in one relation from another relation. The participating relations must be union compatible.

Relation: Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

If we want to find out which actors are not also singers, the following Relational Algebra operation will achieve this:

Relational Algebra operation: Actors difference Singers giving Only-Actors

The result of this operation is to remove from the relation Actors those tuples that also occur in the relation Singers. In effect, we are removing the tuples in the intersection of Actors and Singers in order to create a new relation that contains only actors. The diagram below shows which tuples are in both relations.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

The new relation Only-Actors contains tuples from the relation Actors only if they were not also present in the relation Singers.

Relation: Only-Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0234	Arthur Simpson	35 Fox Corner	52
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

It can be seen that this operation produces a new relation Only-Actors containing all tuples in the relation Actors except those that also occur in the relation Singers.

If it is necessary to find out which singers are not also actors, this can be done by the relational operation 'remove Actors from Singers', or 'Singers difference Actors'. This operation would not produce the same result as 'Actors difference Singers' because the relational operation difference is not commutative (the participating relations cannot be expressed in reverse order and achieve the same result).

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Actors

Actor-id	Actor-name	Address	Age
0107	Basil Bond	14 Mutton Lane	45
0126	Helen Drummond	1 Thorley Street	42
0234	Arthur Simpson	35 Fox Corner	52
0247	Desmond Venables	27 Long Lane	55
0384	Pru Prentice	17 Sea Drive	22
0452	Paresh Patel	3 The Terrace	27
0748	Vanessa Williams	2 Park Avenue	42
0992	May Royle	39 Sea Walk	36

Relational Algebra operation: Singers difference Actors giving Only-Singers

The effect of this operation is to remove from the relation Singers those tuples that are also present in the relation Actors. The intersection of the two relations is removed from the relation Singers to create a new relation containing tuples of those who are only singers. The intersection of the relations Singers and Actors is, of course, the same as before.

Relation: Only Singers

Singer-id	Singer-name	Address	Age
0243	Katerina Christou	12 High Road	37
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	4

This operation produces a new relation Only-Singers containing all tuples in the relation Singers except those that also occur in the relation Actors. The tuples that are removed from one relation when the difference between two relations is generated are those that are in the intersection of the two relations.

Cartesian product

If a relation called Relation-A has a certain number of tuples (call this number N), this can be represented as NA (meaning the number of tuples in Relation-A). Similarly, Relation-B may have a different number of tuples (call this number M), which can be shown as MB (meaning the number of tuples in Relation-B).

The resulting relation from the operation 'Relation-A Cartesian product Relation-B' forms a new relation containing NA * MB tuples (meaning the number of tuples in Relation-A times the number of tuples in Relation-B). Each tuple in the new relation created as a result of this operation will consist of each tuple from Relation-A paired with each tuple from Relation-B, which includes all possible combinations.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Roles

Role-id	Role-name
0101	Figaro
0175	Mimi

In order to create a new relation which pairs each singer with each role, we need to use the relational operation Cartesian product.

Relational Algebra operation: Singers Cartesian product Roles giving Singers-Roles

New relation: Singers-Roles

Singer-id	Name	Address	Age	Role-id	Role-name
0126	Helen Drummond	1 Thorley Street	42	0101	Figaro
0126	Helen Drummond	1 Thorley Street	42	0175	Mimi
0243	Katerina Christou	12 High Road	37	0101	Figaro
0243	Katerina Christou	12 High Road	37	0175	Mimi
0247	Desmond Venables	27 Long Lane	55	0101	Figaro
0247	Desmond Venables	27 Long Lane	55	0175	Mimi
0259	Anne Freeman	5 Tower Hill	40	0101	Figaro
0259	Anne Freeman	5 Tower Hill	40	0175	Mimi
0594	Alphonse Trieste	20 Longchamps	34	0101	Figaro
0594	Alphonse Trieste	20 Longchamps	34	0175	Mimi
0628	Tamanna Patel	9 Crown Hill	23	0101	Figaro
0628	Tamanna Patel	9 Crown Hill	23	0175	Mimi
0855	Swee Hor Tan	4 Long Lane	54	0101	Figaro
0855	Swee Hor Tan	4 Long Lane	54	0175	Mimi
0876	Panos Constantinou	32 Mallet Road	49	0101	Figaro
0876	Panos Constantinou	32 Mallet Road	49	0175	Mimi

The result of Singers Cartesian product Roles gives a new relation Singers-Roles, showing each tuple from one relation with each tuple of the other, producing the tuples in the new relation. Each singer is associated with all roles; this produces a relation with 16 tuples, as there were 8 tuples in the relation Singers, and 2 tuples in the relation Roles.

What do you think would be the result of the following?

Relational Algebra operation: Singers Cartesian product Singers giving Singers-Extra

Relational Algebra operation: Actors Cartesian product Roles giving Actors-Roles

Division

The Relational Algebra operation 'Relation-A divide by Relation-B giving Relation-C' requires that the attributes of Relation-B must be a subset of those of Relation-A. The relations do not need to be union compatible, but they must have some attributes in common. The attributes of the result, Relation-C, will also be a subset of those of Relation-A. Division is the inverse of Cartesian product; it is sometimes easier to think of the operation as similar to division in simple algebra and arithmetic.

In arithmetic:

If, for example:

Value-A = 2

Value-B = 3

Value-C = 6

then: Value-C is the result of Value-A times Value-B (i.e. 6 = 2 * 3) and: Value-C divided by Value-A gives Value-B as a result(i.e. 6/2 = 3) and: Value-C divided by Value-B gives Value-A as a result (i.e. 6/3 = 2)

In Relational Algebra:

If:

Relation-A = Singers

Relation-B = Roles

Relation-C = Singers-Roles

then: Relation-C is the result of Relation-A Cartesian product Relation-B and: Relation-C divided by Relation-A gives Relation-B as a result

and: Relation-C divided by Relation-B gives Relation-A as a result.

If we start with two relations, Singers and Roles, we can create a new relation Singers-Roles by performing the Cartesian product of Singers and Roles. This new relation shows every role in turn with every singer.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relation: Roles

Role-id	Role-name
0101	Figaro
0175	Mimi

The new relation Singers-Roles has a special relationship with the relations Singers and Roles from which it was created, as will be demonstrated below.

New Relation: Singers-Roles

Singer-id	Name	Address	Age	Role-id	Role-name
0126	Helen Drummond	1 Thorley Street	42	0101	Figaro
0126	Helen Drummond	1 Thorley Street	42	0175	Mimi
0243	Katerina Christou	12 High Road	37	0101	Figaro
0243	Katerina Christou	12 High Road	37	0175	Mimi
0247	Desmond Venables	27 Long Lane	55	0101	Figaro
0247	Desmond Venables	27 Long Lane	55	0175	Mimi
0259	Anne Freeman	5 Tower Hill	40	0101	Figaro
0259	Anne Freeman	5 Tower Hill	40	0175	Mimi
0594	Alphonse Trieste	20 Longchamps	34	0101	Figaro
0594	Alphonse Trieste	20 Longchamps	34	0175	Mimi
0628	Tamanna Patel	9 Crown Hill	23	0101	Figaro
0628	Tamanna Patel	9 Crown Hill	23	0175	Mimi
0855	Swee Hor Tan	4 Long Lane	54	0101	Figaro
0855	Swee Hor Tan	4 Long Lane	54	0175	Mimi
0876	Panos Constantinou	32 Mallet Road	49	0101	Figaro
0876	Panos Constantinou	32 Mallet Road	49	0175	Mimi

We can see that the attributes of the relation Singers are a subset of the attributes of the relation Singers-Roles. Similarly, the attributes of the relation Roles are also a subset (although a different subset) of the relation Singers-Roles.

If we now divide the relation Singers-Roles by the relation Roles, the resulting relation will be the same as the relation Singers.

Relational Algebra operation: Singers-Roles divide by Roles giving Our-Singers

New Relation: Our-Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Similarly, if we divide the relation Singers-Roles by the relation Singers, the relation that results from this will be the same as the original Roles relation.

Relational Algebra operation: Singers-Roles divide by Singers giving Our-Roles

Relation: Our-Roles

Role-id	Role-name
0101	Figaro
0175	Mimi

Note that there are only two tuples in this relation, although the attributes role-id and role-name appeared eight times in the relation Singer-Roles, as each role was associated with each singer in turn.

In the case where not all tuples in one relation have corresponding tuples in the 'dividing' relation, the resulting relation will only contain those tuples which are represented in both the 'dividing' and 'divided' relations. In such a case it would not be possible to recreate the 'divided' relation from a Cartesian product of the 'dividing' and resulting relations. The next example demonstrates this.

Consider the relation Recordings shown below, which holds details of the songs recorded by each of the singers.

Relation: Recordings

Singer-id	Singer-name	Song-id	Song
0126	Helen Drummond	9204	Desperado
0126	Helen Drummond	9365	Yesterday
0243	Katerina Christou	9123	Crazy
0243	Katerina Christou	9204	Desperado
0243	Katerina Christou	9365	Yesterday
0247	Desmond Venables	9123	Crazy
0247	Desmond Venables	9365	Yesterday
0259	Anne Freeman	9123	Crazy
0259	Anne Freeman	9204	Desperado
0594	Alphonse Trieste	9204	Desperado
0628	Tamanna Patel	9123	Crazy
0628	Tamanna Patel	9204	Desperado
0628	Tamanna Patel	9365	Yesterday
0855	Swee Hor Tan	9365	Yesterday
0876	Panos Constantinou	9123	Crazy
0876	Panos Constantinou	9204	Desperado

Three individuals, Chris, Mel and Sam, have each created two new relations listing their favourite songs and their favourite singers. The use of the division operation will enable Chris, Mel and Sam to find out which singers have recorded their favourite songs, and also which songs their favourite singers have recorded.

Relation: Chris-Favourite-Songs

Song-id	Song
9123	Crazy
9365	Yesterday

The table above contains Chris's favourite songs. In order to find out which singers have made a recording of these songs, we need to divide this relation into the Recordings relation. The result of this Relational Algebra operation is a new relation containing the details of singers who have recorded all of Chris's favourite songs. Singers who have recorded some, but not all, of Chris's favourite songs are not included.

Relational Algebra operation: Recordings divide by Chris-Favourite-Songs giv-

ing Chris-Singers-Songs

Relation: Chris-Singers-Songs

Singer-id	Singer-name
0243	Katerina Christou
0247	Desmond Venables
0628	Tamanna Patel

Note that the singers in this relation are not the same as those in the relation Chris-Favourite-Singers. The reason for this is that Chris's favourite singers have not all recorded Chris's favourite songs.

The next relation shows Chris's favourite singers. Chris wants to know which songs these singers have recorded. If we divide the Recordings relation by this relation, we will get a new relation that contains the songs that all these singers have recorded; songs that have been recorded by some, but not all, of the singers will not be included in the new relation.

Relation: Chris-Favourite-Singers

Singer-id	Singer-name
0243	Katerina Christou
0259	Anne Freeman
0876	Panos Constantinou

The singers in this relation are not the same as those who sing Chris's favourite songs. The reason for this is that not all of Chris's favourite singers have recorded these songs.

In order to discover the songs recorded by Chris's favourite singer, we need to divide the relation Recordings by Chris-Favourite-Songs.

Relational Algebra operation: Recordings divide by Chris-Favourite-Songs giving Chris-Songs-by-Singers

The new relation created by this operation will provide us with the information about which of Chris's favourite singers has recorded all of Chris's chosen songs. Any singer who has recorded some, but not all, of Chris's favourite songs will be excluded from this new relation.

Relation: Chris-Songs-by-Singers

Song-id	Song
9123	Crazy
9204	Desperado

We can see that the relation created to identify the songs recorded by Chris's favourite singers is not the same as Chris's list of favourite songs, because these singers have not all recorded the songs listed as Chris's favourites.

In this example, we have been able to generate two new relations by dividing into the Recordings relation. These two new relations do not correspond with the other two relations that were divided into the relation Recordings, because there is no direct match. This means that we could not recreate the Recordings relation by performing a Cartesian product operation on the two relations containing Chris's favourite songs and singers.

In the next example, we will identify the songs recorded by Mel's favourite singers, and which singers have recorded Mel's favourite songs. In common with Chris's choice, we will find that the singers and the songs do not match as not all singers have recorded all songs. If all singers had recorded all songs, the relation Recordings would be the result of Singers Cartesian product Songs, but this is not the case.

Mel's favourite songs include all those that have been recorded, but not all singers have recorded all songs. Mel wants to find out who has recorded these songs, but the result will only include those singers who have recorded all the songs.

Relation: Mel-Favourite-Songs

Song-id	Song
9123	Crazy
9204	Desperado
9365	Yesterday

There are no other songs that have been recorded from the list available; Mel has indicated that all of these are favourites.

The relational operation below will produce a new relation, Mel-Singers-Songs,

which will contain details of those singers who have recorded all of Mel's favourite songs.

Relational Algebra operation: Recordings divide by Mel-Favourite-Songs giving Mel-Singers-Songs

Relation: Mel-Singers-Songs

Singer-id	Singer-name
0243	Katerina Christou
0628	Tamanna Patel

We can see from this relation, and the one below, that none of Mel's favourite singers has recorded all of the songs selected as Mel's favourites.

Relation: Mel-Favourite-Singers

Singer-id	Singer-name
0259	Anne Freeman
0594	Alphonse Trieste

We can use the relation Mel-Favourite-Singers to find out which songs have been recorded by both these performers.

Relational Algebra operation: Recordings divide by Mel-Favourite-Singers giving Mel-Songs-by-Singers

There is only one song that has been recorded by the singers Mel has chosen, as shown in the relation below.

Relation: Mel-Songs-by-Singers

Song-id	Song
9204	Desperado

We can now turn our attention to Sam's selection of songs and singers. It happens that Sam's favourite song is the same one that Mel's favourite singers have recorded.

Relation: Sam-Favourite-Songs

Song-id	Song
9204	Desperado

If we now perform the reverse query to find out who has recorded this song, we find that there are more singers who have recorded this song than the two who were Mel's favourites. The reason for this difference is that Mel's query was to find out which song had been recorded by particular singers. This contrasts with Sam's search for any singer who has recorded this song.

Relational Algebra operation: Recordings divide by Sam-Favourite-Songs giving Sam-Singer-Songs

Relation: Sam-Singers-Songs

Singer-id	Singer-name
0126	Helen Drummond
0243	Katerina Christou
0259	Anne Freeman
0594	Alphonse Trieste
0628	Tamanna Patel
0876	Panos Constantinou

Here we can see that Mel's favourite singers include the performers who have recorded Sam's favourite song, but there are many other singers who have also made a recording of the same song. Indeed, there are only two singers who have not recorded this song (Desmond Venables and Swee Hor Tan). This could be considered unfortunate for Sam, as these are the only two singers named as Sam's favourites.

Relation: Sam-Favourite-Songs

Singer-id	Singer-name
0247	Desmond Venables
0855	Swee Hor Tan

We know that the only two singers who have not recorded Sam's favourite song are in fact Sam's favourite singers. It is now our task to discover which songs these two singers have recorded.

Relation Algebra operation: Recordings divide by Sam-Favourite-Singers giving Sam-Songs-by-Singers

This operation creates a new relation that reveals the identity of the song that has been recorded by all of Sam's favourite singers.

Relation: Sam-Songs-by-singers

Song-id	Song
9365	Yesterday

There is only one song that these two singers have recorded. Indeed, Swee Hor Tan has only recorded this song, and therefore whatever other songs had been recorded by Desmond Venables, this song is the only one that fulfils the criteria of being recorded by both these performers.

Join

Join forms a new relation with all tuples from two relations that meet a condition. The relations might happen to be union compatible, but they do not have to be.

The following two relations have a conceptual link, as the stationery orders have been made by some of the singers. Invoices can now be generated for each singer who placed an order. (Note that we would not wish to use Cartesian product here, as not all singers have placed an order, and not all orders are the same.)

The relation Orders identifies the stationery items (from the Stationery relation) that have been requested, and shows which customer ordered each item (here

the customer-id matches the singer-id).

Relations: Orders

Order-id	Item-code	Qty	Unit price	customer-id
17406	19876	3	0.25	0243
17592	20217	1	2.75	0247
18083	33015	2	1.50	0628
18087	20219	2	2.50	0855
18265	33007	5	0.25	0628

The Singers relation contains the names and addresses of all singers (who are the customers), allowing invoices to be prepared by matching the customers who have placed orders with individuals in the Singers relation.

Relation: Singers

Singer-id	Singer-name	Address	Age
0126	Helen Drummond	1 Thorley Street	42
0243	Katerina Christou	12 High Road	37
0247	Desmond Venables	27 Long Lane	55
0259	Anne Freeman	5 Tower Hill	40
0594	Alphonse Trieste	20 Longchamps	34
0628	Tamanna Patel	9 Crown Hill	23
0855	Swee Hor Tan	4 Long Lane	54
0876	Panos Constantinou	32 Mallet Road	49

Relational Algebra operation: Join Singers to Orders where Singers Singer-id = Orders Customer-id giving Invoices

The relation Singers is joined to the relation Orders where the attribute Singer-id in Singers has the same value as the attribute Customer-id in Orders, to form a new relation Invoices.

New Relation: Invoices

Order -id	item-code	qty	Unit price	Cust -id	Singer -id	Singer -name	Address	Age
17406	19876	3	0.25	0243	0243	Katerina Christou	12 High Road	37
17592	20217	1	2.75	0247	0247	Desmond Venables	27 Long Lane	55
18083	33015	2	1.50	0628	0628	Tamanna Patel	9 Crown Hill	23
18087	20219	2	2.50	0855	0855	Swee Hor Tan	4 Long Lane	54
18265	33007	5	0.25	0628	0628	Tamanna Patel	9 Crown Hill	23

The attribute which links together the two relations (the identification number) occurs in both original relations, and thus is found twice in the resulting new relation; the additional occurrence can be removed by means of a project operation. A version of the join operation, in which such a project is assumed to occur automatically, is known as a natural join.

The types of join operation that we have used so far, and that are in fact by far most commonly in use, are called equi-joins. This is because the two attributes to be compared in the process of evaluating the join operation are compared for equality with one another. It is possible, however, to have variations on the join operation using operators other than equality. Therefore it is possible to have a GREATER THAN (>) JOIN, or a LESS THAN (<) JOIN.

It would have been possible to create the relation Invoices by producing the Cartesian product of Singers and Orders, and then selecting only those tuples where the Singer-id attribute from Singers and the Customer-id attribute from Orders has the same value. The join operation enables two relations which are not union compatible to be linked together to form a new relation without generating a Cartesian product, and then extracting only those tuples which are required.

Activities

Activity 1: Relational Algebra I

Let X be the set of student tuples for students studying databases, and Y the set of students who started university in 1995. Using this information, what would be the result of:

1. X union Y
2. X intersect Y
3. X difference Y

Activity 2: Relational Algebra II

Describe, using examples, the characteristics of an equi-join and a natural join.

Activity 3: Relational Algebra III

Consider the following relation A with attributes X and Y,

Attribute X	Attribute Y
M1	C1
M1	C2
M1	C3
M1	C4
M2	C1
M2	C3
M2	C5
M3	C3
M3	C4
M4	C1
M4	C2
M4	C3
M4	C4
M4	C5
M4	C6

and a relation B with only one attribute (attribute Y). Assume that attribute Y of relation A and the attribute of relation B are defined on a common domain. What would be the result of A divided by B if:

1. B = Attribute Y C1
2. B = Attribute Y C2 C3

Review questions

1. Briefly describe the theoretical foundations of Relational database systems.
2. Describe what is meant if a data item contains the value 'null'.
3. Why is it necessary sometimes to have a primary key that consists of more than one attribute?

4. What happens if you test two attributes, each of which contains the value null, to find out if they are equal?
5. What is the Entity Integrity Rule?
6. How are tables linked together in the Relational model?
7. What is Relational Algebra?
8. Explain the concept of union compatibility.
9. Describe the operation of the Relational Algebra operators RESTRICT, PROJECT, JOIN and DIVIDE.

Discussion topics

1. Now that you have been introduced to the structure of the Relational model, and having seen important mechanisms such as primary keys, domains, foreign keys and the use of null values, discuss what you feel at this point to be the strengths and weaknesses of the model. Bear in mind that, although the Relational Algebra is a part of the Relational model, it is not generally the language used for manipulating data in commercial database products. That language is SQL, which will be covered in subsequent chapters.
2. Consider the operations of Relational Algebra. Why do you think Relational Algebra is not used as a general approach to querying and manipulating data in Relational databases? Given that it is not used as such, what value can you see in the availability of a language for manipulating data which is not specific to any one developer of database systems?

Additional content and activities

As we have seen, the Relational Algebra is a useful, vendor-independent, standard mechanism for discussing the manipulation of data. We have seen, however, that the Relational Algebra is rather procedural and manipulates data one step at a time. Another vendor-independent means of manipulating data has been developed, known as Relational Calculus. For students interested in investigating the language aspect of the Relational model further, it would be valuable to compare what we have seen so far of the Relational Algebra, with the approach used in Relational Calculus. Indeed, it possible to map expressions between the Algebra and the Calculus, and it has also been shown that it is possible to convert any expression in one language to an equivalent expression in the other. In this sense, the Algebra and Calculus are formally equivalent.

Chapter 3. Introduction to SQL

Table of contents

Objectives

At the end of this chapter you should be able to:

- Write SQL queries to examine the data in the rows and columns of relational tables.

- Use string, arithmetic, date and aggregate functions to perform various calculations or alter the format of the data to be displayed.
- Sort the results of queries into ascending or descending order.
- Understand the significance of NULL entries and be able to write queries that deal with them.

Introduction to SQL

In parallel with this chapter, you should read Chapter 5 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter introduces the fundamentals of the Structured Query Language, SQL, which is a worldwide standard language for the querying and manipulation of Relational databases. This chapter covers the basic concepts of the language, and sufficient information for you to write simple but powerful queries. The further chapters on the SQL language will build on this knowledge, covering more complex aspects of the query language and introducing statements for adding, changing and removing data and the tables used to contain data. The material you will cover in the SQL chapters provides you with a truly transferable skill, as the language constructs you will learn will work in virtually all cases, unchanged, across a wide range of Relational systems.

Context

This unit presents the basics of the SQL language, and together with the succeeding units on SQL, provides a detailed introduction to the SQL language. The unit relates to the information covered on Relational Algebra, in that it provides a practical example of how the operations of the algebra can be made available within a higher level, non-procedural language. This chapter also closely relates to the material we will later cover briefly on query optimisation in a chapter called Database Administration and Tuning, as it provides the basic concepts needed to understand the syntax of the language, which is the information on which the query optimisation software operates.

There are a number of SQL implementations out there, including Microsoft Access (part of the Office suite), Microsoft SQL server and Oracle. There are also some open-source ones such as MySQL. You should make sure you have an SQL implementation installed for this chapter. Consult the course website for more information about the recommended and/or compatible SQL implementations. Although SQL commands in these notes are written in generic terms, you should be mindful that SQL implementations are different and sometimes what is given here may not work, or will work with slight modification. You should consult

the documentation of your software on the particular command should what is given here not work with your SQL implementation.

SQL overview

SQL is a language that has been developed specifically for querying and manipulating data in database systems. Its facilities reflect this fact; for example, it is very good for querying and altering sets of database records collectively in one statement (this is known as set-level processing). On the other hand, it lacks some features commonly found in general programming languages, such as LOOP and IF…THEN…ELSE statements.

SQL stands for Structured Query Language, and indeed it does have a structure, and is good for writing queries. However, it is structured rather differently to most traditional programming languages, and it can be used to update information as well as for writing queries.

SQL, as supported in most database systems, is provided via a command-line interface or some sort of graphical interface that allows for the text-based entry of SQL statements. For example, the following SQL statement is a query that will list the names of departments from a database table (also known as a relation) called DEPT:

SELECT DNAME FROM DEPT;

SQL language consists of three major components:

- **DDL (data definition language):** Used to define the way in which data is stored.
- **DML (data manipulation language):** Allows retrieval, insertion of data, etc. (This is sometimes called the 'query' language.)
- **DCL (data control language):** Used to control access to the data. For example, granting access to a user to insert data in a particular table.

The query language (DML) is very flexible in that it can be used to express quite complicated queries, sometimes very concisely.

One initial problem that people just starting to learn the language encounter is that it can sometimes be difficult to tell how hard a query will be to express in SQL from its natural language specification. That is, some queries that sound as though they will be hard to code in SQL from their description in a natural language such as English, turn out to be very straightforward. However, some simple-sounding queries turn out to be surprisingly difficult.

As you work through the SQL chapters in this module, you will build up experience and knowledge of the kinds of queries that are straightforward to write in SQL.

The data manipulation language (DML) of SQL allows the retrieval, insertion, updating and removal of rows stored in relational tables. As mentioned above, numbers of rows can be altered in any one statement, and so DML is a very powerful tool.

The data definition language (DDL) is used to create, change the structure of or remove whole tables and other relational structures. So whereas you would use the INSERT statement of the DML to insert new rows into an existing table, you would use the DDL CREATE TABLE statement to establish a new table in the first place.

The data control language (DCL) defines activities that are not in the categories of those for the DDL and DML, such as granting privileges to users, and defining when proposed changes to a databases should be irrevocably made.

The example company database

Throughout this and the succeeding chapters on SQL, we are going to use a standard pair of tables and set of data on which to write SQL statements. This standard data set comprises the tables EMP and DEPT. The structure of each is first described, and then the example records for each are presented.

The EMP table

The EMP table stores records about company employees. This table defines and contains the values for the attributes EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM and DEPTNO.

- EMPNO is a unique employee number; it is the primary key of the employee table.
- ENAME stores the employee's name.
- The JOB attribute stores the name of the job the employee does.
- The MGR attribute contains the employee number of the employee who manages that employee. If the employee has no manager, then the MGR column for that employee is left set to null.
- The HIREDATE column stores the date on which the employee joined the company.
- The SAL column contains the details of employee salaries.
- The COMM attribute stores values of commission paid to employees. Not all employees receive commission, in which case the COMM field is set to null.

- The DEPTNO column stores the department number of the department in which each employee is based. This data item acts a foreign key, linking the employee details stored in the EMP table with the details of departments in which employees work, which are stored in the DEPT table.

The DEPT table

The DEPT table stores records about the different departments that employees work in. This table defines and contains the values for the attributes as follows:

- DEPTNO: The primary key containing the department numbers used to identify each department.
- DNAME: The name of each department.
- LOC: The location where each department is based.

The data contained in the EMP and DEPT tables

The data in the EMP table contains the following 14 rows:

EMPNO	ENAME	JOB	HIREDATE	MGR	SAL	COMM	DEPTNO
7369	SMITH	CLERK	17-DEC-80	7902	800		20
7499	ALLEN	SALESMAN	20-FEB-81	7698	1600	300	30
7521	WARD	SALESMAN	22-FEB-81	7698	1250	500	30
7566	JONES	MANAGER	02-APR-81	7839	2975		20
7654	MARTIN	SALESMAN	28-SEP-81	7698	1250	1400	30
7698	BLAKE	MANAGER	01-MAY-81	7839	2850		30
7782	CLARK	MANAGER	09-JUN-81	7839	2450		10
7788	SCOTT	ANALYST	19-APR-87	7566	3000		20
7839	KING	PRESIDENT	17-NOV-81		5000		10
7844	TURNER	SALESMAN	08-SEP-81	7698	1500	0	30
7876	ADAMS	CLERK	23-MAY-87	7788	1100		20
7900	JAMES	CLERK	03-DEC-81	7698	950		30
7902	FORD	ANALYST	03-DEC-81	7566	3000		20
7934	MILLER	CLERK	23-JAN-82	7782	1300		10

The DEPT table contains the following four rows:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SQL SELECT statement

SQL queries can be written in upper or lower case, and on one or more lines. All queries in SQL begin with the word SELECT. The most basic form of the SELECT statement is as follows:

SELECT <select-list> FROM <table-list>

It is often useful to separate the different parts of a query onto different lines, so we might write this again as:

SELECT <select-list>

FROM <table-list>

Following the SELECT keyword is the list of table columns that the user wishes to view. This list is known as the select-list. As well as listing the table columns to be retrieved by the query, the select-list can also contain various SQL functions to process the data; for example, to carry out calculations on it. The select-list can also be used to specify headings to be displayed above the data values retrieved by the query. Multiple select-list items are separated from each other with commas. The select-list allows you to filter out the columns you don't want to see in the results.

The FROM keyword is, like the SELECT keyword, mandatory. It effectively terminates the select-list, and is followed by the list of tables to be used by the query to retrieve data. This list is known as the table-list. The fact that the tables need to be specified in the table-list means that, in order to retrieve data in SQL, you do need to know in which tables data items are stored. This may not seem surprising from the perspective of a programmer, or database developer, but what about an end-user? SQL has, in some circles, been put forward as a language that can be learned and used effectively by business users. We can see even at this early stage, however, that a knowledge of what data is stored where, at least at the logical level, is fundamental to the effective use of the language.

Exercise 1 - Fundamentals of SQL query statements

1. What keyword do all SQL query statements begin with?
2. What is the general form of simple SQL query statements?

Simple example queries

Sample query 1 - the names of all employees

Suppose we wish to list the names of all employees. The SQL query would be:

SELECT ENAME

FROM EMP;

The single ENAME column we wish to see is the only entry in the select-list in this example. The employee names are stored in the EMP table, and so the EMP table must be put after the keyword FROM to identify from where the employee names are to be fetched.

Note that the SQL statement is terminated with a semi-colon (;). This is not strictly part of the SQL standard. However, in some SQL environments, it ensures that the system runs the query after it has been entered.

The result of this query when executed is as follows (note that your system might reflect this in a different way to what is shown here):

ENAME
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
ADAMS
JAMES
FORD
MILLER

As you can see, the query returns a row for each record in the table, each row containing a single column presenting the name of the employee (i.e. the value of the DNAME attribute for each EMP record).

Sample query 2 - all data (rows and columns) from the DEPT table

There are two usual ways to list all data in a table. The simplest is to use a shorthand notation provided in SQL to list all the columns in any table. This is done simply by specifying an asterisk '*' for the select-list as follows:

SELECT *

FROM DEPT;

The asterisk is called a wild card, and causes all attributes of the specified table to be retrieved by the query.

Note that as it is the details of the DEPT table we wish to view, it is the DEPT table this time that appears in the table-list following the FROM keyword.

The use of * in this way is a very easy way to view the entire contents of any table. The alternative approach is simply to list all of the columns of the DEPT table in the select-list as follows:

SELECT DEPTNO, DENAME, LOC

FROM DEPT;

The result of executing either of these queries on our DEPT table at this time is the following:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

A potential problem of using the asterisk wild card, is that instead of explicitly listing all the attributes we want, the behaviour of the query will change if the table structure is altered — for example, if we add new attributes to the DEPT table, the SELECT * version of the query will then list the new attributes. This is a strong motivation for avoiding the use of the asterisk wild card in most situations.

Sample query 3 - the salary and commission of all employees

If we wish to see details of each employee's salary and commission we would use the following query that specifies just those attributes we desire:

SELECT EMPNO, ENAME, SAL, COMM

FROM EMP;

In this example, we have included the EMPNO column, just in case we had any duplicate names among the employees.

The result of this query is:

EMPNO	ENAME	SAL	COMM
7369	SMITH	800	
7499	ALLEN	1600	300
7521	WARD	1250	500
7566	JONES	2975	
7654	MARTIN	1250	1400
7698	BLAKE	2850	
7782	CLARK	2450	
7788	SCOTT	3000	
7839	KING	5000	
7844	TURNER	1500	0
7876	ADAMS	1100	
7900	JAMES	950	
7902	FORD	3000	
7934	MILLER	1300	

Calculating values and naming query columns

Sample query 4 - example calculation on a select-list

In the queries we have presented so far, the data we have requested has been one or more attributes present in each record. Following the principle of reducing data redundancy, many pieces of information that are useful, and that can be calculated from other stored data, are not stored explicitly in databases. SQL queries can perform a calculation 'on-the-fly' using data from table records to present this kind of information.

The salary and commission values of employees we shall assume to be monthly. Suppose we wish to display the total annual income (including commission) for each employee. This figure for each employee is not stored in the table, since it can be calculated from the monthly salary and commission values. The calculation is simply 12 times the sum of the monthly salary and commission.

A query that retrieves the number and name of each employee, and calculates their annual income, is as follows:

SELECT EMPNO, ENAME, 12 * (SAL + COMM)

FROM EMP;

The calculation here adds the monthly commission to the salary, and then multiplies the result by 12 to obtain the total annual income.

EMPNO	ENAME	12*(SAL + COMM)
7499	ALLEN	22800
7521	WARD	21000
7654	MARTIN	31800
7844	TURNER	18000

Notice that only records for which the commission value was not NULL have been included. This issue is discussion later in the chapter. When using some SQL implementation, such as MS Access, you may have to explicitly request records with NULL values to be excluded. So the above SQL query:

SELECT EMPNO, ENAME, 12 * (SAL + COMM)

FROM EMP;

may need to be written as:

SELECT EMPNO, ENAME, 12 * (SAL + COMM)

FROM EMP

WHERE COMM IS NOT NULL;

(See later to understand the WHERE part of this query)

Depending on which SQL system you run a query like this, the calculated column may or may not have a heading. The column heading may be the expression itself 12 * (SAL + COMM) or may be something indicating that an expression has been calculated: Expr1004 (these two examples are what happens in Oracle and MS Access respectively). Since such calculations usually mean something in particular (in this case, total annual income), it makes sense to name these calculated columns sensibly wherever possible.

Altering the column headings of query results

Sometimes it is desirable to improve upon the default column headings for query results supplied by the system, to make the results of queries more intelligible. For example, the result of a query to calculate annual pay by summing the monthly salary and commission and multiplying by 12, would by default in some systems such as Oracle, have the expression of the calculation as the column heading. The result is more readable, however, if we supply a heading which clearly states what the compound value actually is, i.e. annual income. To do this, simply include the required header information, in double quotes, after the column specification in the select-list. For the annual pay example, this would be:

SELECT EMPNO, ENAME, 12*(SAL + COMM) “ANNUAL INCOME”

FROM EMP;

The result is more meaningful:

EMPNO	ENAME	ANNUAL INCOME
7499	ALLEN	22800
7521	WARD	21000
7654	MARTIN	31800
7844	TURNER	18000

Once again, there are alternative ways to achieve the naming of columns in some systems including MS Access and MySQL, rather than using the double quotation marks around the column heading. The use of the keyword AS and square brackets may also be required.

So the SQL query:

SELECT EMPNO, ENAME, 12*(SAL + COMM) “ANNUAL INCOME”

FROM EMP;

may need to be written in as:

SELECT EMPNO, ENAME, 12*(SAL + COMM) AS ANNUAL INCOME

FROM EMP WHERE COMM IS NOT NULL;

(See next section to understand the WHERE part of this query)

The WHERE clause

Very often we wish to filter the records/rows retrieved by a query. That is, we may only wish to have a subset of the records of a table returned to us by a query.

The reason for this may be, for example, in order to restrict the employees shown in a query result just to those employees with a particular job, or with a particular salary range, etc. Filtering of records is achieved in SQL through the use of the WHERE clause. In effect, the WHERE clause implements the functionality of the RESTRICT operator from Relational Algebra, in that it takes a horizontal subset of the data over which the query is expressed.

Basic syntax of the WHERE clause

The WHERE clause is not mandatory, but when it is used, it must appear following the table-list in an SQL statement. The clause consists of the keyword WHERE, followed by one or more restriction conditions, each of which are separated from one another using the keywords AND or OR.

The format of the basic SQL statement including a WHERE clause is therefore:

SELECT <select-list> FROM <table-list>

[WHERE <condition1> <, AND/OR CONDITION 2, .. CONDITION n>]

The number of conditions that can be included within a WHERE clause varies from DBMS to DBMS, though in most major commercial DBMS, such as Oracle, Sybase, Db2, etc, the limit is so high that it poses no practical restriction on query specifications. We can also use parentheses '(' and ')' to nest conditions or improve legibility.

Examples of using the WHERE clause

WHERE example 1 - records with a value before some date

If we wish to retrieve all of those employees who were hired before, say, May 1981, we could issue the following query:

SELECT EMPNO, ENAME, HIREDATE

FROM EMP

WHERE HIREDATE < '01-MAY-1981';

The result of this query is:

EMPNO	ENAME	HIREDATE
7369	SMITH	17-Dec-80
7499	ALLEN	20-Feb-81
7521	WARD	22-Feb-81
7566	JONES	02-Apr-81

Note, incidentally, the standard form in which some systems such as Oracle handle dates: they are enclosed in single quotes, and appear as: DD-MMM-YYYY (two digits for day 'dd', three letters for month 'mmm' and four digits for year 'yyyy'). In some systems including MS Access, the date should be enclosed with two hash '#' characters, rather than single quotes - for example, #01-JAN-1990#. You should check with your system's documentation for the requirement as to how the dates should be formatted. Below are the two versions of the SQL statements, with different formats for dates:

For systems including Oracle:

SELECT EMPNO, ENAME, HIREDATE

FROM EMP

WHERE HIREDATE < '01-MAY-1981';

For systems including MS Access:

SELECT EMPNO, ENAME, HIREDATE

FROM EMP

WHERE HIREDATE < #01-MAY-1981#;

In our example above, we used the < (less than) arithmetic symbol to form the condition in the WHERE clause. In SQL, the following simple comparison operators are available:

= equals

!= is not equal to (allowed in some dialects)

< > is not equal to (ISO standard)

< = is less than or equal to

< is less than

> = is greater than or equal to

> is greater than

WHERE example 2 - two conditions that must both be true

The logical operator AND is used to specify that two conditions must both be true. When a WHERE clause has more than one condition, this is called a compound condition.

Suppose we wish to retrieve all salesmen who are paid more than 1500 a month. This can be achieved by ANDing the two conditions (is a salesman, and is paid more than 1500 a month) together in a WHERE clause as follows:

SELECT EMPNO, ENAME, JOB, SAL

FROM EMP

WHERE JOB = 'SALESMAN' AND SAL > 1500;

The result of this query is:

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	1600

Only employees fulfilling both conditions will be returned by the query. Note the way in which the job is specified in the WHERE clause. This is an example of querying the value of a field of type character, or as it is called in Oracle, of type varchar2. When comparing attributes against fixed values of type character such as SALESMAN, the constant value being compared must be contained in single quotes, and must be expressed in the same case as it appears in the table. All of the data in the EMP and DEPT tables is in upper case, so when we are comparing character values, we must make sure they are in upper case for them to match the values in the EMP and DEPT tables. In other words, from a database point of view, the job values of SALESMAN and salesman are completely different, and if we express a data item in lower case when it is stored in upper case in the database, no match will be found.

In some systems, including MS Access, the text an attribute is to match should be enclosed with double quote characters, rather than single quotes. For example, "SALESMAN" rather than 'SALESMAN':

SELECT EMPNO, ENAME, JOB, SAL

FROM EMP

WHERE JOB = "SALESMAN" AND SAL > 1500;

WHERE example 3 - two conditions, one of which must be true

The logical operator OR is used to specify that at least one of two conditions must be true.

For example, if we wish to find employees who are based in either department 10 or department 20, we can do it by issuing two conditions in the WHERE clause as follows:

SELECT EMPNO, ENAME, DEPTNO

FROM EMP

WHERE DEPTNO = 10 OR DEPTNO = 20;

The result of this query is:

EMPNO	ENAME	DEPTNO
7369	SMITH	20
7566	JONES	20
7782	CLARK	10
7788	SCOTT	20
7839	KING	10
7876	ADAMS	20
7902	FORD	20
7934	MILLER	10

The use of NOT

The keyword NOT can be used to negate a condition, i.e. only records that do not meet a condition are selected. An example might be to list all employees who are not salesmen:

SELECT EMPNO, ENAME, JOB, SAL

FROM EMP

WHERE NOT(JOB = 'SALESMAN');

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	800
7566	JONES	MANAGER	2975
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7788	SCOTT	ANALYST	3000
7839	KING	PRESIDENT	5000
7876	ADAMS	CLERK	1100
7900	JAMES	CLERK	950
7902	FORD	ANALYST	3000
7934	MILLER	CLERK	1300

Another example might be to list all employees who do not earn more than 1500:

SELECT EMPNO, ENAME, JOB, SAL

FROM EMP

WHERE NOT(SAL > 1500);

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	800
7566	JONES	MANAGER	2975
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7788	SCOTT	ANALYST	3000
7839	KING	PRESIDENT	5000
7876	ADAMS	CLERK	1100
7900	JAMES	CLERK	950
7902	FORD	ANALYST	3000
7934	MILLER	CLERK	1300

The use of !=

The operator != can be used to select where some value is NOT EQUAL TO some other value. So another way to write the query:

SELECT EMPNO, ENAME, JOB, SAL

FROM EMP

WHERE NOT(JOB = 'SALESMAN');

is as follows:

SELECT EMPNO, ENAME, JOB, SAL

FROM EMP

WHERE JOB != 'SALESMAN';

Retrieving from a list of values

An alternative solution to the previous OR example is provided by a variation on the syntax of the WHERE clause, in which we can search for values contained in a specified list. This form of the WHERE clause is as follows:

WHERE ATTRIBUTE IN (<item1>, <item2>, ..., <itemN>)

Using this syntax, the previous query would be rewritten as follows:

SELECT EMPNO, ENAME, DEPTNO

FROM EMP

WHERE DEPTNO IN (10, 20);

The result of the query is just the same, but in many cases this form of the WHERE clause is both shorter and simpler to use.

Querying over a range of values

The BETWEEN keyword can be used in a WHERE clause to test whether a value falls within a certain range. The general form of the WHERE clause using the BETWEEN keyword is:

WHERE <attribute> BETWEEN <value1> AND <value2>

The operands <value1> and <value2> can either be literals, like 1000, or expressions referring to attributes.

For example, if we wish to test for salaries falling in the range 1000 to 2000, then we can code as follows:

SELECT EMPNO, ENAME, SAL

FROM EMP

WHERE SAL BETWEEN 1000 AND 2000;

The result of this query is:

EMPNO	ENAME	SAL
7499	ALLEN	1600
7521	WARD	1250
7654	MARTIN	1250
7844	TURNER	1500
7876	ADAMS	1100
7934	MILLER	1300

Note that the BETWEEN operator is inclusive, so a value of 1000 or 2000 would satisfy the condition and the record included in the query result.

An equally valid solution could have been produced by testing whether the salaries to be returned were >=1000 and <=2000, in which case, the WHERE clause would have been:

SELECT EMPNO, ENAME, SAL

FROM EMP

WHERE (SAL >=1000) AND (SAL <=2000);

However, this version of the query is longer and more complex, and includes the need to repeat the SAL attribute for comparison in the second condition of the WHERE clause.

In general, the solution using BETWEEN is preferable since it is more readable - it is clearer to a human reading the SQL query code what condition is being evaluated.

Searching for partial matches

All of the queries we have seen so far have been to retrieve exact matches from the database. The LIKE keyword allows us to search for items for which we only know a part of the value. The LIKE keyword in SQL literally means 'is approximately equal to' or 'is a partial match with'. The keyword LIKE is used in conjunction with two special characters which can be used as wild card matches - in other words, LIKE expressions can be used to identify the fact that we do not know precisely what a part of the retrieved value is.

LIKE example - search for words beginning with a certain letter

As an example, we can search for all employees whose names begin with the letter S as follows:

SELECT EMPNO, ENAME

FROM EMP

WHERE ENAME LIKE 'S%";

This query returns:

EMPNO	ENAME
7369	SMITH
7788	SCOTT

Here the percentage sign (%) is used as a wild card, to say that we do not know or do not wish to specify the rest of the value of the ENAME attribute; the only criteria we are specifying is that it begins with 'S', and it may be followed by no, one or more than one other character.

The percentage sign can be used at the beginning or end of a character string, and can be used as a wild card for any number of characters.

The other character that can be used as a wild card is the underline character (_). This character is used as a wild card for only one character per instance of the underline character. That is, if we code:

WHERE ENAME LIKE 'S___';

the query will return employees whose names start with S, and have precisely two further characters after the S. So employees called Sun or Sha would be returned, but employee names such as Smith or Salt would not be, as they do not contain exactly three characters.

Note that we can combine conditions using BETWEEN, or LIKE, with other conditions such as simple tests on salary, etc, by use of the keywords AND and OR, just as we can combine simple conditions. However, wild card characters cannot be used to specify members of a list with the IN keyword.

Sorting data

Data can very easily be sorted into different orders in SQL. We use the ORDER BY clause. This clause is optional, and when required appears as the last clause in a query. The ORDER BY keywords are followed by the attribute or attributes on which the data is to be sorted. If the sort is to be done on more than one attribute, the attributes are separated by commas.

The general form of an SQL query with the optional WHERE and ORDER BY clauses looks as follows:

SELECT <select-list> FROM <table-list>

[WHERE <condition1> <, AND/OR CONDITION 2, .. CONDITION n>] [ORDER BY <attribute-list>]

An example would be to sort the departments into department number order:

SELECT DEPTNO, DNAME

FROM DEPT

ORDER BY DEPTNO;

OR

SELECT DEPTNO, DNAME

FROM DEPT

ORDER BY DEPTNO ASC;

Note: SQL provides the keyword ASC to explicitly request ordering in ascending order.

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

Or to sort into alphabetical order of the name of the department:

SELECT DEPTNO, DNAME

FROM DEPT

ORDER BY DNAME;

DEPTNO	DNAME
10	ACCOUNTING
40	OPERATIONS
20	RESEARCH
30	SALES

Descending order

SQL provides the keyword DESC to request sorting in the reverse order. So to sort the departments into reverse alphabetical order, we can write the following:

SELECT DEPTNO, DNAME

FROM DEPT

ORDER BY DNAME DESC;

DEPTNO	DNAME
30	SALES
20	RESEARCH
40	OPERATIONS
10	ACCOUNTING

A sort within a sort

It is very easy to specify a sort within a sort, i.e. to first sort a set of records into one order, and then within each group to sort again by another attribute.

For example, the following query will sort employees into department number order, and within that, into employee name order.

SELECT EMPNO, ENAME, DEPTNO

FROM EMP

ORDER BY DEPTNO, ENAME;

The result of this query is:

EMPNO	ENAME	DEPTNO
7782	CLARK	10
7839	KING	10
7934	MILLER	10
7876	ADAMS	20
7902	FORD	20
7566	JONES	20
7788	SCOTT	20
7369	SMITH	20
7499	ALLEN	30
7698	BLAKE	30
7900	JAMES	30
7654	MARTIN	30
7844	TURNER	30
7521	WARD	30

As can be seen, the records have been sorted into order of DEPTNO first, and then for each DEPTNO, the records have been sorted alphabetically by ENAME. This can be easily seen if you have a repeating DEPTNO - for example, if we had two employees, WARD and KUDO, belonging to DEPTNO 7521. Two DEPTNO 7521 will appear at the end of the table like above, but KUDO will be on top of WARD under the ENAME column.

When a query includes an ORDER BY clause, the data is sorted as follows:

- Any null values appear first in the sort
- Numbers are sorted into ascending numeric order
- Character data is sorted into alphabetical order
- Dates are sorted into chronological order

We can include an ORDER BY clause with a WHERE clause, as in the following

example, which lists all salesman employees in ascending order of salary:

SELECT EMPNO,ENAME,JOB,SAL

FROM EMP

WHERE JOB = 'SALESMAN'

ORDER BY SAL;

EMPNO	ENAME	JOB	SAL
7654	MARTIN	SALESMAN	1250
7521	WARD	SALESMAN	1250
7844	TURNER	SALESMAN	1500
7499	ALLEN	SALESMAN	1600

Handling NULL values in query results (the NVL function)

In the chapter introducing the Relational model, we discussed the fact that NULL values represent the absence of any actual value, and that it is correct to refer to an attribute being set to NULL, rather than being equal to NULL. The syntax of testing for NULL values in a WHERE clause reflects this. Rather than coding WHERE X = NULL, we write WHERE X IS NULL, or, WHERE X IS NOT NULL.

WHERE clauses using IS NULL and IS NOT NULL

For example, to return all employees who do not receive a commission, the query would be:

SELECT EMPNO, ENAME, SAL

FROM EMP

WHERE COMM IS NULL;

EMPNO	ENAME	SAL
7369	SMITH	800
7566	JONES	2975
7698	BLAKE	2850
7782	CLARK	2450
7788	SCOTT	3000
7839	KING	5000
7876	ADAMS	1100
7900	JAMES	950
7902	FORD	3000
7934	MILLER	1300

We can also select records that do not have NULL values:

SELECT EMPNO, ENAME, SAL, COMM

FROM EMP

WHERE COMM IS NOT NULL;

EMPNO	ENAME	SAL	COMM
7499	ALLEN	1600	300
7521	WARD	1250	500
7654	MARTIN	1250	1400
7844	TURNER	1500	0

The NVL function

There is an extremely useful function available for the handling of NULLs in query results. (It is important to remember that NULL is not the same as, say, zero for a numeric attribute.) This is the NVL function, which can be used to substitute other values in place of NULLs in the results of queries. This may be required for a number of reasons:

- By default, arithmetic and aggregate functions ignore NULL values in query results. Sometimes this is what is required, but at other times we might explicitly wish to consider a NULL in a numeric column as actually representing the value zero, for example.
- We may wish to replace a NULL value, which will appear as a blank column in the displayed results of a query, with a more explicit indication that there was no value for that column instance.

The format of the NVL function is:

NVL(<column>, <value>)

<column> is the attribute in which NULLs are to be replaced, and <value> is the substitute value.

Examples of using the NVL function

An example of using NVL to treat all employees with NULL commissions as if they had zero commission:

SELECT EMPNO,NVL(COMM, 0)

FROM EMP;

To display the word 'unassigned' wherever a NULL value is retrieved from the JOB attribute:

SELECT EMPNO,NVL(job, 'unassigned')

FROM EMP;

Exercise

What would happen in the cases of employees who do not receive a commission, i.e. whose commission attribute is set to NULL?

Answer: The short, and somewhat surprising answer to this question, is that the records of employees receiving NULL commission will simply not be included in the result. The reason for this is that as we saw in the chapter on the Relational model, NULL simply indicates the absence of a real value, and so the result of adding a salary value to a NULL commission value is indeterminate. For this reason, SQL cannot return a value for the annual pay of employees where those employees receive no commission. There is a very useful solution to this problem,

which will be dealt with later in this chapter, under the heading "Handling NULL values in query results".

REFERENCE MATERIAL: SQL functions

SQL functions help simplify different types of operations on the data. SQL supports four types of functions:

- Arithmetic functions
- Character functions
- Date functions
- Aggregate functions

The functions are used as part of a select-list of a query, or if they refer to a specific row, they may be used in a WHERE clause. They are used to modify the values or format of data being retrieved.

Arithmetic functions

The most commonly used arithmetic functions are as follows:

- **greatest**
- greatest(object-list) - returns the greatest of a list of values

  Example:

  greatest(sal,comm) - returns whichever of the SAL or COMM attributes has the highest value
- **least**
- least(object-list) - returns the smallest of a list of values

  Example:

  least(sal,comm) - returns whichever of the SAL or COMM attributes has the lowest value
- **round**
- round(number[,d]) - rounds the number to d digits right of the decimal point (d can be negative)

  Example:

  round(sal,2) - rounds values of the SAL attribute to two decimal places
- **trunc**

- Trunc(number,d) – truncates number to d decimal places (d can be negative)

  Example:

  trunc(sal,2) - truncates values of the SAL attribute to two decimal places

  Note: The difference between the round and truncate functions is that round will round up digits of five or higher, whilst trunc always rounds down.

- **abs**
- abs(number) - returns the absolute value of the number

  Example:

  abs(comm-sal) - returns the absolute value of COMM - SAL; that is, if the number returned would be negative, the minus sign is discarded

- **sign**
- sign(number) - returns 1 if number greater than zero, 0 if number = zero, -1 if number less than zero

  Example:

  sign(comm-sal) - returns 1 if COMM - SAL $>$ 0, 0 if COMM - SAL = 0, and - 1 if COMM - SAL $<$ 0

- **mod**
- mod(number1,number2) - returns the remainder when number1 is divided by number2

  Example:

  mod(sal,comm) - returns the remainder when SAL is divided by COMM

- **sqrt**
- sqrt(number) - returns the square root of the number. If the number is less than zero then sqrt returns null

  Example:

  sqrt(sal) - returns the square root of salaries

- **to_char**
- to_char(number[picture]) - converts a number to a character string in the format specified

  Example:

  to_char(sal,9999.99) - represents salary values with four digits before the decimal point, and two afterwards

- **decode**
- decode(column,starting-value,substituted-value..) - substitutes alternative values for a specified column

  Example:

  decode(comm,100,200,200,300,100) - returns values of commission increased by 100 for values of 100 and 200, and displays any other comm values as if they were 100
- **ceil**
- ceil(number) - rounds up a number to the nearest integer

  Example:

  ceil(sal) - rounds up salaries to the nearest integer
- **floor**
- floor(number) - truncates the number to the nearest integer

  Example:

  floor(sal) - rounds down salary values to the nearest integer

Character functions

The most commonly used character string functions are as follows:

- **string1 || string2**
- string1 || string2 - concatenates (links) string1 with string2

  Example:

  deptno || empno - concatenates the employee number with the department number into one column in the query result
- **decode**
- decode(column,starting-value,substitute-value, ….) - translates column values to specified alternatives. The final parameter specifies the value to be substituted for any other values.

  Example:

  decode(job,'CLERK','ADMIN WORKER','MANAGER','BUDGET HOLDER',PRESIDENT','EXECUTIVE','NOBODY') This example translates values of the JOB column in the employee table to alternative values, and represents any other values with the string 'NOBODY'.
- **distinct**

- distinct <column> - lists the distinct values of the specific column

  Example:

  Distinct job - lists all the distinct values of job in the JOB attribute

- **length**
- length(string) - finds number of characters in the string

  Example:

  length(ename) - returns the number of characters in values of the ENAME attribute

- **substr**
- substr(column,start-position[,length]) - extracts a specified number of characters from a string

  Example:

  substr(ename,1,3) - extracts three characters from the ENAME column, starting from the first character

- **instr**
- instr(string1,string2[,start-position]) - finds the position of string2 in string1. The parentheses around the start-position attribute denote that it is optional

  Example:

  instr(ename,'S') - finds the position of the character 'S' in values of the ENAME attribute

- **upper**
- upper(string) - converts all characters in the string to upper case

  Example:

  upper(ename) - converts values of the ENAME attribute to upper case

- **lower**
- lower(string) - converts all characters in the string to lower case

  Example:

  lower(ename) - converts values of the ENAME attribute to lower case

- **to_number**
- to_number(string) - converts a character string to a number

  Example:

  to_number('11') + sal - adds the value 11 to employee salaries

- **to_date**
- to_date(str[,pict]) - converts a character string in a given format to a date

  Example:

  to_date('14/apr/99','dd/mon/yy') - converts the character string '14/apr/99' to the standard system representation for dates
- **soundex**
- soundex(string) - converts phonetically similar strings to the same value

  Example:

  soundex('smith') - converts all values that sound like the name Smith to the same value, enabling the retrieval of phonetically similar attribute values
- **vsize**
- vsize(string) - finds the number of characters required to store the character string

  Example:

  vsize(ename) - returns the number of bytes required to store values of the ENAME attribute
- **lpad**
- lpad(string,len[,char]) - left pads the string with filler characters

  Example:

  lpad(ename,10) - left pads values of the ENAME attribute with filler characters (spaces)
- **rpad**
- rpad(string,len[,char]) - right pads the string with filler characters

  Example:

  rpad(ename,10) - right pads values of the ENAME attribute with filler characters (spaces)
- **initcap**
- initcap(string) - capitalises the initial letter of every word in a string

  Example:

  initcap(job) - starts all values of the JOB attribute with a capital letter
- **translate**

- translate(string,from,to) - translates the occurrences of the 'from' string to the 'to' characters

  Example:

  translate(ename,'ABC','XYZ') - replaces all occurrences of the string 'ABC' in values of the ENAME attribute with the string 'XYZ'

- **ltrim**
- ltrim(string,set) - trims all characters in a set from the left of the string

  Example:

  ltrim(ename,'') - removes all spaces from the start of values of the ENAME attribute

- **rtrim**
- rtrim(string,set) - trims all characters in the set from the right of the string

  Example:

  rtrim(job,'.') - removes any full-stop characters from the right-hand side of values of the JOB attribute

Date functions

The date functions in most commercially available database systems are quite rich, reflecting the fact that many commercial applications are very date driven. The most commonly used date functions in SQL are as follows:

- **Sysdate Sysdate** - returns the current date
- **add-months add-months(date, number)** - adds a number of months from/to a date (number can be negative). For example: add-months(hiredate, 3). This adds three months to each value of the HIREDATE attribute
- **months-between months-between(date1, date2)** - subtracts date2 from date1 to yield the difference in months. For example: months-between(sysdate, hiredate). This returns the number of months between the current date and the dates employees were hired
- **last-day last-day(date)** - moves a date forward to last day in the month. For example: last-day(hiredate). This moves hiredate forward to the last day of the month in which they occurred
- **next-day next-day(date,day)** - moves a date forward to the given day of week. For example: next-day(hiredate,'monday'). This returns all hiredates moved forward to the Monday following the occurrence of the hiredate

- **round round(date[,precision])** - rounds a date to a specified precision. For example: round(hiredate,'month'). This displays hiredates rounded to the nearest month
- **trunc trunc(date[,precision])** - truncates a date to a specified precision. For example: trunc(hiredate,'month'). This displays hiredates truncated to the nearest month
- **decode decode(column,starting-value,substituted-value)** - substitutes alternative values for a specified column. For example: decode(hiredate,'25-dec-99','christmas day',hiredate). This displays any hiredates of the 25th of December, 1999, as Christmas Day, and any default values of hiredate as hiredate
- **to_char to_char(date,[picture])** - outputs the data in the specified character format. The format picture can be any combination of the formats shown below. The whole format picture must be enclosed in single quotes. Punctuation may be included in the picture where required. Any text should be enclosed in double quotes. The default date format is: 'dd-mon-yy'. Example: numeric format, description
- cc, century, 20
- y,yyy, year, 1,986
- yyyy, year, 1986
- yyy, last three digits of year, 986
- yy, last two digits of year, 86
- y, last digits of year, 6
- q, quarter of year, 2
- ww, week of year, 15
- w, week of month, 2
- mm, month, 04
- ddd, day of year, 102
- dd, day of month, 14
- d, day of week, 7
- hh or hh12, hour of day (01-12), 02
- hh24, hour of day (01-24), 14
- mi, minutes, 10
- ss, seconds, 5
- sssss, seconds past midnight, 50465

- j, julian calendar day, 2446541

The following suffixes may be appended to any of the numeric formats (suffix, meaning, example):

- th, st, nd, rd, after the number, 14th
- sp, spells the number, fourteen
- spth/st/nd/rd, spells the number, fourteenth

There is also a set of character format pictures (character format, meaning, example):

- year, year, nineteen-eighty-six
- month, name of month, april
- mon, abbreviated month, apr
- day, day of week, saturday
- dy, abbreviated day, sat
- am or pm, meridian indicator, pm
- a.m. or p.m., meridian indicator, p.m.
- bc or ad, year indicator, ad
- b.c. or a.d., year indicator, a.d.

If you enter a date format in upper case, the actual value will be output in upper case. If the format is lower case, the value will be output in lower case. If the first character of the format is upper case and the rest lower case, the value will be output similarly.

`For example:`

```
to-char(hiredate,'dd/mon/yyyy')

to-char(hiredate,'day,"the"Ddspth"of"month')
```

Aggregate functions

All aggregate functions with the exception of COUNT operate on numerical columns. All of the aggregate functions below operate on a number of rows:

- **avg**
- avg(column) - computes the average value and ignores null values

  Example:

  SELECT AVG(SAL) FROM EMP;

Gives the average salary in the employee table, which is 2073.21

- **Sum**
- Sum(column) - computes the total of all the values in the specified column and ignores null values

  Example:

  sum(comm) - calculates the total commission paid to all employees
- **min**
- min(column) - finds the minimum value in a column

  Example:

  min(sal) - returns the lowest salary
- **max**
- max(column) - finds the maximum value in a column

  Example:

  max(comm) - returns the highest commission
- **count**
- count(column) - counts the number of values and ignores nulls

  Example:

  count(empno) - counts the number of employees
- **variance**
- variance(column) - returns the variance of the group and ignores nulls

  Example:

  variance(sal) - returns the variance of all the salary values
- **Stddev**
- Stddev(column) - returns the standard deviation of a set of numbers (same as square root of variance)

  Example:

  stddev(comm) - returns the standard deviation of all commission values

Activity - EMPLOYEE AND DEPARTMENT QUERIES

Using the EMP and DEPT tables, create the following queries in SQL, and test them to ensure they are retrieving the correct data.

You may wish to review the attributes of the EMP and DEPT tables, which are shown along with the data near the start of the section called Introduction to the SQL language.

1. List all employees in the order they were hired to the company.
2. Calculate the sum of all the salaries of managers.
3. List the employee numbers, names and hiredates of all employees who were hired in 1982.
4. Count the number of different jobs in the EMP table without listing them.
5. Find the average commission, counting only those employees who receive a commission.
6. Find the average commission, counting employees who do not receive a commission as if they received a commission of 0.
7. Find in which city the Operations department is located.
8. What is the salary paid to the lowest-paid employee?
9. Find the total annual pay for Ward.
10. List all employees with no manager.
11. List all employees who are not managers.
12. How many characters are in the longest department name?

Review questions

1. Distinguish between the select-list and the table-list in an SQL statement, explaining the use of each within an SQL statement.
2. What restrictions are there on the format and structure of the basic SQL queries as covered so far in this chapter? Describe the use of each of the major components of SQL query constructs that we have covered up to this point.
3. How are NULL values handled when data is sorted?
4. What facilities exist for formatting dates when output from an SQL statement?
5. What facilities are provided for analysing data in the same column across different rows in a table?
6. What is the role of the NVL function?

Discussion topics

1. Is SQL for end-users?

   As mentioned earlier in the chapter, a number of people in the database community believe that SQL is a viable language for end-users - that is, people whose jobs are not primarily involved with computing. From your introductory experience of the language so far, you should consider reasons for and against this view of the SQL language.

2. Can you think of any reasons why use of the wild card '*' as we have seen in a select-list may lead to problems?

Chapter 4. Intermediate SQL

Table of contents

Objectives

At the end of this chapter you should be able to:

- Group and summarise data together.
- Combine the grouping mechanisms with the aggregation functions covered in the previous chapter to provide useful summarised reports of data.
- Write queries that retrieve results by combining information from a number of tables.

- Combine the results of multiple queries in various ways.

Introduction

In parallel with this chapter, you should read Chapter 5 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter builds on the foundations laid in the previous chapter, which introduced the SQL language. We examine a range of facilities for writing more advanced queries of SQL databases, including queries on more than one table, summarising data, and combining the results of multiple queries in various ways.

Context

This chapter forms the bridge between the chapter in which the SQL language was introduced, and the coverage of the data definition language (DDL) and data control language (DCL) provided in the next chapter called Advanced SQL.

It is possible to express a range of complex queries using the data manipulation language (DML) previously introduced. The earlier chapter showed how fairly simple queries can be constructed using the select-list, the WHERE clause to filter rows out of a query result, and the ORDER BY clause to sort information. This chapter completes the coverage of the DML facilities of SQL, and will considerably increase the range of queries you are able to write. The final SQL chapter will then address aspects of SQL relating to the updating of data and the manipulation of the logical structures, i.e. tables that contain data.

Grouping and summarising information

Information retrieved from an SQL query can very easily be placed into separate groups or categories by use of the GROUP BY clause. The clause is similar in format to ORDER BY, in that the specification of the words GROUP BY is followed by the data item or items to be used for forming the groups. The GROUP BY is optional. If it appears in the query, it must appear before the ORDER BY if the ORDER BY is present.

Example: count the number of employees in each department

To answer this question, it is necessary to place the employees in the EMP table into separate categories, one for each department. This can be done easily enough through the use of the DEPTNO column in the EMP table as follows (with the select-list temporarily omitted):

SELECT ….

FROM EMP

GROUP BY DEPTNO

As far as counting the employees is concerned, this is an example of something that is seen very commonly with the GROUP BY clause; that is, the use of an aggregation function, in this case the COUNT function, in conjunction with GROUP BY. To complete the query, we simply need to include on the select-list the DEPTNO column, so that we can see what we are grouping by, and the COUNT function. The query then becomes:

SELECT DEPTNO,COUNT(EMPNO)

FROM EMP

GROUP BY DEPTNO;

DEPTNO	COUNT(EMPNO)
10	3
20	5
30	6

Comments: The query works in two steps. The first step is to group all employees by DEPTNO. The second step is to count the number of employees in each group. Of course, headings could be used to improve the clarity of the results; for example, specifying that the second column is "no. of employees".

```
COUNT(EMPNO) AS no. of employees
```

We have specified between the parentheses of the COUNT function that we are counting EMPNOs, because we are indeed counting employees. We could in fact have merely specified an asterisk, "*" in the parentheses of the COUNT function, and the system would have worked out that we were counting instances of records in the EMP table, which equates to counting employees. However, it is more efficient to specify to the system what is being counted.

GROUP BY, like ORDER BY, can include more than one data item, so for example if we specify:

GROUP BY DEPTNO, JOB

the results will be returned initially categorised within departments, and then within that, categorised into employees who do the same job.

GROUP BY example

Find the average salary for each JOB in the company:

```
SELECT JOB, AVG(round(SAL,2))
FROM EMP
GROUP BY JOB;
```

JOB	AVG(ROUND(SAL,2))
ANALYST	3000
CLERK	1037.5
MANAGER	2758.3333
PRESIDENT	5000
SALESMAN	1400

Comments: This is a fairly straightforward use of the GROUP BY clause, once again in conjunction with an aggregate function, AVG.

A very common error with GROUP BY

All column names in the select-list must appear in the GROUP BY clause, unless the name is used only in an aggregate function. Many people when first starting to use the GROUP BY clause, fall into the trap of asking the system to retrieve and display information at two different levels. This arises if, for example, you GROUP BY a data item such as JOB, but then on the select-list, include a data item at the individual employee level, such as HIREDATE, SAL, ETC. You might think that we displayed salaries in the previous example, where we listed the average salaries earned by employees doing the same job. It makes all the difference, however, that these are average salaries, the averages being calculated for each category that we are grouping by, in this case the average salary for each job. It is fine to display average salaries, as these are averaged across the group, and are therefore at the group level. However, if we had asked to display individual salaries, we would have had the error message "not a group by expression", referring to the fact that SAL is an individual attribute of an employee, and not in itself an item at the group level. Whenever you see the "not a group by expression" message, the first thing to check is the possibility that you have included a request on your select-list to view information at the individual record level, rather than at the group level. The one individual level item you can of course include on the select-list, is the item which is shared by

a number of individuals that you are in fact grouping by. So when we asked to retrieve the average salaries for each JOB, it was of course fine to include the JOB column in the select-list, because for that query, JOB is an item at the group level, i.e. we were grouping by JOB.

The HAVING clause

The HAVING clause is used to filter out specific groups or categories of information, exactly in the same way that the WHERE clause is used to filter out individual rows. The HAVING clause always follows a GROUP BY clause, and is used to test some property or properties of the grouped information.

For example, if we are grouping information at the department level, we might use a HAVING clause in which to exclude departments with less than a certain number of employees. This could be coded as follows:

```
SELECT DEPTNO,COUNT(EMPNO)
FROM EMP
GROUP BY DEPTNO
HAVING COUNT(EMPNO) > 4;
```

DEPTNO	COUNT(EMPNO)
20	5
30	6

Comments: Department number 10 has four employees in our sample data set, and has been excluded from the results through the use of the HAVING clause.

The properties that are tested in a HAVING clause must be properties of groups, i.e. one must either test against individual values of the grouped-by item, such as:

HAVING JOB = 'SALESMAN'

OR

JOB = 'ANALYST'

or test against some property of the group, i.e. the number of members in the group (as in the example to exclude departments with less than five employees, or, for instance, tests on aggregate functions of the group - for our data set these

could be properties such as the average or total salaries within the individual groups).

The HAVING clause, when required, always follows immediately after the GROUP BY clause to which it refers. It can contain compound conditions, linked by the boolean operators AND or OR (as above), and parentheses may be used to nest conditions.

Writing queries on more than one table - JOIN

It is not usually very long before a requirement arises to combine information from more than one table, into one coherent query result. For example, using the EMP and DEPT tables, we may wish to display the details of employee numbers and names, alongside the name of the department in which employees work. To do this, we will need to combine information from both the tables, as the employee details are stored in the EMP table, while the department name information is stored in the DEPT table (in the DNAME attribute).

The first point to note, is that this will mean listing both the EMP and DEPT tables in the table-list, following the FROM keyword in the query. In general, the table-list will contain all of the tables required to be accessed during the execution of a query. So far, as our queries have only ever accessed one table, the table-list has contained only one table. To list employee numbers and names with department names, however, the FROM clause will read:

FROM EMP, DEPT

Note that from a purely logical point of view, the order in which the tables are listed after the keyword FROM does not matter at all. In practice, however, if we are dealing with larger tables, the order of tables in the table-list may make a difference to the speed of execution of the query, as it may affect the order in which data from the tables is loaded into main memory from disk. This will be discussed further in the chapter on Advanced SQL.

Listing both the EMP and the DEPT tables after the FROM keyword, however, is not sufficient to achieve the results we are seeking. We don't merely wish for the tables to be accessed in the query; we want the way in which they are accessed to be coordinated in a particular way. We wish to relate the display of a department name with the display of employee numbers and names of employees who work in that department. So we require the query to relate employee records in the EMP table with their corresponding department records in the DEPT table. The way this is achieved in SQL is by the Relational operator JOIN. The JOIN is an absolutely central concept in Relational databases, and therefore in the SQL language. It is such a central concept because this logical combining or relating of data from different tables is a common and important requirement in almost all applications. The ability to relate information from different tables

in a uniform manner has been an important factor in the widespread adoption of Relational database systems.

A curious feature of performing JOINs, or relating information from different tables in a logical way as required in the above query, is that although the process is universally referred to as performing a JOIN, the way it is expressed in SQL does not always involve the use of the word JOIN. This can be particularly confusing for newcomers to JOINs. For example, to satisfy the query above, we would code the WHERE clause as follows:

WHERE EMP.DEPTNO = DEPT.DEPTNO

What this is expressing is that we wish rows in the EMP table to be related to rows in the DEPT table, by matching rows from the two tables whose department numbers (DEPTNOs) are equal. So we are using the DEPTNO column from each employee record in the EMP table, to link that employee record with the department record for that employee in the DEPT table.

The full query would therefore be:

SELECT EMPNO,ENAME,DNAME

FROM EMP,DEPT

WHERE EMP.DEPTNO = DEPT.DEPTNO;

This gives the following results for our test data set:

EMPNO	ENAME	DNAME
7369	SMITH	RESEARCH
7499	ALLEN	SALES
7521	WARD	SALES
7566	JONES	RESEARCH
7654	MARTIN	SALES
7698	BLAKE	SALES
7782	CLARK	ACCOUNTING
7788	SCOTT	RESEARCH
7839	KING	ACCOUNTING
7844	TURNER	SALES
7876	ADAMS	RESEARCH
7900	JAMES	SALES
7902	FORD	RESEARCH
7934	MILLER	ACCOUNTING

A few further points should be noted about the expression of the above query:

- Because we wish to display values of the DNAME attribute in the result, it has, of course, to be included in the select-list.
- We need not include any mention of the DEPTNO attribute in the select-list. We require the EMP.DEPTNO and DEPT.DEPTNO columns to perform the JOIN, so we refer to these columns in the WHERE clause, but we do not wish to display any DEPTNO information, therefore it is not included in the select-list.
- As mentioned above, the order in which the EMP and DEPT tables appear after the FROM keyword is unimportant, at least assuming we can ignore issues of performance response, which we certainly can for tables of this size.
- Similarly, the order in which the columns involved in the JOIN operation are expressed in the WHERE clause is also unimportant.

Example on joining two tables

List the names and jobs of employees, together with the locations in which they work:

SELECT ENAME,JOB,LOC

FROM EMP,DEPT

WHERE EMP.DEPTNO = DEPT.DEPTNO;

ENAME	JOB	LOC
SMITH	CLERK	DALLAS
ALLEN	SALESMAN	CHICAGO
WARD	SALESMAN	CHICAGO
JONES	MANAGER	DALLAS
MARTIN	SALESMAN	CHICAGO
BLAKE	MANAGER	CHICAGO
CLARK	MANAGER	NEW YORK
SCOTT	ANALYST	DALLAS
KING	PRESIDENT	NEW YORK
TURNER	SALESMAN	CHICAGO
ADAMS	CLERK	DALLAS
JAMES	CLERK	CHICAGO
FORD	ANALYST	DALLAS
MILLER	CLERK	NEW YORK

Comments: The exercise requires a simple modification to our first JOIN example – replacing EMPNO and DNAME in the select-list with the JOB and LOC attributes. LOC, like DNAME, is stored in the DEPT table, and so requires the coordination provided by a JOIN, in order to display employee information along with the locations of the departments in which those employees work.

The SQL standard provides the following alternative ways to specify this join:

FROM EMP JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;

FROM EMP JOIN DEPT USING DEPTNO;

FROM EMP NATURAL JOIN DEPT;

In each case, the FROM clause replaces the original FROM and WHERE clauses.

Avoiding ambiguously named columns

DEPTNO has been used as the data item to link records in the EMP and DEPT tables in the above examples. For our EMP and DEPT data set, DEPTNO is in fact the only semantically sensible possibility for use as the JOIN column. In the DEPT table, DEPTNO acts as the primary key (and as such must have a different value in every row within the DEPT table), while in the EMP table, DEPTNO acts as a foreign key, linking each EMP record with the department record in the DEPT table to which the employee record belongs. If we wish to refer to DEPTNO in the select-list, we would need to be careful to specify which instance of DEPTNO we are referring to: the one in the EMP table, or the one in the DEPT table. Failure to do this will lead to an error message indicating that the system is unable to identify which column we are referencing. The way to be specific about which instance of DEPTNO we require is simply to prefix the reference to the DEPTNO column with the table name containing that DEPTNO instance, placing a full stop (.) character between the table name and column name: for example, EMP.DEPTNO, or DEPT.DEPTNO. In this way, the system can identify which instance of DEPTNO is being referenced.

In general, if there is any possible ambiguity about which column is being referenced in a query, because a column with that name appears in more than one table, we use the table prefixing approach to clarify the reference. Note that this was not necessary when referencing any of the columns in the example JOIN queries above, as all of these appeared only once within either the EMP table or the DEPT table.

Outer JOINs

In addition to the basic form of the JOIN, also called a NATURAL JOIN and used to relate rows in different tables, we sometimes require a little more syntax than we have seen so far, in order to obtain all the information we require. Supposing, for example, we wish to list all departments with the employee numbers and names of their employees, plus any departments that do not contain employees.

As a first attempt, we might code:

SELECT DEPT.DEPTNO,DNAME,EMPNO,ENAME

FROM EMP,DEPT

WHERE EMP.DEPTNO = DEPT.DEPTNO

ORDER BY DEPT.DEPTNO;

DEPTNO	DNAME	EMPNO	ENAME
10	ACCOUNTING	7782	CLARK
10	ACCOUNTING	7839	KING
10	ACCOUNTING	7934	MILLER
20	RESEARCH	7369	SMITH
20	RESEARCH	7876	ADAMS
20	RESEARCH	7902	FORD
20	RESEARCH	7788	SCOTT
20	RESEARCH	7566	JONES
30	SALES	7499	ALLEN
30	SALES	7698	BLAKE
30	SALES	7654	MARTIN
30	SALES	7900	JAMES
30	SALES	7844	TURNER
30	SALES	7521	WARD

Comments: Note the use of DEPT.DEPTNO to specify the instance of the JOIN column unambiguously in the select-list. The ORDER BY clause is helpful in sorting the results into DEPT.DEPTNO order. In fact, ordering by DEPTNO, EMPNO would have been even more helpful, particularly in a larger data set. Incidentally, being clear which instance of DEPTNO we are referring to is just as important in the ORDER BY clause as it is in the select-list.

The results of this first attempt are, however, not the complete answer to the original query. Department number 40, called Operations, has no employees currently assigned to it, but it does not appear in the results.

The problem here is that the basic form of the JOIN only extracts matching instances of records from the joined tables. We need something further to force in any record instances that do not match a record in the other table. To do this,

we use a construct called an OUTER JOIN. OUTER JOINs are used precisely in situations where we wish to force into our results set, rows that do and do not match a usual JOIN condition. There are three types of OUTER JOIN: LEFT, RIGHT, and FULL OUTER JOINS. To demonstrate the OUTER JOINs, we will use the following tables.

Person table

ID	NAME	ADDRESS	CAR
100007	John Smith	Pretoria	ABC 56789
200008	Jabulani Dube	Cape Town	ACD 5900
704555	Ruth White	Cape Town	NULL

The person table holds the information of people. The ID is the primary key. A person can own a car or not.

Car table

REG	MODEL	OWNER
ABC 56789	KIA	100007
ACD 5900	Nissan xTrail	200008
AZA 97	BMW x3	NULL

The car table holds information of cars. The REG is the primary key. A car can have an owner or not.

LEFT OUTER JOIN

The syntax of the LEFT OUTER JOIN involves including the LEFT JOIN keyword in the query. Here's an example: List all persons together with their car's registration and model, including any person without any car. The requirement is to force into the result set any person that does not have a car. To satisfy the requirement, we would write our query as such:

SELECT ID,NAME,REG,MODEL

FROM Person LEFT JOIN car ON Person.ID = Car.OWNER;

ID	NAME	REG	MODEL
100007	John Smith	ABC 56789	KIA
200008	Jabulani Dubee	ACD 5900	Nissan xTrail
704555	Ruth White	NULL	NULL

Comment: The query returns all the persons with cars, plus the one instance of a person (ID 704555) having no car.

RIGHT OUTER JOIN

Like a LEFT JOIN, the syntax of the RIGHT OUTER JOIN involves including the RIGHT JOIN keyword in the query. An example would be: List all cars together with their owner's identification and name, including any car not owned by anyone.

SELECT REG,MODEL,ID,NAME

FROM Person RIGHT JOIN car ON Person.ID = Car.OWNER;

REG	MODEL	ID	NAME
ABC 56789	KIA	100007	John Smith
ACD 5900	Nissan xTrail	200008	Jabulani Dubee
AZA 97	BMW x3	NULL	NULL

Comment: The query returns all the cars that are owned, plus the one instance of a car not owned by anyone.

FULL OUTER JOIN

If you wish to show both person records of those that don't own any car and car records that don't have any owner, then you need to use the FULL OUTER JOIN:

SELECT REG,MODEL,ID,NAME

FROM Person FULL JOIN car ON Person.ID = Car.OWNER;

REG	MODEL	ID	NAME
ABC 56789	KIA	100007	John Smith
ACD 5900	Nissan xTrail	200008	Jabulani Dubee
AZA 97	BMW x3	NULL	NULL
NULL	NULL	704555	Ruth White

Using table aliases

Table aliasing involves specifying aliases, or alternative names, that can be used to refer to the table during the processing of a query. The table aliases are specified in the table-list, following the FROM keyword. For example, the above FULL OUTER JOIN query can be written using aliases:

SELECT REG,MODEL,ID,NAME

FROM Person p FULL JOIN car c ON p.ID = c.OWNER;

SELF JOINS

Sometimes it is necessary to JOIN a table to itself in order to compare records from the same table. An example of this might be if we wish to compare values of salary on an individual basis between employees.

Example: find all employees who are paid more than "JONES"

What is required here is to compare the salaries of employees with the salary paid to JONES. A way of doing this, involves JOINing the EMP table with itself, so that we can carry out salary comparisons in the WHERE clause of an SQL query. However, if we wish to JOIN a table to itself, we need a mechanism for referring to the different rows being compared.

In order to specify the query to find out which employees are paid more than JONES, we shall use two table aliases, X and Y for the EMP table. We shall use X to denote employees whom we are comparing with JONES, and Y to denote JONES' record specifically. This leads to the following query specification:

SELECT X.EMPNO,X.ENAME,X.SAL,Y.EMPNO,Y.ENAME,Y.SAL

FROM EMP X,EMP Y

WHERE X.SAL > Y.SAL

AND Y.ENAME = 'JONES'

EMPNO	ENAME	SAL	EMPNO	ENAME	SAL
7788	SCOTT	3000	7566	JONES	2975
7839	KING	5000	7566	JONES	2975
7902	FORD	3000	7566	JONES	2975

Comments: Note the use of the aliases for each of the column specifications in the select-list. We ensure that the alias Y is associated with the employee JONES through the second condition in the WHERE clause, "AND Y.ENAME = 'JONES'". The first condition in the WHERE clause, comparing salaries, ensures that apart from JONES' record, which is listed in the result as a check on the query results, only the details of employees who are paid more than JONES are retrieved.

Summary of JOINs

We have seen three forms of the JOIN condition. The basic JOIN, also called a NATURAL JOIN, is used to combine or coordinate the results of a query in a logical way across more than one table. In our examples, we have seen that JOINing two tables together involves one JOIN condition and, in general, JOINing N tables together requires the specification of N-1 JOIN conditions. A lot of work has gone into the development of efficient algorithms for the execution of JOINs in all the major database systems, with the result being that the overall performance of Relational database systems has seen a very considerable improvement since their introduction in the early '80s. In spite of this, JOINs are still an expensive operation in terms of query processing, and there can be situations where we seek ways of reducing the number of JOINs required to perform specific transactions.

Two further variants we have seen on the basic JOIN operation are the OUTER JOIN and the SELF JOIN. The OUTER JOIN is used to force non-matching records from one side of a JOIN into the set of retrieved results. The SELF JOIN is used where it is required to compare rows in a table with other rows from the same table. This comparison is facilitated through the use of aliases, alternative names which are associated with the table, and so can be used to reference the table on different sides of a JOIN specification.

Nested queries

The power of the SQL language is increased considerably through the ability to include one query within another. This is known as nesting queries, or writing sub-queries.

Nested query example

Find all employees who are paid more than JONES:

This might be considered a two-stage task:

1. Find Jones' salary.
2. Find all those employees who are paid more than the salary found in step 1.

We might code step 1 as follows:

SELECT SAL

FROM EMP

WHERE ENAME = 'JONES'

The nested query mechanism allows us to enclose this query within another one, which we might use to perform step 2:

SELECT EMPNO,ENAME,SAL

FROM EMP

WHERE SAL > …..

We simply need the syntax to enclose the query to implement step 1 in such a way that is provides its result to the query which implements step 2.

This is done by enclosing the query for step 1 in parentheses, and linking it to the query for step 2 as follows:

SELECT EMPNO,ENAME,SAL

FROM EMP

WHERE SAL >

(SELECT SAL FROM EMP WHERE ENAME = 'JONES');

This gives the following results:

EMPNO	ENAME	SAL
7788	SCOTT	3000
7839	KING	5000
7902	FORD	3000

These are indeed the employees who earn more than JONES (who earns 2975).

Whenever a query appears to fall into a succession of natural steps such as the one above, it is a likely candidate to be coded as a nested query.

An important point has to be kept in mind when testing for equality of values across inner and outer queries.

If the inner query returns just one value, then we can use the equal sign, e.g.

SELECT …. FROM ….

WHERE ATTRIBUTE 1 = (SELECT ….

FROM …..)

If, however, the inner query might return more than one row, we must use the keyword IN, so that we can check whether the value of the attribute being tested in the WHERE clause of the outer query is IN the set of values returned by the inner query. Sub-queries can be included linked to a HAVING clause, i.e. they can retrieve a result which forms part of the condition in the evaluation of a HAVING clause. In this situation the format of the HAVING clause is:

HAVING ….

(SELECT … FROM .. WHERE ….)

The inner query may of course itself have inner queries, with WHERE, GROUP BY and HAVING clauses.

The depth and breadth of nested queries

The number of queries that can be nested varies from one database system to another, but there is support for this SQL construct in all the leading databases such that there is no practical limit to the number of queries that can be nested.

In a similar way, a number of queries can be included at the same level of nesting, their results being combined together using the AND or OR keywords, according to the following syntax:

SELECT …. FROM ….

WHERE CONDITION 1 (SELECT ….

FROM ….. WHERE …..)

AND/OR (SELECT ….. FROM …. WHERE ….)

AND/OR

……

The UNION operator

To find the details of any employees receiving the same salaries as either SCOTT or WARD, we could code:

SELECT EMPNO,ENAME,SAL

FROM EMP

WHERE SAL IN

(SELECT SAL FROM EMP

WHERE ENAME = ‘WARD’

OR

ENAME = ‘SCOTT’);

But suppose SCOTT and WARD are in different tables. If this is the case, we need to use the UNION operator in order to combine the results of queries on two different tables as follows:

SELECT EMPNO,ENAME,SAL

```
FROM EMP
WHERE SAL IN
(SELECT SAL
FROM EMP1
WHERE ENAME = 'WARD'
UNION
SELECT SAL
FROM EMP2
WHERE ENAME = 'SCOTT');
```

Comments: We are assuming here that WARD is in a table called EMP1, and SCOTT in EMP2. The two salary values retrieved from these sub-queries are combined into a single results set, which is retrieved for comparison with all salary values in the EMP table in the outer query. Because there is more than one salary returned from the combined inner query, the IN keyword is used to make the comparison. Note that as with the Relational Algebra equivalent, the results of the SQL UNION operator must be union compatible, as we see they are in this case, as they both return single salary columns.

The INTERSECT operator

Again, like its Relational Algebra equivalent, the SQL INTERSECT operator can be used to extract the rows in common between two sets of query results:

```
SELECT JOB
FROM EMP
WHERE SAL > 2000 INTERSECT
SELECT JOB
FROM SHOPFLOORDETAILS;
```

Here the INTERSECT operator is used to find all jobs in common between the two queries. The first query returns all jobs that are paid more than 2000, whereas the second returns all jobs from a separate table called SHOPFLOORDETAILS. The final result, therefore, will be a list of all jobs in the SHOPFLOORDETAILS table that are paid more than 2000. Again, note that the sets of results compared with one another using the INTERSECT operator must be union compatible.

The MINUS operator

MINUS is used, like the DIFFERENCE operator of Relational Algebra, to subtract one set of results from another, where those results are derived from different tables.

For example:

```
SELECT EMPNO,ENAME,SAL
FROM EMP
WHERE ENAME IN
(SELECT ENAME
FROM EMP1
MINUS
SELECT ENAME
FROM EMP2);
```

Comments: The result of this query lists the details for employees whose names are the same as employees in table EMP1, with the exception of any names that are the same as employees in table EMP2.

ANY or ALL operator

The ANY or ALL operators may be used for sub-queries that return more than one row. They are used on the WHERE or HAVING clause in conjunction with the logical operators (=, !=, >, >=, <=, <). ANY compares a value to each value returned by a sub-query.

To display employees who earn more than the lowest salary in Department 30, enter:

```
SELECT ENAME, SAL, JOB, DEPTNO
FROM EMP
WHERE SAL >>
ANY
(SELECT DISTINCT SAL
FROM EMP
WHERE DEPTNO = 30)
ORDER BY SAL DESC;
```

ENAME	SAL	JOB	DEPTNO
KING	5000	PRESIDENT	10
SCOTT	3000	ANALYST	20
FORD	3000	ANALYST	20
JONES	2975	MANAGER	20
BLAKE	2850	MANAGER	30
CLARK	2450	MANAGER	10
ALLEN	1600	SALESMAN	30
TURNER	1500	SALESMAN	30
MILLER	1300	CLERK	10
WARD	1250	SALESMAN	30
MARTIN	1250	SALESMAN	30
ADAMS	1100	CLERK	20

Comments: Note the use of the double >> sign, which is the syntax used in conjunction with the ANY and ALL operators to denote the fact that the comparison is carried out repeatedly during query execution. "= ANY" is equivalent to the keyword IN. With ANY, the DISTINCT keyword is often used in the sub-query to avoid the same values being selected several times. Clearly the lowest salary in department 30 is below 1100.

ALL compares a value to every value returned by a sub-query.

The following query finds employees who earn more than every employee in Department 30:

```
SELECT ENAME, SAL, JOB, DEPTNO
FROM EMP
WHERE SAL >>ALL
(SELECT DISTINCT SAL
FROM EMP
WHERE DEPTNO = 30)
```

ORDER BY SAL DESC;

ENAME	SAL	JOB	DEPTNO
KING	5000	PRESIDENT	10
SCOTT	3000	ANALYST	20
FORD	3000	ANALYST	20
JONES	2975	MANAGER	20

Comments: The inner query retrieves the salaries for Department 30. The outer query, using the All keyword, ensures that the salaries retrieved are higher than all of those in department 30. Clearly the highest salary in department 30 is below 2975.

Note that the NOT operator can be used with IN, ANY or ALL.

Correlated sub-queries

A correlated sub-query is a nested sub-query that is executed once for each 'candidate row' considered by the main query, and which on execution uses a value from a column in the outer query. This causes the correlated sub-query to be processed in a different way from the ordinary nested sub-query.

With a normal nested sub-query, the inner select runs first and it executes once, returning values to be used by the main query. A correlated sub-query, on the other hand, executes once for each candidate row to be considered by the outer query. The inner query is driven by the outer query.

Steps to execute a correlated sub-query:

1. The outer query fetches a candidate row.
2. The inner query is executed, using the value from the candidate row fetched by the outer query.
3. Whether the candidate row is retained depends on the values returned by the execution of the inner query.
4. Repeat until no candidate row remains.

Example

We can use a correlated sub-query to find employees who earn a salary greater than the average salary for their department:

SELECT EMPNO,ENAME,SAL,DEPTNO

FROM EMP E

WHERE SAL >> (SELECT AVG(SAL)

FROM EMP

WHERE DEPTNO = E.DEPTNO)

ORDER BY DEPTNO;

Giving the results:

EMPNO	ENAME	SAL	DEPTNO
7839	KING	5000	10
7566	JONES	2975	20
7788	SCOTT	3000	20
7902	FORD	3000	20
7499	ALLEN	1600	30
7698	BLAKE	2850	30

Comments: We can see immediately that this is a correlated sub-query since we have used a column from the outer select in the WHERE clause of the inner select. Note that the alias is necessary only to avoid ambiguity in column names.

Interactive queries

A very useful facility is provided to enable users to run the same query again, entering a different value of a parameter to a WHERE or HAVING clause. This is done by prefixing the column specification for which different values are to be supplied by the "&" sign.

Example

Find the number of clerks based in department 10. Find the number of clerks in other departments by running the same query, in each case entering the value of the department number interactively.

SELECT COUNT(EMPNO) "NUMBER OF CLERKS"

FROM EMP

WHERE JOB = ‘CLERK’

AND DEPTNO = &DEPTNO

The user will be asked to enter a value for DEPTNO. The result for entering 10 is:

NUMBER OF CLERKS
1

This syntax provides a limited amount of interactivity with SQL queries, which can avoid the need to recode in order to vary the values of interactively specified parameters.

Activities

The following individual activities will provide practice by focusing on specific SQL constructs in each activity. These will be supplemented by the succeeding review questions, which will draw on all of the SQL material covered in this and the introductory chapter to SQL. This first activity will concentrate on various types of SQL JOIN.

Activity 1: JOINs

1. Find all employees located in Dallas.
2. List the total annual pay for the Sales department (remember salary and commission data are provided as monthly figures).
3. List any departments that do not contain any employees.
4. Which workers earn more than their managers (hint: remember that the MGR attribute stores the EMPNO of an employee’s manager).

Activity 2: GROUP BY

1. List the total monthly pay for each department.
2. List the number of employees located in Chicago and New York.
3. Find all jobs with more than two employees.

Activity 3: Nested queries

1. List the details of the highest-paid employee.
2. Find whether anyone in department 30 has the same job as JONES.
3. Find the job with the most employees.

Review questions

1. Why is the JOIN operation such a core concept in Relational database systems? Describe how JOINs are expressed in SQL.
2. How can we express in SQL where it is required to JOIN more than two tables together?
3. Differentiate between the terms SELF JOIN and OUTER JOIN, and give a practical example of the use of each (you need not necessarily restrict yourself to the use of the data tables used in earlier examples).
4. Describe the use of the GROUP BY clause for categorising data in SQL.
5. What restrictions exist on the contents of a select-list which appears in the same query as a GROUP BY clause?
6. It is sometimes said that the HAVING keyword relates to the GROUP BY clause, in the same way that the WHERE keyword relates to SELECT. Explain the meaning of this statement, and draw parallels between the SELECT….WHERE and the GROUP BY …. HAVING constructs in SQL.
7. Describe the use of nested queries within the SQL language.

Discussion topic

JOINs versus nested queries

In general, Relational database systems are optimised to perform JOIN operations very efficiently. It is also true that many SQL queries can be expressed either as a JOIN or as a nested query. Consider for yourself, and discuss online with colleagues, which of these two constructs you find easier to understand and to code. Do you find that any previous programming experience you may have had influences your ease of understanding and application of these concepts? For example, most people who have experience of conventional programming languages are familiar with loop statements and nesting one loop inside another, a construct which is very similar to a nested query. In general, do you think having had previous experience is an advantage or disadvantage when learning a language such as SQL?

Additional content

Following on from the Additional Content section of the introductory chapter to SQL, you are encouraged to explore further the SQL support provided within Microsoft Access, or some other database of your choice, for the SQL constructs we have covered in this chapter.

Whereas you will have found relatively consistent support for all of the SQL features covered in the introductory chapter, now that we have covered the majority of the constructs available within the DML part of SQL, you are much more likely to find variations in support for the different features. These variations are likely to include:

- Complete lack of support for some of the constructs covered, e.g. some databases do not allow nested queries at all, or do not support JOINs.
- Partial support for some constructs; for example, some systems support nested queries, but do not support the keywords ANY and ALL.
- Variations in the limits to how different constructs can be used; for example, some databases only allow query nesting to two or three levels, or support conventional JOINs but not the SELF or OUTER JOIN.

Using the sample tables provided in the database you have chosen, investigate the use of the SQL constructs described in this chapter, noting down differences and limitations in their implementation between your chosen database and the Oracle implementation.

Chapter 5. Advanced SQL

Table of contents

Objectives

At the end of this chapter you should be able to:

- Create, alter and drop tables in SQL.
- Insert, update and delete rows from SQL tables.
- Create, alter and remove views based on SQL tables, and describe some of the strengths and limitations of views.

Introduction

In parallel with this chapter, you should read Chapter 6 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter introduces further features of the SQL language, and seeks to integrate the material of all three chapters which have provided coverage of SQL in this module. The chapter introduces the means by which tables are created, changed and removed in SQL. The statements for inserting, updating and deleting rows from tables are also covered. Views are an important feature in SQL for tailoring the presentation of data, and acting as a security mechanism. The statements for creating and using views will be described, along with some of the inherent limitations of the view mechanism.

Context

This chapter is the final one specifically dedicated to the SQL language, and so it forms an important role in drawing together the information covered in all three of the SQL-related chapters of the module. SQL continues to be an important vehicle for explaining and illustrating concepts in many of the later chapters of the module, and provides a medium through which many relevant practical exercises can be performed.

Although this chapter is called Advanced SQL, the material covered is not in general more difficult than that of previous chapters. The previous two chapters on SQL have provided a fairly comprehensive coverage of the data manipulation (DML) part of the language, enabling the specification of a wide range of queries. This chapter introduces the mechanisms for creating, changing and removing tables, and for inserting, updating and removing rows from tables. The mechanisms for performing these actions in SQL are relatively straightforward, but are extremely powerful. Because SQL is a command-level language, these commands do not include the checks that we have grown to expect from a typical Graphical User Interface (GUI), and so they must be used with care.

Creating tables in SQL

Data definition language (DDL) statements are used for creating, modifying and removing data objects. They affect both physical and logical data structures. Their syntax is generally much more varied than the data manipulation language (DML) statements we have covered in the previous two chapters.

The CREATE statement in SQL can be used to bring into being a range of different data objects, including the following:

- Data tables

- Views on existing tables
- Indexes (data structures which speed up access to data)
- Database user accounts

In this section we shall concentrate on the use of the CREATE statement for establishing new tables. The CREATE TABLE statement has a range of different options, and so we shall start by showing a simplified version of the syntax as follows:

CREATE TABLE "TABLE NAME" (COLUMN SPECIFICATION 1, …… COLUMN SPECIFICATION n);

Where column specification includes:

- A column name
- The data type of the column
- Where appropriate, a specification of the length of the column
- An optional indicator of whether or not the column is to contain null values

Data objects are subject to a number of restrictions, and these will vary between different database systems. We shall describe the restrictions on naming tables and columns in Oracle, as they are fairly typical limitations encountered in databases generally, the main exceptions being older PC-based database environments.

- Table names must start with an alphabetic character, and can contain up to 30 characters.
- Table names can contain the letters A-Z, the numbers 0-9, and the characters – and _.
- Table names must be unique within any specific user account.
- Column names must start with a character, and may comprise up to 30 characters.
- They can contain the same characters as table names.
- Column names must be unique within a table, but you can specify the same column names in different tables.
- In Oracle there can be up to 254 columns in a table.

Referring to the simplified version of the CREATE TABLE statement above, the column specifications are contained in parentheses.

Data types

We shall focus on three specific data types for use in our practical work, as their equivalents (though they may be differently named) can be found in almost any database environment. These data types appear in Oracle as the following:

1. **VARCHAR2:** is used to store variable-length character strings. In Oracle the strings can store up to 2000 characters. The syntax for specifying a data item of type VARCHAR2 is:

VARCHAR2 (length)

where length is the maximum length of the character string to be stored.

Note: Some DBMSs, including MySQL, uses VARCHAR instead.

2. **NUMBER:** is used to store general numbers. The NUMBER data type offers the greatest flexibility for storing numeric data. It accepts positive and negative integers and real numbers, and has from 1 to 38 digits of precision. The syntax for specifying the NUMBER data type is:

NUMBER (precision, scale)

where precision is the maximum number of digits to be stored and scale indicates number of digits to the right of the decimal point. If scale is omitted, then integer (whole) numbers are stored.

Note: Some DBMSs, including MySQL, expect you to use exact data types for numeric data. For example, if you want to hold integers, then you must use the INT datatype. If you wish to hold decimal numbers, then you must use the DOUBLE datatype.

3. **DATE:** is used to specify an attribute is of the type 'date'. The format in which dates are represented within attributes of type date is: dd-mon-yyyy; for example, 10-jan-2000.The syntax to specify an attribute is of type date is simply to specify the word DATE after the name of the attribute in the CREATE TABLE statement.

Like many other systems, Oracle contains a number of other data types in addition to the most commonly found ones just described. As an example of other data types that may be provided, here are a few of the rather more Oracle-specific data types available:

- Decimal: is used to store fixed-point numbers, and provides compatibility with IBM's DB2 and SQL/DS systems.
- Float: is used to store floating-point numbers, and provides compatibility with the ANSI float datatype.
- Char: is used to store fixed-length character strings. It is most commonly used for representing attributes of one character in length.

- Long: is used to store a little over two gigabytes of data. However, none of Oracle's built-in functions and operators can be used to search an attribute of type 'long'.

The final part of a column specification allows us to specify whether or not the column is to contain null values, i.e. whether or not it is a mandatory column. The syntax is simply to specify either "Not null" or Null after the data type (and possible length) specification.

Example CREATE TABLE statement

Suppose we wished to create a table called MUSIC_COLLECTION, which we would use to store the details of CDs, cassettes, minidiscs, etc. This could be done with the following statement:

CREATE TABLE MUSIC_COLLECTION (ITEM_ID NUMBER(4),

TITLE VARCHAR2(40),

ARTIST VARCHAR2(30),

ITEM_TYPE VARCHAR2(1),

DATE_PURCHASED DATE);

We use a unique numeric identifier called ITEM_ID to identify each of the items in the collection, as we cannot rely on either the TITLE or ARTIST attributes to identify items uniquely. The ITEM_TYPE attribute is used to identify which format the item is in, i.e. cassette, CD, etc.

Defining primary keys

Remember that a primary key is used to identify uniquely each instance of an entity. For the MUSIC_COLLECTION table, the primary key of ITEM_ID will identify uniquely each of the items in the music collection. The CREATE TABLE statement provides the syntax to define primary keys as follows:

CREATE TABLE "TABLE NAME"

(COLUMN SPECIFICATION 1,

COLUMN SPECIFICATION n,

PRIMARY KEY (columnA, …., columnX));

where columns columnA,….,columnX are the columns to be included in the primary key, separated from each other by commas, and all of the columns included in the primary key being enclosed in parentheses. In SQL the definition of a primary key on a CREATE TABLE statement is optional, but in practice this is virtually always worth doing. It will help maintain the integrity of the database. Oracle will ensure all the values of a primary key are different, and

will not allow a null value to be entered for a primary key. In Oracle there is an upper limit of 16 columns that can be included within a primary key.

Example of creating table with a primary key

If we wanted to specify that ITEM_ID is to be used as the primary key in the MUSIC-COLLECTION table, we could code the following version of our CREATE TABLE statement:

CREATE TABLE MUSIC_COLLECTION (ITEM_ID NUMBER(4),

TITLE VARCHAR2(40),

ARTIST VARCHAR2(30), ITEM_TYPE VARCHAR2(1),

DATE_PURCHASED DATE, PRIMARY KEY (ITEM_ID));

Defining foreign keys

A foreign key is used to form the link between rows stored in one table and corresponding rows in another table. For example, in the sample data set we have used in the previous two chapters on SQL, the foreign key EMP.DEPTNO in the employee table was used to link employees to their corresponding departments in the DEPT table. The CREATE TABLE statement allows us to specify one or more foreign keys in a table as follows:

CREATE TABLE "TABLE NAME"

(COLUMN SPECIFICATION 1,

COLUMN SPECIFICATION n,

PRIMARY KEY (columnA, ..., columnX), CONSTRAINT "constraint name"

FOREIGN KEY (columnAA, ..., columnXX) REFERENCES "primary key specification")

.....);

As for primary keys, foreign key specifications are not mandatory in a CREATE TABLE statement. But again, specifying foreign keys is desirable in maintaining the integrity of the database. Oracle will ensure that a value entered for a foreign key must either equal a value of the corresponding primary key, or be null. CREATE TABLE statements can contain both a primary key specification, and a number of foreign key specifications.

An explanation of the foreign key specification is as follows:

- The first item is the keyword "CONSTRAINT", followed by an optional constraint name. Although specifying a constraint name is optional, it is recommended that you always include it. The reason for this is that if no constraint name is specified, most systems, including Oracle, will

allocate one, and this will not be in any way easy to remember. If you later wish to refer to the foreign key constraint - for instance, because you wish to remove it - then providing your own name at the point you enter the CREATE TABLE statement will make this much easier.

- The words FOREIGN KEY are followed by a list of the columns to be included in the foreign key, contained in parentheses and separated by commas (this list of column names is in general different from the list of column names in the "Primary key" clause).
- REFERENCES is the mandatory keyword, indicating that the foreign key will refer to a primary key.
- The primary key specification starts with the name of the table containing the referenced primary key, and then lists the columns comprising the primary key, contained in parentheses and separated by commas as usual.

The full stops (……) shown in the version of the syntax above indicate that there may be more than one foreign key specification.

Example of defining a foreign key in SQL

Supposing we have a second table, which we use to keep track of recording artists whose recordings we buy. The ARTIST table could be created with the following statement:

CREATE TABLE ARTIST (

ARTIST_ID NUMBER(2),

ARTIST_NAME VARCHAR2(30),

COUNTRY_OF_ORIGIN VARCHAR2(25),

DATE_OF_BIRTH DATE, PRIMARY KEY (ARTIST_ID));

To relate the MUSIC_COLLECTION table to the ARTIST table, we could make the following modifications to the CREATE TABLE statement for the MUSIC_COLLECTION TABLE, which replaces the ARTIST_NAME with a foreign key reference to the ARTIST-ID:

CREATE TABLE MUSIC_COLLECTION (ITEM_ID NUMBER(4),

TITLE VARCHAR(40),

ARTIST_ID NUMBER(2),

ITEM_TYPE VARCHAR2(1),

DATE_PURCHASED DATE, PRIMARY KEY (ITEM_ID),

CONSTRAINT FK_ARTIST

FOREIGN KEY (ARTIST_ID) REFERENCES ARTIST (ARTIST_ID));

The following points should be noted:

1. We have modified the attribute specified on line four, from containing the Artist name, and being of type Varchar2 and length 30, to be the ARTIST_ID, of type number and length2.
2. We have then used the ARTIST_ID as the foreign key, which references the primary key of the table ARTIST.

Note that in general, it would not be possible to make changes such as these to existing tables. It would require some existing tables and data to be deleted. Therefore it is good practice to consider very carefully the design of tables and specification of the primary and foreign keys that are going to be required, and to specify this correctly the first time in CREATE TABLE statements. It is however possible to add and remove both primary key and foreign key constraints, and this will be covered, along with the details of a range of other constraints mechanisms, in the chapter Declarative Constraints and Database Triggers.

Copying data by combining CREATE TABLE and SELECT

An extremely useful variant of the CREATE TABLE statement exists for copying data. Essentially, this consists of using a SELECT statement to provide the column specifications for the table to be created and, in addition, the data that is retrieved by the SELECT statement is copied into the new table structure. The syntax for this form of the statement is as follows:

CREATE TABLE “TABLE NAME”

AS “select statement”;

where “select statement” can be any valid SQL query.

This form of the CREATE TABLE statement can be used to, for example:

- Copy entire tables.
- Copy subsets of tables using the select-list and WHERE clause to filter rows.
- Create tables which combine data from more than one table (using JOINs).
- Create tables containing aggregated data (using GROUP BY).

Examples of copying data using CREATE TABLE.....SELECT

Example 1:

To create a copy of the EMP table we have used in previous exercises:

CREATE TABLE EMPCOPY

AS SELECT * FROM EMP;

Example 2:

To create a table containing a list of employees and their locations we can code:

```
CREATE TABLE EMPLOC
AS SELECT EMPNO, EMP.DEPTNO, ENAME, LOC
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO;
```

To examine the contents of the new table:

```
SELECT *
FROM EMPLOC;
```

EMPNO	DEPTNO	ENAME	LOC
7369	20	SMITH	DALLAS
7499	30	ALLEN	CHICAGO
7521	30	WARD	CHICAGO
7566	20	JONES	DALLAS
7654	30	MARTIN	CHICAGO
7698	30	BLAKE	CHICAGO
7782	10	CLARK	NEW YORK
7788	20	SCOTT	DALLAS
7839	10	KING	NEW YORK
7844	30	TURNER	CHICAGO
7876	20	ADAMS	DALLAS
7900	30	JAMES	CHICAGO
7902	20	FORD	DALLAS
7934	10	MILLER	NEW YORK

Copying table structures without data

Sometimes you may wish to copy the structure of a table without moving any of the data from the old table into the new one. For example, to take a copy of the structure of the EMP table, but without copying any employee records into the new table, we could use:

```
CREATE TABLE EMPSTRUCT
AS SELECT *
FROM EMP
WHERE 1 = 2;
```

To verify we have copied the structure:

```
DESCRIBE EMPSTRUCT
```

NAME	NULL?	TYPE
EMPNO		NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
DEPTNO		NUMBER(2)

To verify the new structure contains no data:

```
SELECT *
FROM EMPSTRUCT;
```

no rows selected

This is an example of the way the SQL language can be made to fit a particular purpose. We wish in this example to copy the structure of a table, but ensure no rows are selected from it. By supplying a WHERE clause which contains a condition, namely WHERE 1 = 2, that can never be satisfied, we ensure that no rows are copied along with the structure.

The ALTER TABLE statement

The ALTER statement in SQL, like the CREATE statement, can be used to change a number of different types of data objects, including tables, access privileges and constraints. Here we shall concentrate on its use to change the structure of tables.

You can use the ALTER TABLE statement to modify a table's definition. This statement changes the structure of a table, not its contents. You can use the ALTER TABLE statement to:

- Add a new column to an existing table.
- Increase or decrease the width of an existing column.
- Change an existing column from mandatory to optional (i.e. specify that it may contain nulls).

Using ALTER TABLE to add columns

Columns can be added to existing tables with this form of the ALTER TABLE statement. The syntax is:

ALTER TABLE "TABLE NAME"

ADD "COLUMN SPECIFICATION 1",

……,

"COLUMN SPECIFICATION n";

For example, to add a department-head attribute to the DEPT table, we could specify:

ALTER TABLE DEPT

ADD DEPT_HEAD NUMBER(4);

We could imagine that the new DEPT_HEAD column would contain EMPNO values, corresponding to the employees who were the department heads of particular departments. Incidentally, if we had wished to make the DEPT_HEAD field mandatory, we could not have done so, as the ALTER TABLE statement does not enable the addition of mandatory fields to tables that already contain data.

We can add a number of columns with one ALTER TABLE statement.

Modifying columns with ALTER TABLE

This form of the ALTER TABLE statement permits changes to be made to existing column definitions. The format is:

ALTER TABLE “TABLE NAME”

MODIFY “COLUMN SPECIFICATION 1”,

……,

COLUMN SPECIFICATION n“;

For example, to change our copy of the EMP table, called EMPCOPY, so that the DEPTNO attribute can contain three digit values:

ALTER TABLE EMPCOPY

MODIFY DEPTNO NUMBER(3);

This form of the ALTER TABLE statement can be used to:

- Increase the length of an existing column.
- Transform a column from mandatory to optional (i.e. specify it can contain nulls).

There are a number of restrictions in the use of the ALTER TABLE statement for modifying columns, most of which might be guessed through a careful consideration of what is being required of the system. For example, you cannot:

- Reduce the size of an existing column (even if it has no data in it).
- Change a column from being optional to mandatory.

Removing tables using the DROP TABLE statement

To remove a table, the DDL statement is:

DROP TABLE “TABLE NAME”;

It is deceptively easy to issue this command, and unlike most systems one encounters today, there is no prompt at all about whether you wish to proceed with the process. Dropping a table involves the removal of all the data and constraints on the table and, finally, removal of the table structure itself.

Example to remove our copy of the EMP table, called EMPCOPY:

DROP TABLE EMPCOPY;

Table dropped.

Using DROP TABLE when creating tables

Sometimes we wish to recreate an existing table, perhaps because we wish to add new constraints to it, or to carry out changes that are not easy to perform using the ALTER TABLE or other DDL statements. If this is the case, it will be

necessary to drop the table before issuing the new CREATE TABLE statement. Clearly this should only be done if the data in the table can be lost, or can be safely copied elsewhere, perhaps through the use of a CREATE TABLE with a SELECT clause.

For a little further information about the use of the DROP TABLE statement when creating tables, see the section on using SQL scripts later in this chapter.

Adding new rows to table with INSERT

The INSERT statement is used to add rows to an existing table. The statement has two basic forms:

1. To insert a single row into a table:

INSERT INTO “TABLE NAME” (COLUMN-LIST) VALUES

(LIST OF VALUES TO BE INSERTED);

The COLUMN-LIST describes all of the columns into which data is to be inserted. If values are to be inserted for every column, i.e. an entire row is to be added, then the COLUMN-LIST can be omitted.

The LIST OF VALUES TO BE INSERTED comprises the separate values of the new data items, separated by commas.

Example 1:

To insert a new row into the table DEPTCOPY (this is a copy of the DEPT table):

INSERT INTO DEPTCOPY VALUES (50,‘PURCHASING’,‘SAN FRANCISCO’);

1 row created.

Example 2:

To insert a new department for which we do not yet know the location:

INSERT INTO DEPTCOPY (DEPTNO,DNAME)

VALUES (60,‘PRODUCTION’);

1 row created.

2. To insert a number of rows using a SELECT statement.

The syntax for this form of the INSERT statement is as follows:

INSERT INTO “TABLE NAME” (COLUMN-LIST)

“SELECT STATEMENT”;

The COLUMN-LIST is optional, and is used to specify which columns are to be filled when not all the columns in the rows of the target table are to be filled.

The "SELECT STATEMENT" is any valid select statement.

This is, rather like the case of using SELECT with the CREATE TABLE statement, a very powerful way of moving existing data (possibly from separate tables) into a new table.

Example:

Supposing we have created a table called MANAGER, which is currently empty. To insert the numbers, names and salaries of all the employees who are managers into the table we would code:

INSERT INTO MANAGER

SELECT EMPNO, ENAME, SAL

FROM EMP

WHERE JOB = 'MANAGER';

3 rows created.

To verify the employees in the table are managers, we can select the data and compare the jobs of those employees in the original EMP table:

SELECT *

FROM MANAGER;

EMPNO	ENAME	SAL
7566	JONES	2975
7698	BLAKE	2850
7782	CLARK	2450

Changing column values with UPDATE

The UPDATE statement is used to change the values of columns in SQL tables. It is extremely powerful, but like the DROP statement we encountered earlier, it does not prompt you about whether you really wish to make the changes you have specified, and so it must be used with care.

The syntax of the UPDATE statement is as follows:

UPDATE “TABLE NAME” SET “column-list” = expression | sub-query WHERE “CONDITION”;

The SET keyword immediately precedes the column or columns to be updated, which are specified in the column list. If there is more than one column in the list, they are separated by commas.

Following the equals sign “=” there are two possibilities for the format of the value to be assigned. An expression can be used, which may include mathematical operations on table columns as well as constant values. If an expression is supplying the update value, then only one column can be updated.

Alternatively, a sub-query or SELECT statement can be used to return the value or values to which the updated columns will be set. If a sub-query is used to return the updated values, then the number of columns to be updated must be the same as the number of columns in the select-list of the sub-query.

Finally, the syntax includes a WHERE clause, which is used to specify which rows in the target table will be updated. If this WHERE clause is omitted, all rows in the table will be updated.

Example 1:

To give all the analysts in the copy of the EMP table (called EMPCOPY) a raise of 10%:

UPDATE EMPCOPY

SET SAL = SAL * 1.1

WHERE JOB = ‘ANALYST’;

2 rows updated.

Example 2:

Suppose we wish to flatten the management structure for the employees stored in the EMPCOPY table. Recall that the MGR of each employee contains the employee number of their manager. We might implement this flattening exercise, at least as far as the database systems are concerned, by setting all employees’ MGR fields to that of KING, who is the president of the company. The update statement to do this would be as follows:

UPDATE EMPCOPY

SET MGR =

(SELECT EMPNO

FROM EMP

WHERE ENAME = ‘KING’) WHERE ENAME != ‘KING’;

13 rows updated.

Note that we have been careful to include the final WHERE clause, in this case to avoid updating KING's MGR field.

To verify that the updates have taken place correctly:

SELECT EMPNO,ENAME,MGR

FROM EMPCOPY;

EMPNO	ENAME	MGR
7369	SMITH	7839
7499	ALLEN	7839
7521	WARD	7839
7566	JONES	7839
7654	MARTIN	7839
7698	BLAKE	7839
7782	CLARK	7839
7788	SCOTT	7839
7839	KING	
7844	TURNER	7839
7876	ADAMS	7839
7900	JAMES	7839
7902	FORD	7839
7934	MILLER	7839

Note that all MGR fields, except that of KING, have been set to 7839, which is of course KING's EMPNO. It is a nice feature of the SQL language that we were able to code this query without knowing KING's EMPNO value, though we did have to know something unique about KING in order to retrieve the EMPNO value from the table. In this case, we used the value of ENAME, but this is in general unsafe - it would have been better to use KING's EMPNO value. Why? We could equally have used the value of JOB, providing we could rely on there being only one President in the table.

Removing rows with DELETE

The DELETE statement is the last of the DDL statements we shall look at in detail. It is used to remove single rows or groups of rows from a table. Its format is as follows:

DELETE FROM "TABLE NAME"

WHERE "COLUMN-LIST" = | IN

CONSTANT | EXPRESSION | SUB-QUERY;

As for the UPDATE statement, if the WHERE clause is omitted, all of the rows will be removed from the table. However, unlike the DROP TABLE statement, a DELETE statement leaves the table structure in place.

Example 1: To remove an individual employee from the EMPCOPY table:

DELETE FROM EMPCOPY

WHERE ENAME = 'FORD';

1 row deleted.

Note that, had there been more than one employee called FORD, all would have been deleted.

Example 2: To delete a number of rows based on an expression: To remove all employees paid more than 2800:

DELETE FROM EMPCOPY

WHERE SAL > 2800;

5 rows deleted.

Example 3: Deleting using a sub-query: To remove any employees based in the SALES department:

DELETE FROM EMPCOPY

WHERE DEPTNO IN

(SELECT DEPTNO FROM DEPT WHERE DNAME = 'SALES');

5 rows deleted

Creating views in SQL

Views are an extremely useful mechanism for providing users with a subset of the underlying data tables. As such, they can provide a security mechanism, or simply be used to make the user's job easier by reducing the rows and columns of irrelevant data to which users are exposed.

Views are the means by which, in SQL databases, individual users are provided with a logical, tailored schema of the underlying database. Views are in effect virtual tables, but appear to users in most respects the same as normal base tables. The difference is that when a view is created, it is not stored like a base table; its definition is simply used to recreate it for use each time it is required. In this sense, views are equivalent to stored queries.

Views are created using the CREATE VIEW statement. The syntax of this statement is very similar to that for creating tables using a SELECT.

Example: To create a view showing the names and hiredates of employees, based on the EMP table:

CREATE VIEW EMPHIRE

AS SELECT ENAME,HIREDATE

FROM EMP;

View created.

To examine the structure of the view EMPHIRE, we can use the DESCRIBE command, just as for table objects:

DESCRIBE EMPHIRE

NAME	NULL?	TYPE
ENAME		VARCHAR2(10)
HIREDATE		DATE

To see the data in the view, we can issue a SELECT statement just as if the view EMPHIRE is a table:

SELECT *

FROM EMPHIRE;

ENAME	HIREDATE
SMITH	17-DEC-80
ALLEN	20-FEB-81
WARD	22-FEB-81
JONES	02-APR-81
MARTIN	28-SEPT-81
BLAKE	01-MAY-81
CLARK	09-JUN-81
SCOTT	19-APR-87
KING	17-NOV-81
TURNER	08-SEP-81
ADAMS	23-MAY-87
JAMES	03-DEC-81
MILLER	23-JAN-82
FORD	03-DEC-81

Views and updates

When specifying the rows and columns to be included in a view definition, we can use all of the facilities of a SELECT statement. However, there are a number of situations in which the data in base tables cannot be updated via a view. These are as follows:

- When the view is based on one table, but does not contain the primary key of the table.
- When the view is based on a JOIN.
- When a view is based on a GROUP BY clause or aggregate function, because there is no underlying row in which to place the update.
- Where rows might migrate in or out of a view as a result of the update being made.

Renaming tables

The syntax of this extremely useful command is as follows:

RENAME “old table name” TO “new table name”;

Example: to rename the EMP table to NEWEMP:

RENAME EMP TO NEWEMP;

Table renamed.

The RENAME command is an extremely useful one when carrying out DDL operations. This is in part because of the shortcomings of the ALTER TABLE statement, which makes it necessary sometimes to copy a sub- or super-set of the table, drop the former version of the table, and rename the new version to the old.

For example, if we wish to remove a column from a table, it is necessary to do the following:

1. Use the CREATE TABLE statement to make a copy of the old table, excluding the column which is no longer required.
2. Drop the old copy of the table.
3. Rename the new copy of the table to the old.

Creating and deleting a database

SQL allows us to create and drop a database in an easy way.

Creating a database uses the following syntax:

CREATE SCHEMA database-name;

For example, to create a database called STUDENTS that holds student information, we write the create command as follows:

CREATE SCHEMA students;

Deleting the database is also fairly easy:

DROP SCHEMA database-name;

For example, to delete the student database, we write the delete command as follows:

DROP SCHEMA students;

Warning: Be careful when using the DROP SCHEMA command. It deletes all the tables created under that database, including the data.

Using SQL scripts

SQL statements can be combined into a file and executed as a group. This is particularly useful when it is required to create a set of tables together, or use a large number of INSERT statements to enter rows into tables. Files containing SQL statements in this way are called SQL script files. Each separate SQL statement in the file must be terminated with a semi-colon to make it run. Comments can be included in the file with the use of the REM statement, e.g.

REM insert your comment here

The word REM appears at the start of a new line in the script file. REM statements do not require a semi-colon (;) terminator.

Having created one or more tables, if you then decide you wish to make changes to them, some of which may be difficult or impossible using the ALTER TABLE statement, the simplest approach is to drop the tables and issue a new CREATE TABLE statement which implements the required changes. If the tables whose structures you wish to change contain any data you wish to retain, you should first use CREATE TABLE with a sub-query to copy the data to another table, from which it can be copied back when you have carried out the required table restructuring.

The restructuring of a number of tables is best implemented by including the required CREATE TABLE statements in a script file. To avoid errors when this file is re-run, it is customary to place a series of DROP TABLE statements at the beginning of the file, one for each table that is to be created. In this way you can re-run the script file with no problems. The first time it runs, assuming the tables have not already been created outside the script, the DROP TABLE statements will raise error messages, but these can be ignored. It is of course essential, if the tables contain data, to ensure this has been copied to other tables before such a restructuring exercise is undertaken.

Activities

Activity 1: Data definition language

1. Create the following tables, choosing appropriate data types for the attributes of each table. In your CREATE TABLE statements, create primary keys for both tables, and an appropriate foreign key for the student table.

Tables:

TUTOR (TUTOR_ID, TUTOR_NAME, DEPARTMENT, SALARY, ADVICE_TIME)

STUDENT(STUDENT_NO, STUDENT_NAME, DATE_JOINED, COURSE, TUTOR_ID)

Important note: Because of the foreign key constraint, you should create the TUTOR table first, so that it will be available to be referenced by the foreign key from the STUDENT table when it is created.

Use DESCRIBE to check the table structure.

2. Perform the following checks and modifications to the tables above. After each modification, use DESCRIBE to verify the change has been made correctly.

Add an ANNUAL_FEE column to the STUDENT table.

Ensure that the STUDENT_NO field is sufficiently large to accommodate over 10,000 students. If it is not, change it so that it can deal with this situation.

Add an ANNUAL_LEAVE attribute to the tutor table.

Ensure that the tutor's salary attribute can handle salaries to a precision of two decimal places. Remove the ADVICE_TIME attribute from the tutor table.

Activity 2: Manipulating rows in tables

1. Populate the TUTOR and STUDENT tables with appropriate data values. Ensure that some of the student records you insert have null values in their foreign key of TUTOR_ID, and that other students have foreign key values which match the TUTOR_IDs of tutors in the TUTOR table. To place null values into the TUTOR_ID attribute for a student, you need to put the word 'null' in the position where the TUTOR_ID would appear in the column-list of the INSERT statement; for example:

INSERT INTO STUDENT VALUES (1505,'KHAN','04-OCT-1999','COMPUTING',NULL,5000);

1 row created.

Note that because inserting data a row at a time with the INSERT statement is rather slow, it is only necessary to put small samples of tutors and students into the tables; for example, about four tutor records and eight student records should be sufficient.

2. Use CREATE TABLE with a sub-query to make copies of your TUTOR and STUDENT tables before you proceed to the following steps of the activity, which involve updating and removing data. Having done this, if you accidentally remove more data than you intended, you can copy it back from your backup tables by dropping the table, and then using the CREATE TABLE statement with a sub-query.

3. Update a specific student record in order to change the course he or she is attending.

4. Update the TUTOR_ID of a specific student in order to change the tutor for that student. Write the update statement by using the tutor's name, in order to retrieve the TUTOR_ID supplying the update value.
5. Remove all students who do not have a tutor.

Activity 3: Creating and removing views

1. Create two views on the STUDENT and TUTOR tables as follows:

- View1 should contain details of all students taking Computing.
- View2 should include the names of tutors and the names of their tutees.

2. Remove the two views using the DROP VIEW statement.

Review questions

1. Describe the ways in which tables can be created in SQL.
2. What details need to be included on a CREATE TABLE statement to establish a foreign key?
3. Briefly describe the functionality of the ALTER TABLE statement, and describe some of the limitations in its use.
4. What happens to the data in a table when that table is dropped?
5. Describe the forms that the INSERT statement can take.
6. What options are there for supplying values to update columns in an UPDATE statement?
7. What happens to a table structure when rows are deleted from the table?
8. The command-oriented nature of the SQL language means that it does not contain the usual confirmation messages if requests are made to remove data or storage structures such as tables. Identify the SQL statements where it is necessary to pay particular attention to the statement specification, in order to avoid unwanted changes to data or data structures. Identify which parts of the statements require specific attention in this way.
9. Describe two uses of views in database systems, and identify any limitations in their use.

Discussion topic

The strengths and weaknesses of SQL

The three chapters covering SQL have introduced a wide range of mechanisms for querying and manipulating data in relational tables. This is not the complete story as far as SQL is concerned, but you have now encountered a major part of the facilities available within standard implementations of the language. You are encouraged to discuss with your colleagues your views on the SQL language that you have been learning. Particular aspects of interest for discussion include:

- What do you feel are the strengths of the language, in terms of learnability, usability and flexibility?
- On the other hand, which aspects of the language have you found difficult or awkward either to learn or to use?
- Are there ways in which you feel the language could be improved?
- How does use of the SQL language compare with other database systems or programming languages you have encountered?
- How feasible is it to use natural language (e.g. English) statements instead of SQL, to retrieve data in an Oracle database? What are the potential problems and how might they be overcome?

Additional content and activities

The SQL language, as we have seen, provides a standardised, command-based approach to the querying and manipulation of data. Most database systems also include Graphical User Interfaces for carrying out many of the operations that can be performed in SQL. You are encouraged to explore these interfaces, either for the Microsoft Access database system, and/or for the Personal Oracle system you have installed to carry out the practical work so far.

The Microsoft Access system does not provide the DDL part of SQL, relying on its graphical front-end for the creation and alteration of tables. Examine the ways in which new tables are established or changed within the Microsoft Access environment, comparing it with the approach in SQL.

The Personal Oracle system includes a graphical tool called the Navigator, which provides a graphical means of carrying out a large number of database administration tasks.

Examine the facilities in the Navigator for creating tables and other data objects, again comparing it with the equivalent mechanisms in SQL.

Chapter 6. Entity-Relationship Modelling

Table of contents

        - Optional for both entities
    - Converting one-to-many relationships into relations
        - Mandatory for both entities
        - Mandatory for one entity, optional for another entity: many end mandatory
        - Mandatory for one entity, optional for another entity: many end optional
        - Optional for both entities
    - Converting many-to-many relationships into relations
        - Mandatory for both entities
        - Mandatory for one entity, optional for the other entity
        - Optional for both entities
    - Summary of conversion rules
- Review questions

Objectives

At the end of this chapter you should be able to:

- Analyse a given situation to identify the entities involved.
- Be able to identify the relationships between entities, and carry out any necessary transformations.
- Develop the model further by identifying attributes for each entity.
- Map the entities into tables suitable for Relational database implementation.

Introduction

In parallel with this chapter, you should read Chapter 11 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter is the first to address in detail the extremely important topic of database design. The main approach described in this chapter is called Entity-Relationship Modelling. This technique has become a widely used approach in the development of database applications. The approach is essentially top-down, in that the first step is to look overall at the requirements for the application being developed, identifying the entities involved. The approach progresses from that point through the development of a detailed model of the entities, their attributes and relationships. The Entity-Relationship Modelling process is not formal, in the mathematical sense, but to be done well, it requires a consistent precision to be applied to the way that entities, their relationships and their attributes are discussed. The approach can be supplemented by methods which

are more formal in their approach, and that provide a bottom-up perspective to the design process. The most commonly used of these approaches is Normalisation, which will be a core topic of the later chapters on database design.

Context

This chapter introduces the ideas of top-down database design, and provides the starting point in learning how to develop a database application. The chapter links closely with the others covering database design (Normalisation and other design topics). The chapter also has considerable relevance for the material in the module on performance tuning, such as the chapter on indexing, as the decisions made during database design have a major impact on the performance of the application.

Entities, attributes and values

Entities

Many organisations (such as businesses, government departments, supermarkets, universities and hospitals) have a number of branches, divisions or sections in order to deal with a variety of functions or different geographical areas. Each branch, division or section may itself be split up into smaller units. It is possible to regard each branch, division or section (or each unit within these) as an organisation in its own right. Organisations require information in order to carry out the tasks and activities for which they are responsible. The information that these organisations need could be categorised in a number of ways, for example:

People

- Payroll
- Pensions
- Annual leave
- Sick leave

Things

- Furniture
- Equipment
- Stationery
- Fire extinguishers

Locations

- Offices

- Warehouses
- Stock rooms

Events

- Sale is made
- Purchase order is raised
- Item is hired
- Invoice is issued

Concepts

- Image of product
- Advertising
- Marketing
- Research and development.

Each of these can be regarded as an entity.

Important

Entity

An entity may represent a category of people, things, events, locations or concepts within the area under consideration. An entity instance is a specific example of an entity. For example, John Smith is an entity instance of an employee entity.

Attributes

Entities have attributes. The following are typical of the attributes that an entity might possess:

Entity: House

Attributes:

Number of bedrooms	Location	Area of grounds	type

Entity: Book

Attributes:

Author	Title	Category	publisher	ISBN

Entity: Employee

Attributes:

Name	address	Job title	division	staff number

Important

Attribute

An entity may have one or more attributes associated with it. These attributes represent certain characteristics of the entity; for a person, attributes might be name, age, address, etc.

Values

Using the entities and attributes shown above, the following are examples of one set of values for a particular instance of each entity. Every occurrence of an entity will have its own set of values for attributes it possesses.

Entity: House

Attributes:

number of bedrooms	Location	area of grounds	Type

Values:

3	London	75 sq metres	Terraced
5	Paris	125 sq metres	Detached

Entity: Book

Attributes:

Author	Title	category	publisher	ISBN

Values:

Hanson	Data Files	Computing	Pitman	580239
Carter	Night Sun	Fiction	Portland	504297

Entity: Employee

Attributes:

Name	Address	Job title	division	Staff number

Values:

Tan	24 Barn Lane	Manager	Customer Liaison	23563
Smith	99 Red Road	Accountant	Finance	93845

Primary key data elements

If the value of certain attributes (or perhaps just one attribute) is known for a particular entity, this enables us to discover the value of other attributes associated with that entity. The attributes (or attribute) which possess this quality are known as keys, because they are able to 'unlock' the values of the other attributes that are associated with that specific instance of an entity. Why do we need a key? Suppose we had two members of staff with the same (or similar) names, such as Linda Clark and Lydia Clark. It would be a simple mistake to record something in the file of Linda Clark that should be kept in the file for Lydia Clark (or the other way around). It would be even more difficult to tell them apart if the name was given as just an initial and surname.

Some names may be spelt slightly differently, but sound similar (such as Clark and Clarke), and therefore pose a further risk of identifying the wrong member of staff.

Key

The addition of a staff number as the primary key would enable us to be sure that when we needed to refer to one or other of these members of staff, we had identified the correct individual. In this way 11057 Clark can be distinguished from 28076 Clark.

The following are examples of key data elements:

- The payroll number (primary key) of a member of staff enables us to find out the name, job title and address for that individual.

- The account number (primary key) enables us to find out whether the balance of that account is overdrawn.
- The item code (primary key) in a stationery catalogue enables us to order a particular item in a particular size and colour (e.g. a red A4 folder).

Sometimes we may need to use more than one attribute in order to arrive at a key that will provide unique identification for all the other data elements. When considering which attribute (or combination of attributes) might be used as a primary key, these attributes are known as candidate keys.

Candidate keys

Where there is more than one set of attributes which could be chosen as the primary key for an entity, each of these groups of attributes are known as candidate keys.

A company might choose either an employee's staff number or an employee's National Insurance number as the primary key, as each will provide unique identification of an individual. (Note that in different countries, a slightly different term might be used for a national code that is used to identify any one individual, such as national ID number, etc.) The staff number and the National Insurance number are candidate keys, until one is selected as the primary key.

At times we may refer to a collection of attributes that includes the primary key (for example, staff number and staff name); this group of attributes is sometimes known as a superkey.

When we need to connect together different items of data (for example, customers and items, in order to produce orders and invoices), we can do this by including the primary key of one entity as a data item in another entity; for example, we would include the primary key of Customer in the Order entity to link customers to the Orders they have placed.

Foreign keys

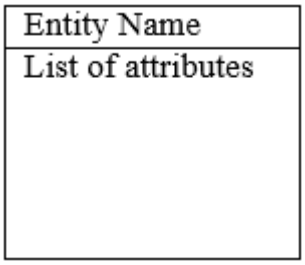
When a copy of the primary key for one entity is included in the collection of attributes of another entity, the copy of the primary key held in the second entity is known as a foreign key.

A foreign key enables a link to be made between different entities.

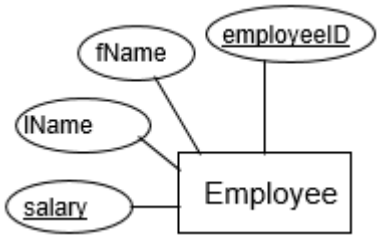
Entity-Relationship Modelling

Entity representation

One common method to represent an entity is to use entity-relationship diagrams, where each entity is represented by a box with two compartments, the first for entity name and the second for attributes.



You may also come across diagrams that employ ellipses to represent the attributes belonging to each entity.



The relationships that exist between two entities can be categorised according to the following:

- one-to-one
- one-to-many
- many-to-many

In some cases, for simplicity, the attributes are omitted in the entity diagram.

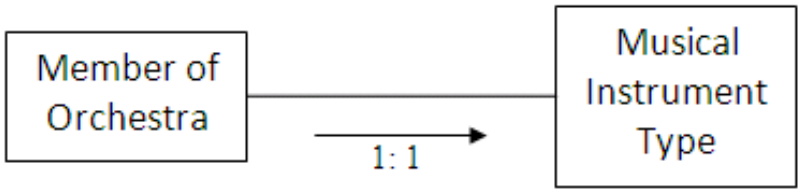
One-to-one relationships between two entities

In a concert hall, each ticket holder has a seat for a single performance (the seat number will appear on the ticket). Only one person can sit in one seat at each performance; the relationship between a member of the audience and a seat is therefore one-to-one.

Each seat in the concert hall can be sold to one person only for a particular performance; the relationship between the seat and the member of the audience with a ticket for that seat is also one-to-one.

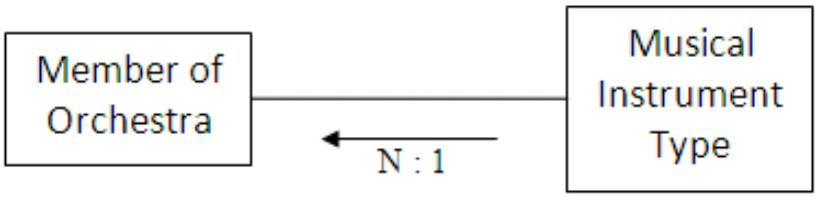
Relationships between entities and attributes, between attributes, and between entities can be shown in a variety of diagrammatic formats. The common format is to represent each relationship as a line. The style of the line shows the type of relationship being represented. Here, in order to represent a one-to-one relationship, a single straight line is used between the two entities.

The overall relationship between ticket holders and seats is one-to-one for each performance. The entity-relationship diagram above shows the one-to-one link between a ticket holder and a concert hall seat.



In an orchestra, each individual will play one type of musical instrument; for example, a person who plays a violin will not play a trumpet. The relationship is one-to-one from a member of the orchestra to a type of instrument.

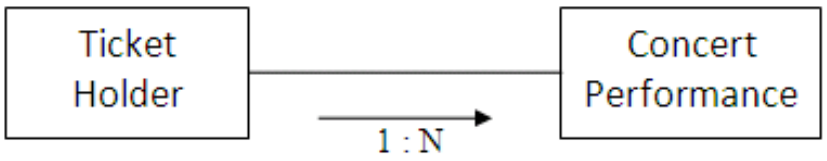
One-to-many relationships between two entities



An orchestra will have more than one musician playing a particular type of instrument; for example, it is likely that there will be several members of the orchestra each playing a violin. The relationship is therefore one-to-many from a type of musical instrument to a member of the orchestra.

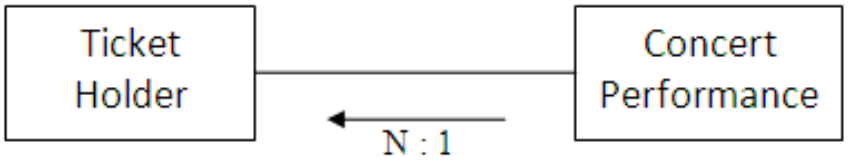
The entity-relationship diagram shows that there is a one-to-many relationship between musical instrument types and members of the orchestra. The 'crow's foot' link shows that there may be more than one member of the orchestra for each type of musical instrument.

Many-to-many relationships between two entities



An individual may attend a series of concerts during each season as a member

of the audience; the relationship between an individual and the concerts is one-to-many.



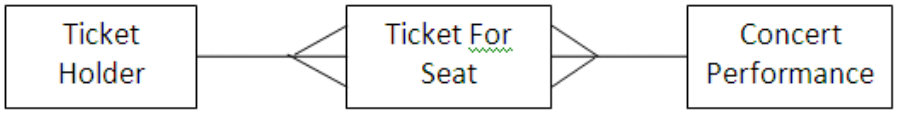
Many ticket holders will attend each concert; the relationship between a concert and members of the audience is also one-to-many.

As the relationship is one-to-many on both sides of the relationship, the relationship that exists between the two entities can be described as many-to-many.

The entity-relationship diagram above has a 'crow's foot' connection at each end, illustrating that there is a many-to-many relationship between ticket holders and concert performances, as one ticket holder may attend many performances, and each performance is likely to have many ticket holders present.

As it is difficult to implement a many-to-many relationship in a database system, we may need to decompose a many-to-many relationship into two (or more) one-to-many relationships. Here, we might say that there is a one-to-many relationship between a ticket holder and a ticket (each ticket holder may have several tickets, but each ticket will be held by only one person).

We could also identify a one-to-many relationship between a concert performance and a ticket (each ticket for a particular seat will be for only one performance, but there will be many performances each with a ticket for that seat).



This allows us to represent the many-to-many relationship between ticket holder and concert performance: two one-to-many relationships involving a new entity called Ticket For Seat. This new structure can then be implemented within a Relational database system.

Recursive relationships

The relationships we have seen so far have all been between two entities; this does not have to be the case. It is possible for an entity to have a relationship with itself; for example, an entity Staff could have a relationship with itself, as one member of staff could supervise other staff. This is known as a recursive or involute relationship, and would be represented in an entity-relationship diagram as shown below.

Exercises

Exercise 1: Identifying entities and attributes

Benchmarque International, a furniture company, keeps details Of items it supplies to homes and offices (tables, chairs, bookshelves, etc). What do you think would be the entities and attributes the furniture company would need to represent these items?

Exercise 2: Identification of primary keys

What do you think would make a suitable primary key for the entity (or entities) representing the tables, chairs, bookshelves and other items of furniture for Benchmarque International?

In other words, what are the candidate keys?

Exercise 3: Identifying relationships

At a conference, each delegate is given a bound copy of the proceedings, containing a copy of all the papers being presented at the conference and biographical details of the speakers.

What is the relationship between a delegate and a copy of the proceedings?

Draw the entity-relationship diagram.

Exercise 4: Identifying relationships II

Many papers may be presented at a conference.

Each paper will be presented once only by one individual (even if there are multiple authors).

Many delegates may attend the presentation of a paper.

Papers may be grouped into sessions (two sessions in the morning and three in the afternoon).

What do you think is the relationship between:

- a speaker and a paper
- a paper and a session

Exercise 5 — Identifying relationships III

A conference session will be attended by a number of delegates. Each delegate may choose a number of sessions. What is the relationship between conference delegates and sessions? Draw the entity-relationship diagram.

Relationship participation condition (membership class)

Mandatory and optional relationships

We can extend the entity-relationship model by declaring that some relationships are mandatory, whereas others are optional. In a mandatory relationship, every instance of one entity must participate in a relationship with another entity. In an optional relationship, any instance of one entity might participate in a relationship with another entity, but this is not compulsory.

Important

Participation condition/membership class

The participation condition defines whether it is mandatory or optional for an entity to participate in a relationship. This is also known as the membership class of a relationship.

As there are two kinds of participation conditions (mandatory and optional), and most entities are involved in binary relationships, it follows that there are four main types of membership relationships, as follows:

1. Mandatory for both entities
2. Mandatory for one entity, optional for the other
3. Optional for one entity, mandatory for the other
4. Optional for both entities

It might be tempting to think that options 2 and 3 are the same, but it is important to recognise the difference, particularly when thinking about whether the relationship is one-to-one, one-to-many or many-to-many. A useful analogy is to think of a bank, with customers who have savings accounts and loans. It may be the bank's policy that any customer must have a savings account before they are eligible to receive a loan, but not all customers who have savings accounts will require a loan.

We can examine how these different types of membership classes can be used to reflect the policies of allocating staff within departments. We would expect any member of staff in an organisation to work in a given department, but what happens if a new department is created, or a new member of staff joins? If we look at each combination in turn, we can see what the possibilities are:

1. **Mandatory for both entities:** A member of staff must be assigned to a given department, and any department must have staff. There can be no unassigned staff, and it is not possible to have an 'empty' department.
2. **Mandatory for one entity, optional for the other:** Any member of staff must be attached to a department, but it is possible for a department to have no staff allocated.
3. **Optional for one entity, mandatory for the other:** A member of staff does not have to be placed in a department, but all departments must have at least one member of staff.
4. **Optional for both entities:** A member of staff might be assigned to work in a department, but this is not compulsory. A department might, or might not, have staff allocated to work within it.

We can elaborate the standard entity-relationship notation with a solid circle to indicate a mandatory entity, and a hollow circle for an optional entity (think of the hollow circle like 'o' for optional). (You may find alternative notations in other texts - for example, a solid line to represent a mandatory entity, and a dotted line to indicate an optional entity. Another method places solid circles inside entity boxes for mandatory participation, or outside entity boxes for optional membership.) The use of a graphical technique enables us to represent the membership class or participation condition of an entity and a relationship in an entity-relationship diagram.

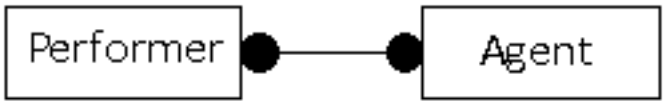
We will now explore these possibilities using a performer, agents and bookings scenario as an example, but experimenting with different rules to see what effect they have on the design of the database. Supposing to start with, we have the following situation.

There are a number of performers who are booked by agents to appear at different venues. Performers are paid a fee for each booking, and agents earn commission on the fee paid to each performer. We will now consider relationships of different kinds between these entities.

One-to-one relationships and participation conditions

Both ends mandatory

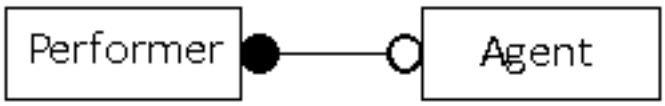
It might be the case that each performer has only one agent, and that all bookings for any one performer must be made by one agent, and that agent may only make bookings for that one performer. The relationship is one-to-one, and both entities must participate in the relationship.



The solid circle at each end of the relationship shows that the relationship is mandatory in both directions; each performer must have an agent, and each agent must deal with one performer.

One end mandatory, other end optional:

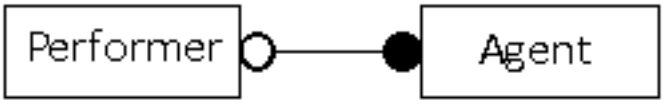
It might be possible for agents to make bookings that do not involve performers; for example, a venue might be booked for an art exhibition. Each performer, however, must have an agent, although an agent does not have to make a booking on behalf of a performer.



The solid circle at the performer end of the relationship illustrates that a performer must be associated with an agent. The hollow circle at the agent end of the relationship shows that an agent could be associated with a performer, but that this is not compulsory. Each performer must have an agent, but not all agents represent performers.

One end optional, other end mandatory:

It might be possible for performers to make bookings themselves, without using an agent. In this case, one performer might have an agent, and that agent will make bookings for that performer. On the other hand, a different performer might elect to make their own bookings, and will not be represented by an agent. All agents must represent a performer, but not all performers will be represented by agents. The relationship is optional for the performer, but mandatory for the agent, as shown in the diagram below.



The solid circle at the agent end of the relationship shows each agent must be associated with a performer. The hollow circle at the performer end of the relationship indicates that a performer could be represented by an agent, but that this is not compulsory. Each agent must deal with only one performer, but each performer does not have to have an agent.

Both ends optional:

Another possibility is that agents may make bookings that do not involve performers; for example, a venue might be booked for an art exhibition. In addition, performers may make bookings themselves, or might have bookings made by an agent, but if a performer has an agent, there must be a one-to-one relationship between them. This relationship is optional for both entities.

The hollow circles show that there is an optional relationship between a performer and an agent; if there is a relationship, it will be one-to-one, but it is not compulsory either for the performer or for the agent.

One-to-many relationships and participation conditions

It might be the case that a performer has only one agent, and that all bookings for any one performer must be made by one agent, although any agent may make bookings for more than one performer.

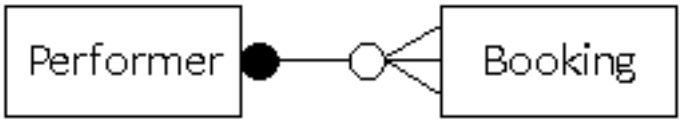
Both ends mandatory:

A performer must have one or more bookings; each booking must involve one performer.

The membership class is mandatory for both entities, as shown by the solid circle. In this case, it is not possible for a booking to be made for an event that does not involve a performer (for example, a booking could not be for an exhibition).

One end mandatory, other end optional:

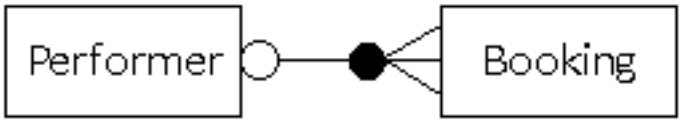
A performer must have one or more bookings, but a booking might not involve a performer (e.g. a booking might be for an exhibition, not a performer).



The solid circle shows the compulsory nature of the relationship for a performer; all performers must have bookings. The hollow circle shows that it is optional for a booking to involve a performer. This means that a performer must have a booking, but that a booking need not have a performer.

One end optional, other end mandatory:

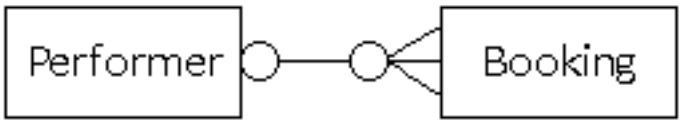
A performer might have one or more bookings; each booking must involve one performer.



The membership class is mandatory for a booking, but optional for a performer. This means that it would not be possible for a booking to be for an exhibition, as all bookings must involve a performer. On the other hand, it is not compulsory for a performer to have a booking.

Both ends optional:

A performer might have one or more bookings; a booking might be associated with a performer.



In this case, a booking could be for an exhibition as it is optional for a booking to involve a performer, as indicated by the hollow circle. A performer might decline to accept any bookings; this is acceptable, as it is optional for a performer to have a booking (shown by the hollow circle).

Many-to-many relationships and participation conditions

We could say that there is a many-to-many relationship between performers and agents, with each agent making bookings for many performers, and each performer having bookings made by many agents. We know that we need to decompose many-to-many relationships into (usually) two one-to-many relationships, but we can still consider what these many-to-many relationships would look like before this decomposition has taken place. We will see later that many-to-many relationships can be converted into relations either after they have been decomposed, or directly from the many-to-many relationship. The result of the conversion into relations will be the same in either case.

Both ends mandatory:

An example here might be where each performer must be represented by one or more agents, and each agent is required to make bookings for a number of performers.

There is a many-to-many relationship between the two entities, in which both entities must participate. Agents are not allowed to make bookings for events that do not involve performers (such as conferences or exhibitions). Performers must have bookings made by agents, and are not allowed to make their own bookings.

One end mandatory, other end optional:

In this example, it is still necessary for performers to be represented by a number of agents, but the agents now have more flexibility as they do not have to make bookings for performers.

There is a many-to-many relationship between the two entities; one must participate, but it is optional for the other entity.

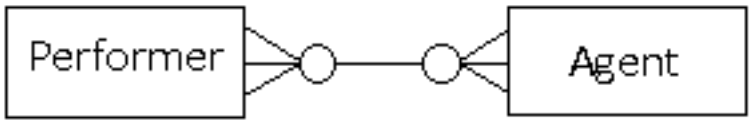
One end optional, other end mandatory:

Here, performers have the flexibility to make their own bookings, or to have bookings made by one or more agents. Agents are required to make bookings for performers, and may not make arrangements for any other kind of event.

There is a many-to-many relationship between the two entities; it is optional for one to participate, but participation is mandatory for the other entity.

Both ends optional

Here, performers and agents are both allowed a degree of flexibility. Performers may make their own bookings, or may have agents make bookings for them. Agents are permitted to make bookings for a number of performers, and also have the ability to make other kinds of bookings where performers are not required.



There is a many-to-many relationship between the two entities; participation is optional for both entities.

These many-to-many relationships are likely to be decomposed into one-to-many relationships. The mandatory/optional nature of the relationship must be preserved when this happens.

Weak and strong entities

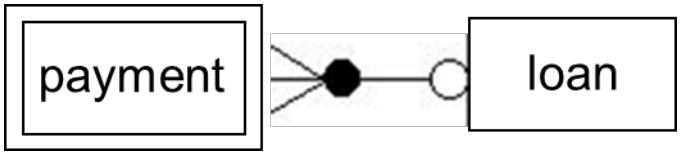
An entity set that does not have a primary key is referred to as a weak entity set. The existence of a weak entity set depends on the existence of a strong entity set, called the identifying entity set. Its existence, therefore, is dependent on the identifying entity set.

The relationship must be many-to-one from weak to identifying entity. Participation of the weak entity set in the relationship must be mandatory. The discriminator (or partial key) of a weak entity set distinguishes weak entities that depend on the same specific strong entity. The primary key of a weak entity is the primary key of the identifying entity set + the partial key of the weak entity set.

Example: Many payments are made on a loan

- Payments don't exist without a loan.
- Multiple loans will each have a first, second payment and so on. So, each payment is only unique in the context of the loan which it is paying off.

The weak entity is commonly represented by two boxes.



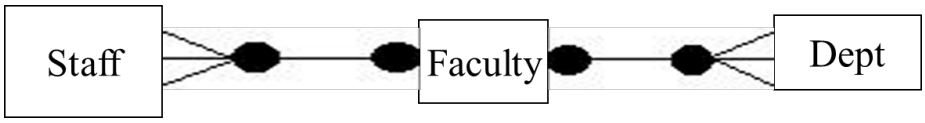
The payment is a weak entity; its existence is dependent on the loan entity.

Problems with entity-relationship (ER) models

In this section we examine problems that may arise when creating an ER model. These problems are referred to as connection traps, and normally occur due to a misinterpretation of the meaning of certain relationships. We examine two main types of connection traps, called fan traps and chasm traps, and illustrate how to identify and resolve such problems in ER models.

Fan traps

These occur when a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous. Look at the model below.

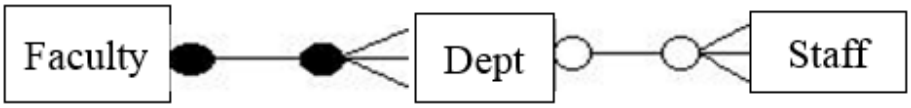


The above model looks okay at first glance, but it has a pitfall. The model says a faculty has many departments and many staff. Although the model seems to capture all the necessary information, it is difficult to know which department staff are affiliated to. To find out the departments the staff belong to, we will start from the staff entity. Through the relationship between staff and faculty, we are able to easily identify the faculties staff belong to. From the faculty, it's difficult to know the exact department because one faculty is associated with many departments.

The model below removes the fan trap from the model.

Chasm traps

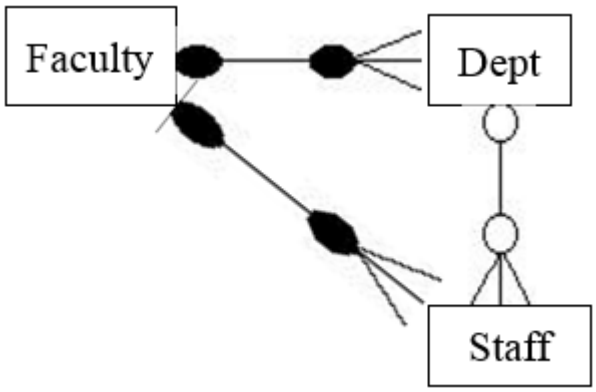
These occur when a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.



The model represents the facts that a faculty has many departments and each department may have zero or many staff. We can clearly note that, not all departments have staff and not all staff belong to a department. Examples of such staff in a university can include the secretary of the dean. He/she does not belong to any department.

It's difficult to answer the question, "Which faculty does the dean's secretary belong to?", as the secretary to the dean does not belong to any department.

We remove the 'chasm trap' by adding an extra relationship from staff to faculty.



Converting entity relationships into relations

When we have identified the main entities and the relationships that exist between them, we are in a position to translate the entity-relationship model we have created from a diagram into tables of data that will form the relations for our database. The nature of the relationships between entities will make a difference to the nature of the relations we construct; the cardinality, degree and membership class will all affect the structure of the database.

If we design a database by using an entity-relationship model, we need to be able to convert our design from a diagrammatic format into a series of relations that will hold the values of the actual data items.

It would be possible to create a number of relations so that each represented either an entity or relationship. This approach would generate a relational database that represented the entities and the relationships between them as identified in our data model, but it would suffer from a number of disadvantages. One disadvantage would be that the number of relations created could result in the database being unnecessarily large. There are also a number of insertion, update and deletion anomalies, which will be examined in the chapter on Normalisation, to which a database created in such a way would be vulnerable. To avoid these problems, we need to specify a method that allows us to create only those relations that are strictly necessary to represent our data model as a database. The way we do this is guided by the nature of the relationships between the entities, in terms of the cardinality and the membership class (participation condition).

Converting one-to-one relationships into relations

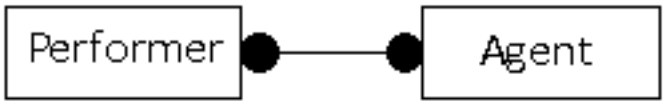
We can transform entity-relationship diagrams into relations by following simple rules which will specify the number of relations needed, depending on the car-

dinality (one-to-one, one-to-many or many-to-many) and the membership class (mandatory or optional) of the entities participating in the relationship. In the case of one-to-one relationships, the creation of one or two relations is sufficient, depending on whether participation is mandatory or optional.

Mandatory for both entities

A single relation will be able to represent the information represented by each entity and the relationship that exists between them.

If we consider an earlier example, with a one-to-one mandatory relationship between performers and agents, this could now be converted from a diagram into a relation as part of our database.



This part of an entity-relationship model can be converted into a single relation, Performer-details. This relation holds information about all the performers and their agents. The agents do not need to be held in a separate relation as each performer has one agent, and each agent represents only one performer.

Relation: Performer-details

Perf -id	Perf-name	Perf -type	Perf -Loc'n	Agent -id	Agent -name	Agent -Loc'n
0548	Takis Bakalis	Comedian	York	1653	Smith	Moscow
0556	Mary Marsh	Singer	Dublin	1304	Alton	Sydney
0717	Anjuli Misra	Dancer	Paris	1592	West	Penang
0832	David Ho	Actor	Beijing	1727	Elgin	Rome

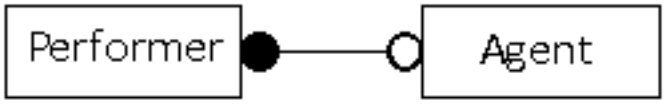
In the relation Performer-details above, we can see that all performer information is stored and can be accessed by the performer-id attribute, and all agent information can be extracted by means of the agent-id attribute.

As the relationship is one-to-one and mandatory in both directions, we do not need to store the performers and agents in separate relations, although we could choose to do so. (If we stored performers and agents in separate relations, we would then need to use the identifying attributes of performer-id and agent-id as foreign keys. This means that we would be able to identify the relevant agent in the Performer relation, and identify the appropriate performer in the Agent relation.)

Mandatory for one entity, optional for the other entity

In this case, two relations will be needed, one for each entity. The relationship could be mandatory for the first entity and optional for the second, or the other way around. There are therefore two possibilities for performers and agents.

In this first example, a performer must be represented by an agent, but an agent does not have to represent a performer. The relationship is therefore mandatory for a performer, but optional for an agent.



This would convert into two relations, one for each entity. The agent identifier is stored in the Performer relation in order to show the connection between agents and performers where appropriate. This is known as posting an identifier (or posting an attribute). It is important that the value of a posted identifier is not null.

Relation: Performer

Perf-id	Perf-name	Perf-type	Perf-Loc'n	Agent-id
1589	Thomas Wong	Magician	Taipei	4837
1873	Shirley Sure	Dancer	Chicago	5490
1903	Darryl Burns	Comedian	Berlin	2936
2005	Charlotte Chong	Musician	Beijing	3895

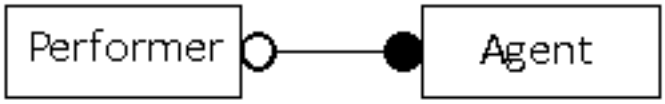
Note that the agent identifier, agent-id, is held in the Performer relation. The attribute agent-id is a foreign key in the Performer relation. This means that we can identify which agent represents a particular performer.

We would not want to store the performer-id in the Agent relation for this example, as there are agents who do not represent performers, and there would therefore be a null value for the performer-id attribute in the Agent relation. We can see that there are agents in the Agent relation who do not represent performers, but all performers are represented by only one agent.

Relation: Agent

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5490	Charles Patterson	Rome

In the second example, an agent must represent a performer, but a performer does not need to have an agent. Here, the relationship is optional for a performer, but mandatory for an agent.



Again, this would translate into two relations, one for each entity. On this occasion, however, the link between performers and agents will be represented in the Agent relation rather than the Performer relation. This is because every agent will be associated with a performer, but not all performers will be linked to agents. The performer-id is a foreign key in the Agent relation. We cannot have the agent identifiers in the Performer relation as in some instances there will be no agent for a performer, and a null value for an agent identifier is not allowed, as it would contravene the rules on entity integrity.

Relation: Performer

Perf-id	Perf-name	Perf-type	Perf-Loc'n
1597	Andrew Chase	Singer	London
1896	Michael Castle	Actor	Madrid
1976	Sau Mun Lo	Dancer	Penang
1988	William Chong	Actor	Rome
1990	David Collins	Musician	Paris

Relation: Agent

Agent-id	Agent-name	Agent-Loc'n	Perf-id
8393	Davidson	Paris	1990
8467	Gordon	Sydney	1896
8476	Lopez	Lima	1976

Optional for both entities

In this scenario, a performer might or might not have an agent. Similarly, an agent might or might not represent a performer. However, if a performer does have an agent, that agent will not represent any other performers. The relationship between the two entities is one-to-one, but optional on both sides. In order to convert this relationship into a relational format, three relations will be needed, one for each entity and one for the relationship.

This means that it is possible to have a performer without an agent, and it is also permissible for an agent to have no performers. All performer details will be stored in the Performers relation, and all agent data will be held in the Agent relation. Where there is a performer with an agent, this will be shown in the relation Works-with, which will represent the relationship between the two entities.

Relation: Performer

Perf-id	Perf-name	Perf-type	Perf-Loc'n
0549	Mavis Ringer	Dancer	Taipei
1454	Claude Poisson	Actor	Leeds
2079	Nita Chotalia	Singer	Chicago
3127	Walter Tan	Magician	Berlin

The relation Performers holds details of all the performers relevant to the database.

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
1053	Walter Woo	Penang
1279	Chris Batten	Athens
1738	Lisa Chong	London
1885	Irene Locke	Los Angeles

All agents within the database are stored in the relation Agents.

Relation: Works-with

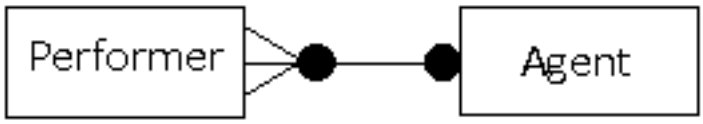
Perf-id	Agent-id
2079	1885
3127	1053

Note that the relation Works-with only has entries for those agents and performers who are linked together.

Converting one-to-many relationships into relations

Mandatory for both entities

If we consider the situation where a performer has a single agent, but each agent may represent a number of performers, and the relationship is mandatory for both entities, we have an entity-relationship as shown below.



If we convert this part of our data model into tables of data, we will have two relations (one for each entity). In order to maintain the relationship that exists between the two entities, we will hold a copy of the primary key of the entity at the "one" end of the relationship as one of the attributes associated with the entity at the "many" end of the relationship. In this example, the attribute agent-id is a foreign key in the relation Performers.

Relation: Performers

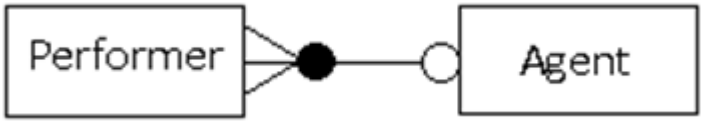
Perf-id	Perf-name	Perf-type	Perf-Loc'n	Agent-id
0468	Gerry Wise	Musician	Bombay	1971
0591	Margaret Gupta	Actor	Paris	1305
0844	Terry Peace	Singer	Milan	1971
1447	Rupert Mendez	Dancer	Sydney	2857
2718	Gloria Yeung	Actor	Toronto	2857
3762	Hsiao Kang Kim	Magician	London	2348

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
1305	Alex Sheraton	Cairo
1971	Cliff Drysdale	Rome
2348	Amy Newton	Tokyo
2857	Sarah Ng	Lisbon

Mandatory for one entity, optional for another entity: many end mandatory

In this example, all performers must be represented by agents, and each performer has only one agent. The agents themselves need not be responsible for making bookings for performers, and can be involved in other activities.



The mandatory nature of the relationship for the performer is shown by the solid circle; the hollow circle indicates an optional relationship for an agent. This means that there must be a relation to represent performers, and another relation to represent agents. The links between performers and agents are shown by having the agent identifier stored against the appropriate performer in the Performer relation. The attribute agent-id is therefore a foreign key in the Performer relation. All performers must have an agent associated with them, but not all agents will be involved in a booking for a performer.

Relation: Performers

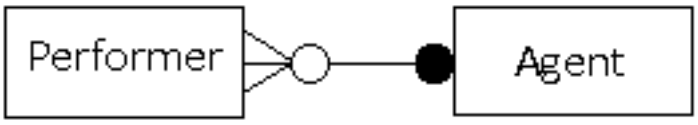
Perf-id	Perf-name	Perf-type	Perf-Loc'n	Agent-id
1204	Anna Church	Singer	Kiev	3895
1589	Thomas Wong	Magician	Taipei	4837
1873	Shirley Sure	Dancer	Chicago	5490
1903	Darryl Burns	Comedian	Berlin	2936
1982	Emily Spencer	Dancer	Paris	4837
2005	Charlotte Chong	Musician	Beijing	3895

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5490	Charles Patterson	Rome

Mandatory for one entity, optional for another entity: many end optional

Here, agents may make bookings for performers, and performers may also make bookings for themselves. It is only possible for agents to make bookings for functions that involve performers. An agent may be responsible for making bookings for more than one performer. If a performer is represented by an agent, each performer may have only one agent.



The mandatory nature of the relationship for the agent is shown by the solid circle; the hollow circle indicates an optional relationship for a performer. This means that there must be a relation to represent performers, another relation to represent agents, and a third relation to represent those occasions when performers have booked through agents. The links between performers and agents are shown by having the agent identifier stored against the appropriate

performer in the third relation.

Relation: Performers

Perf-id	Perf-name	Perf-type	Perf-Loc'n
1204	Anna Church	Singer	Kiev
1589	Thomas Wong	Magician	Taipei
1873	Shirley Sure	Dancer	Chicago
1903	Darryl Burns	Comedian	Berlin
1982	Emily Spencer	Dancer	Paris
2005	Charlotte Chong	Musician	Beijing

Relation: Agents

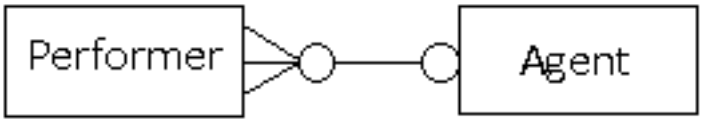
Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi

Relation: Agent-Performer

Perf-id	Agent-id
1204	2936
1873	2936
1982	3246

Optional for both entities

Here, agents may make bookings for performers, and performers may also make bookings for themselves. It is also possible for agents to make bookings for other functions that do not involve performers. An agent may be responsible for making bookings for a number of performers. If a performer is represented by an agent, each performer may have only one agent. The relationship is optional for both entities.



This relationship can be converted into three relations. There will be one relationship to represent the performers, another for the agents, and a third will store details of the relationship between performers and agents (where such a relationship exists).

Relation: Performers

Perf-id	Perf-name	Perf-type	Perf-Loc'n
1204	Anna Church	Singer	Kiev
1589	Thomas Wong	Magician	Taipei
1873	Shirley Sure	Dancer	Chicago
1903	Darryl Burns	Comedian	Berlin
1982	Emily Spencer	Dancer	Paris
2005	Charlotte Chong	Musician	Beijing

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5490	Charles Patterson	Rome

Relation: Agent-Performer

Perf-id	Agent-id
1204	3895
1589	4837
1982	4837
2005	3895

We can see from these relations that a performer may be represented by an agent, and an agent may represent more than one performer. Some performers do not have agents, and some agents do not represent performers.

Converting many-to-many relationships into relations

We know that if we are dealing with many-to-many relationships, we have to decompose them into two one-to-many relationships. Here we can see that if we leave a many-to-many relationship as it is, it will be represented by three relations just as if we had converted it into two one-to-many relationships.

Mandatory for both entities

In this example, all performers must be represented by agents, and all agents must represent performers. It is not possible for performers to represent themselves when making bookings, neither is it possible for agents to make bookings that do not involve performers. (Note that this does not imply that each performer has one agent, and each agent represents one performer; that would imply a one-to-one relationship).

Three relations are required to represent a relationship of this kind between two entities, one for each entity and one for the relationship itself, i.e. one to represent the performers, another to represent the agents, and a third to represent the relationship between the performers and the agents.

Relation: Performers

Perf-id	Perf-name	Perf-type	Perf-Loc'n
1654	Anita Hall	Dancer	Chicago
1953	Sam Wilton	Singer	London
1982	Fergus Lance	Comedian	New York
1993	Deepti Ghadia	Dancer	Milan
2002	David Tsang	Actor	Moscow
2015	Lauren Greene	Actor	Paris

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5490	Charles Patterson	Rome

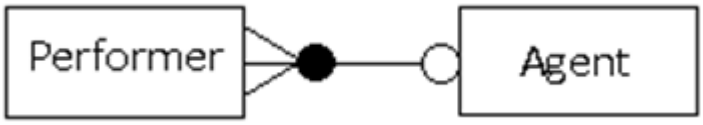
Relation: Agent-Performers

Perf-id	Agent-id
1654	5386
1654	2936
1953	5490
1982	5490
1993	4837
2002	5386
2002	3895
2015	3246

The Agent-Performers relation shows us that all performers are represented by agents, and that all agents represent performers. Some performers are represented by more than agent, and some agents represent more than one performer. We now have three relations representing the many-to-many relationship mandatory for both entities.

Mandatory for one entity, optional for the other entity

The first possibility is that the performer entity is mandatory, but the agent entity is optional. This would mean that performers cannot make bookings for themselves, but depend on a number of agents to make bookings for them. The relationship is mandatory for the performer. An agent, however, is allowed to make bookings for a number of performers, and may also agree bookings for events that do not involve performers, such as exhibitions or conferences. The relationship is optional for the agent.



The entity relationship diagram above shows that it is mandatory for performers, but optional for agents to participate. This is translated into three relations below. Note that in the relation Agent-Performers, all performers are represented by an agent (or more than one agent). There are some agents in the Agent relation who do not appear in Agent-Performers because they do not represent performers.

Relation: Performers

Perf-id	Perf-name	Perf-type	Perf-Loc'n
4240	Nita Shah	Dancer	Paris
4598	Reena Chotalia	Dancer	Rome
4837	Panos Kotzias	Actor	Milan
5930	Yuen Chan	Musician	Taipei
5928	Terry Ford	Singer	Sydney
6050	Keith Buchanan	Musician	Beijing

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5490	Charles Patterson	Rome

Relation: Agent-Performers

Perf-id	Agent-id
4240	2936
4240	5386
4598	5386
4837	2936
5930	3895
5930	5386
5928	3895
6050	2936

The second possibility for this kind of relationship is that the performer entity is optional but the agent entity is mandatory. In this case, a performer might have one or more agents, but an agent must represent several performers. Here, a performer could make a booking personally, or could have a booking made by a number of different agents. The agents can only make bookings for performers, and for no other kind of event.

The entity relationship diagram above illustrates optional participation for a performer, but mandatory participation by an agent.

Relation: Performers

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5490	Charles Patterson	Rome

Relation: Agents

Perf-id	Perf-name	Perf-type	Perf-Loc'n
1403	Michael Simpson	Actor	Bombay
1906	Theo Berdekas	Singer	Athens
1974	Anil Shah	Actor	Los Angeles
1935	Kang Hae Lin	Dancer	London
1968	Suk Lo	Musician	Beijing
2027	Ruth Windsor	Actor	London

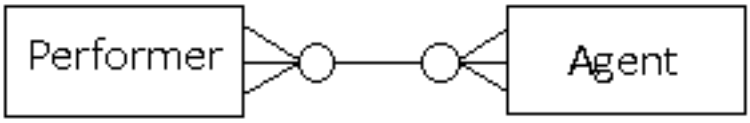
Relation: Agent-Performers

Perf-if	Agent-id
1403	2936
1403	3246
1974	5490
1935	4837
1935	3895
1968	5490
1968	5386

The relation Agent-Performers shows that all agents represent one or more performers. Some performers are represented by more than one agent, whereas other performers are not represented by agents at all.

Optional for both entities

We could imagine a situation where each performer could be represented by a number of different agents, and could also make bookings without using an agent. In addition, each agent could act for a number of different performers, and the agents could also make bookings that did not involve performers. This would be modelled by a many-to-many relationship between performers and agents that was optional for both entities.



In order to represent this relationship between two entities, we would need three relations, one for each entity and one for the relationship itself. The reason we need three relations rather than just two (one for each entity) is that the relationship is optional. This means that if we were to store the identifier of one entity in the relation of the other, there would be times when we would have a null value for the identifier as no relationship exists for a particular instance of the entity. We cannot have a null value for an identifier, and therefore we show the relationships that do exist explicitly in a third relation.

Relation: Performers

Perf-id	Perf-name	Perf-type	Perf-Loc'n
4374	Mary East	Singer	Tokyo
5495	Gordon Tripp	Actor	Sydney
6087	Donna Apinoko	Dancer	Moscow
7343	Frances Heaton	Actor	Kiev
8903	Hilary Wishart	Comedian	Los Angeles
8342	Liang Hong	Singer	New York
9475	Wing Keung Lee	Musician	Paris

Relation: Agents

Agent-id	Agent-name	Agent-Loc'n
2936	Alice Truman	London
3246	Jaimin Khetia	Nairobi
3617	Andreas Pericleous	Nicosia
3895	Steve Murphy	Cairo
4837	Angela Demetriou	Athens
5386	Priti Popat	Chicago
5421	Mei Choo Lin	Taipei
5490	Charles Patterson	Rome

Relation: Agent-Performers

Perf-id	Agent-id
4374	4837
6087	2936
6087	3617
6087	5386
7343	2936
8342	4837
8342	5490
9475	5386

Summary of conversion rules

The following table provides a summary of the guidelines for converting components of an entity-relationship diagram into relations. We need to be certain that if we store an identifier for one entity in a relation representing another entity, that the identifier never has a null value. If we have a null value for an identifier, we will never be able to find the other details that should be associated with it.

Cardinality	Membership Class	Number of Relations	Notes
1 : 1	Both Mandatory	**1**	all attributes in a single table
1 : 1	One Mandatory One Optional	**2**	Identifier of optional entity held in the mandatory entity relation
1 : 1	Both Optional	**3**	one relation for each entity and the relationship between them
1 : N	Both Mandatory	**2**	One relation for each entity and identifier of "one" end held in "many" end entity relation
1 : N	One Mandatory One Optional	**2**	*if the many end is mandatory:* one relation for each entity and identifier of the optional entity (the "one" end) held in the mandatory entity relation (the "many" end)
1 : N	One Mandatory One Optional	**3**	*if the many end is optional:* one relation for each entity and one for the relationship between them
1 : N	Both Optional	**3**	one relation for each entity and one for the relationship between them
M : N	Both Mandatory	**3**	one relation for each entity and one for the relationship between them
M : N	One Mandatory One Optional	**3**	one relation for each entity and one for the relationship between them
M : N	Both Optional	**3**	one relation for each entity and one for the relationship between them

Review questions

- Case study: Theatrical database

Consider the design of a database in the context of the theatre. From the description given below, identify the entities and the relationships that exist between them. Use this information to create an entity-relationship diagram, with optional and mandatory membership classes marked. How many entities have you found? Now translate this data model into relations (tables of data). Don't forget the guidelines in order to decide how many relations you will need to represent entities and the relationships between them. You should also think

about areas where you don't have enough information, and how you would deal with this kind of problem. You might also find that there is information that you don't need for building the data model.

"Authors are responsible for writing plays that are performed in theatres. Every time a play is performed, the author will be paid a royalty (a sum of money for each performance).

Plays are performed in a number of theatres; each theatre has maximum auditorium size, and many people attend each performance of a play. Many of the theatres have afternoon and evening performances.

Actors are booked to perform roles in the plays; agents make these bookings and take a percentage of the fee paid to the actor as commission. The roles in the plays can be classified as leading or minor roles, speaking or non-speaking, and male or female."

- Explain the difference between entities and attributes. Give examples of each.
- Distinguish between the terms 'entity type' and 'entity instance', giving examples.
- Distinguish between the terms 'primary key' and 'candidate key', giving examples.
- Explain what is meant by one-to-one, one-to-many and many-to-many relationships between entities, giving an example of each.
- How are many-to-many relationships implemented in Relational databases?

Chapter 7. Enhanced Entity-Relationship Modelling

Table of contents

Objectives

At the end of this chapter you should be able to:

- Describe the concepts of specialization/generalization, aggregation and composition.
- Illustrate how specialization/generalization, aggregation and composition are represented in ER diagrams.
- Map the specialization/generalization relationship to tables suitable for Relational database implementation.

Introduction

In parallel with this chapter, you should read Chapter 12 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter builds on the previous chapter which addressed the basic concepts of Entity-Relationship (ER) modelling. The chapter discussed the concepts of an entity, participation, recursive relationships, weak entities and strong entities. It also illustrated how these concepts can be represented in the ER diagrams. Improved computer speed and memory has, in recent years, triggered the development of sophisticated software applications like Geographical Information Systems (GIS). The basic features of ER modelling are not sufficient to represent all the concepts in such applications. To address these needs, many different semantic data models have been proposed and some of the most important semantic concepts have been successfully incorporated into the original ER model. This chapter discusses and illustrates advanced ER modelling concepts, namely specialization/generalization, aggregation and composition.

Context

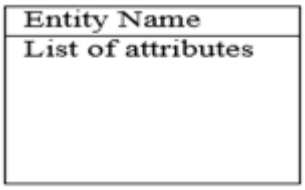
This chapter continues to address the top-down database design concepts. Like the previous chapters, it links closely with the other chapters on database design, Normalisation and other design topics. The chapter also has considerable relevance for the material in the module on performance tuning, such as the chapter on indexing, as the decisions made during database design have a major impact on the performance of the application.

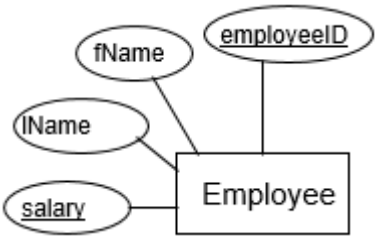
Recap on previous concepts

In the previous chapter, we discussed basic concepts of ER modelling. This sections revisits some of the important concepts covered.

Entities

An entity may represent a category of people, things, events, locations or concepts within the area under consideration. An entity can have one or more attributes or characteristics. Two notations for representing an entity are common: box notation, and the notation that employs ellipses to represent the attributes belonging to an entity.





Relationship types

These express the number of entities with which another entity can be associated via a relationship. The relationships that exist between two entities can be categorised by the following:

- one-to-one

- one-to-many

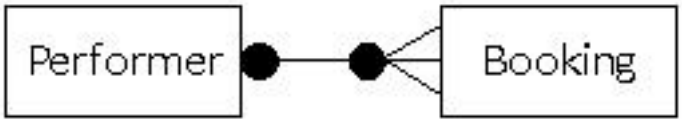
- many-to-many

Relationship participation

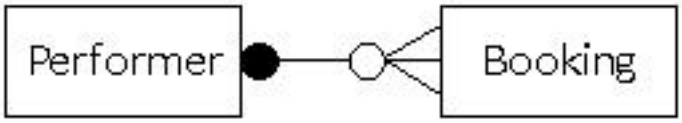
The participation condition defines whether it is mandatory or optional for an entity to participate in a relationship. This is also known as the membership class of a relationship.

There are two kinds of participation conditions: mandatory and optional. Most entities are involved in binary relationships, so it follows that there are four main types of membership relationships:

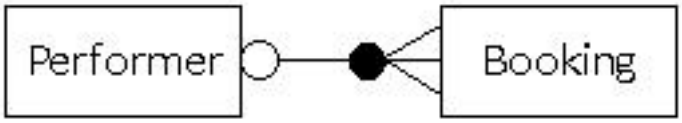
1. Mandatory for both entities



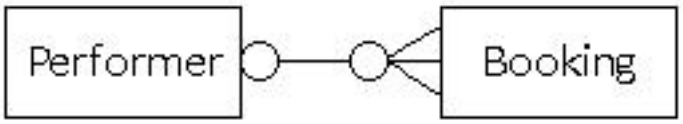
2. Mandatory for one entity, optional for the other



3. Optional for one entity, mandatory for the other



4. Optional for both entities



Note: We have used the one-to-many relationship type to illustrate participation. Refer to the previous chapter for more details on how to model participation for other relationship types.

Specialization/generalization

We have discussed different types of relationships that can occur between entities. Some entities have relationships that form a hierarchy. For example, a shipping company can have different types of ships for its business. The relationship that exists between the concept of the ship and the specific types of ships forms a hierarchy. The ship is called a superclass. The specific types of ships are called subclasses.

Superclass: An entity type that represents a general concept at a high level.

Subclass: An entity type that represents a specific concept at lower levels.

A subclass is said to inherit from a superclass. A subclass can inherit from many superclasses in the hierarchy. When a subclass inherits from one or more superclasses, it inherits all their attributes. In addition to the inherited attributes, a subclass can also define its own specific attributes. A subclass also inherits participation in the relationship sets in which its superclass (higher-level entity) participates.

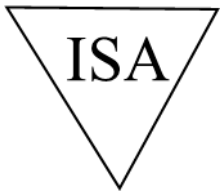
The process of making a superclass from a group of subclasses is called generalization. The process of making subclasses from a general concept is called specialization.

Specialization: A means of identifying sub-groups within an entity set which have attributes that are not shared by all the entities (top-down).

Generalization: Multiple entity sets are synthesized into a higher-level entity set, based on common features (bottom-up).

Representation of specialization/generalization in ER diagrams

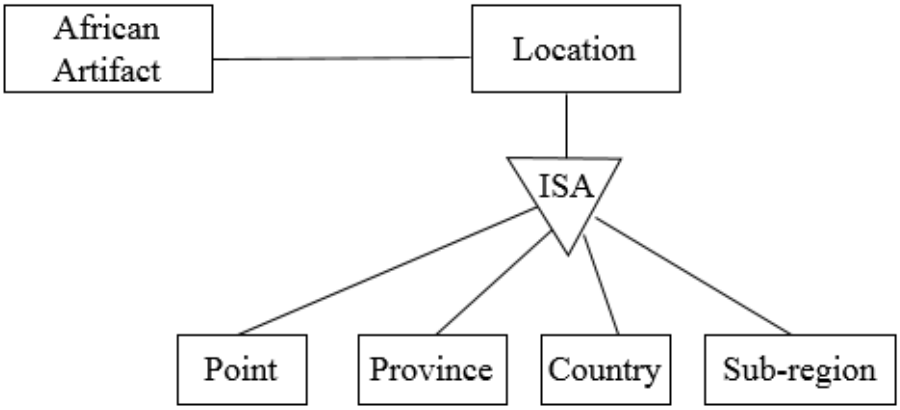
A diamond notation is a common representation of specialization/generalization relationships in ER diagrams.



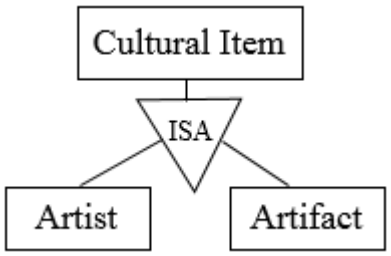
As an example, let's consider the following scenario:

Africa holds many historical artefacts in different locations. Each artefact is kept in a specific location. A location can be a point, province, country or sub-region of Africa.

The scenario has a specialization relationship between the location and different specific types of locations (i.e. point, province, country and sub-region). This specialization relationship is represented in the ER diagram below.



To demonstrate generalization, let's imagine that an Artefact is one of the examples of the African cultural items. Another type of a cultural item is an Artist. It is clear to see that a cultural item is a superclass of an artefact and artist. This generalization relationship can be represented in the ER diagram as show below.



Constraints on specialization/generalization

There are three constraints that may apply to a specialization/generalization: membership constraints, disjoint constraints and completeness constraints.

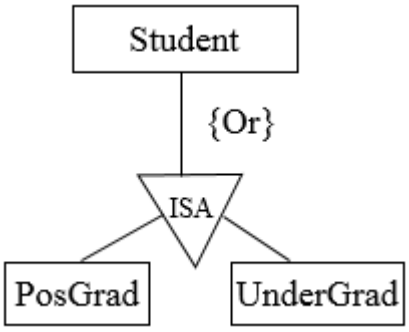
- **Membership constraints**

  Condition defined: Membership of a specialization/generalization relationship can be defined as a condition in the requirements e.g. tanker is a ship where cargo = "oil"

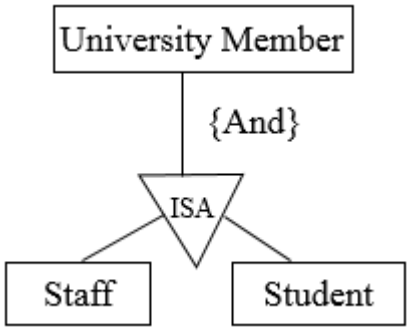
  User defined: Sometimes the designer can define the superclass-subclass relationship. This can be done to simplify the design model or represent a complex relationship that exists between entities.

- **Disjoint constraints**

  Disjoint: The disjoint constraint only applies when a superclass has more than one subclass. If the subclasses are disjoint, then an entity occurrence can be a member of only one of the subclasses, e.g. postgrads or undergrads – you cannot be both. To represent a disjoint superclass/subclass relationship, 'Or' is used.

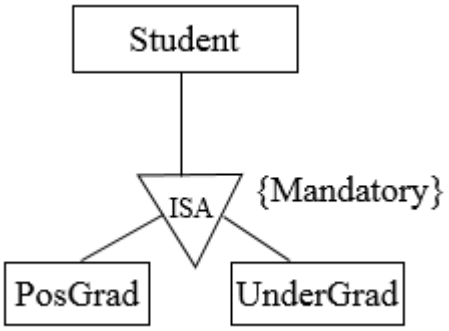


Overlapping: This applies when an entity occurrence may be a member of more than one subclass, e.g. student and staff – some people are both. 'And' is used to represent the overlapping specialization/generalization relationship in the ER diagram.

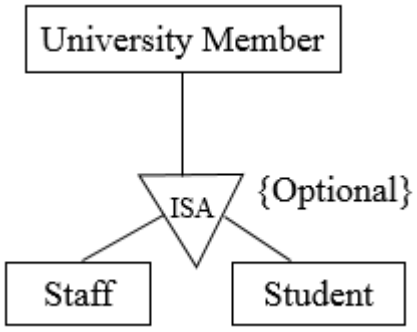


- **Completeness constraints**

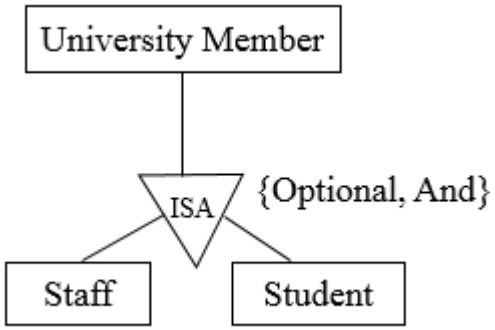
  Total: Each superclass (higher-level entity) must belong to subclasses (lower-level entity sets), e.g. a student must be postgrad or undergrad. To represent completeness in the specialization/generalization relationship, the keyword 'Mandatory' is used.



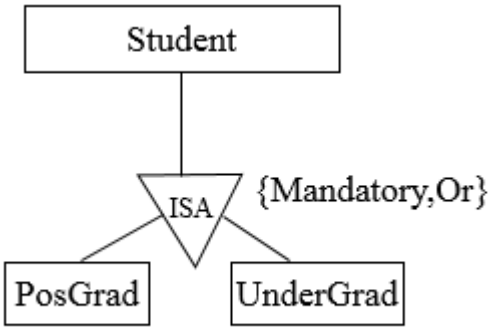
  Partial: Some superclasses may not belong to subclasses (lower-level entity sets), e.g. some people at UCT are neither student nor staff. The keyword 'Optional' is used to represent a partial specialization/generalization relationship.



We can show both disjoint and completeness constraints in the ER diagram. Following our examples, we can combine disjoint and completeness constraints.



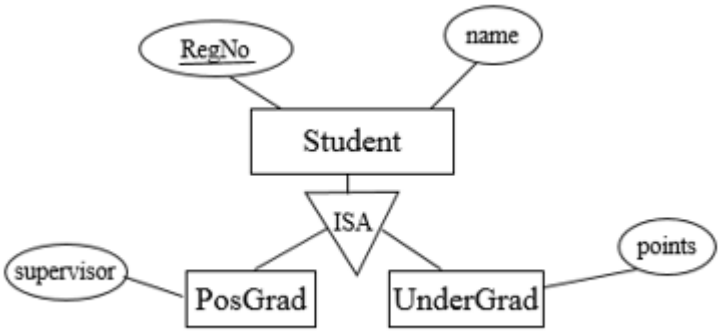
Some members of a university are both students and staff. Not all members of the university are staff and students.



A student in the university must be either an undergraduate or postgraduate, but not both.

Mapping specialization/generalization to relational tables

Specialization/generalization relationship can be mapped to relational tables in three methods. To demonstrate the methods, we will take the student, postgraduate and undergraduate relationship. A student in the university has a registration number and a name. Only postgraduate students have supervisors. Undergraduates accumulates points through their coursework.



Method 1

All the entities in the relationship are mapped to individual tables.

Student (*Regno*, name)

PosGrad (*Regno*, supervisor)

UnderGrad (*Regno*, points)

Method 2

Only subclasses are mapped to tables. The attributes in the superclass are duplicated in all subclasses.

PosGrad (*Regno*, name, supervisor)

UnderGrad (*Regno*, name, points)

This method is most preferred when inheritance is disjoint and complete, e.g. every student is either PosGrad or UnderGrad and nobody is both.

Method 3

Only the superclass is mapped to a table. The attributes in the subclasses are taken to the superclass.

Student (*Regno*, name, supervisor, points)

This method will introduce null values. When we insert an undergraduate record in the table, the supervisor column value will be null. In the same way, when we insert a postgraduate record in the table, the points value will be null.

Review question 1

Discuss the specialization/generalization relationship in ER modelling.

Review question 2

Explain the three constraints that can be applied on the specialization/generalization relationship.

Aggregation

Aggregation represents a 'has-a' relationship between entity types, where one represents the 'whole' and the other the 'part'.

An example of aggregation is the Car and Engine entities. A car is made up of an engine. The car is the whole and the engine is the part. Aggregation does not represent strong ownership. This means, a part can exist on its own without the whole. There is no stronger ownership between a car and the engine. An engine of a car can be moved to another car.

Representation of aggregation in ER diagrams

A line with a diamond at the end is used to represent aggregation.

The 'whole' part must be put at the end of the diamond. For example, the Car-Engine relationship would be represented as shown below:

Composition

Composition is a form of aggregation that represents an association between entities, where there is a strong ownership between the 'whole' and the 'part'. For example, a tree and a branch have a composition relationship. A branch is 'part' of a 'whole' tree - we cannot cut the branch and add it to another tree.

Representation of composition in ER diagrams

A line with a filled diamond at the end is used to represent composition.

The example of the Tree-Branch relationship can be represented as shown below:

Review question 3

Using an example, explain the concepts of aggregation and composition.

Exercise 1

Draw the ER diagram for a small database for a bookstore. The database will store information about books for sale. Each book has an ISBN, title, price and short description. Each book is published by a publisher in a certain publishing year. For each publisher, the database maintains the name, address and phone number.

Each book is written by one or more authors. For each author, the database maintains his/her ID, name and a short introduction. Each book is stored in exactly one warehouse with a particular quantity. For each warehouse, the database maintains the warehouse name, the location and the phone number. Each book has one or more sellers, which may be either companies (corporate vendors) or individuals (individual vendors).

For each company, the database maintains a name of the company, its address, its phone numbers (there could be more than one phone number, each with a number and a description) and its contact person. For each individual vendor, the database keeps a name, a phone number and an email address. A contact person whose company sells a book cannot be selling the same book as an individual vendor at the same time (he/she may sell other books as an individual seller).

Additional content - XML

What is XML?

In previous chapters, we introduced database technology and how it is used by businesses to store data in a structured format. XML (eXtensible Markup

Language) has become a standard for structured data interchange among businesses. It was formally ratified by the World Wide Web Consortium (W3C) in 1998. XML uses markup for formatting plain text. Markup refers to auxiliary information (tags) in the text that give structure and meaning.

We have demonstrated how to use relational tables to represent entities and their attributes. XML also supports the representation of entities and attributes.

In this section, we will introduce XML. Students are encouraged to study detailed books for further information. One useful website for learning XML is http://www.w3schools.com/xml/default.asp.

Element

An element is a building block of an XML document.

- All elements are delimited by < and >.
- Element names are case-sensitive and cannot contain spaces.

The representation of an element is shown below:

<Element> …. </Element>

An XML document can contain many elements, but one must be the root element. A root element is a parent element of all other elements.

```
<root>
 <child>
  <subchild>.....</subchild>
 </child>
</root>
```

Attribute

Elements can have attributes. Attributes are specified by name=value pairs inside the starting tag of an element:

<Element attribute = “value” >.. </Element >

All values of the attributes are enclosed in double quotes.

An element can have several attributes, but each attribute name can only occur once.

<Element attribute1 = “value1” attribute2=“value2”>

Example representing relational table records in XML

To demonstrate XML, let’s imagine we have a customer table that holds information of customers.

CUSTOMER_ID	NAME	LOCATION
100078	Doris	Mowbray
200009	Cindy	Fish Hoek
899900	Neo	Retreat

We can represent the information in XML as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Customers>
        <Customer customerID=”100078”>
                <Name> Doris <Name>
                <Location> Mowbray </Location>
        </Customer>
        <Customer customerID=”200009”>
                <Name> Cindy <Name>
                <Location> Fish Hoek </Location>
        </Customer>
        <Customer customerID=”899900”>
                <Name> Noe <Name>
                <Location> Retreat </Location>
        </Customer>
</Customers>
```

Explanation

- **<?xml version=“1.0” encoding=“UTF-8”?>:** is the XML prolog. The prolog is used to specify the version of XML and the encoding used. It is optional, but if it appears in the document, it must be the first line in the document.
- **Customers element:** Customers is the root element.
- **Customer element:** A Customer element represents a tuple in the Customers table. The table has three attributes, CUSTOMER_ID, NAME and LOCATION. In our XML, CUSTOMER_ID is represented as an

attribute of the Customer element. NAME and LOCATION are represented as child elements of the Customer element. Notice that we have repeated the Customer element three times to capture the three records in the Customer table.

Document type definition

The XML technology specifies the syntax for writing well-formed documents but does not impose the structure of the document. XML document writers are free to structure an XML document in any way they want. This can be problematic when verifying a document. How many elements can a document have? What elements should a document have? These questions are difficult to answer unless we also specify the structure of the document. Document type definition (DTD) is used to define the structure of an XML document.

DTD specifies the following:

- What elements can occur.
- What attributes an element can/must have.
- What sub-elements can/must occur inside each element, and how many times.

DTD element syntax:

<!ELEMENT element (subelements-specification) >

DTD attribute syntax:

<!ATTLIST element (attributes) >

The DTD for the XML we defined above can be defined as shown below:

```
<!DOCTYPE Customers [
  <!ELEMENT Customers (Customer+)>
  <!ELEMENT Customer (Name, Location)>
  <!ELEMENT Name(#PCDATA)>
  <!ELEMENT Location(#PCDATA)>
  <!ATTLIST Customers customerID CDATA>
  ]>
```

Explanation

- **!DOCTYPE:** Defines that the Customers element is the root element of the document.
- **<IELEMENT>:** Defines an XML element. The first element to be defined is the Customers element. A Customers element has one child element, Customer, indicated in brackets. The + symbol means that the Customer element can appear one or more times under the Customers element. The Customer has two sub-elements, Name and Location. The Name and Location elements have character data as a child element.
- **<!ATTLIST>:** Defines the attribute. The Customers element has one attribute, customerID, of type character data.

Namespaces

XML data has to be exchanged between organisations. The same element name may have different meaning in different organisations, causing confusion on exchanged documents.

Specifying a unique string as an element name avoids confusion. A better solution is to use a unique name followed by an element name.

unique-name:element-name

Adding a unique name to all element names can be cumbersome for long documents. To avoid using long unique names all over a document, we can use XML namespaces.

```
<sailing xmlns:FB='http://www.FancyBoats.com'>
    ...
    <FB:boat>
        <FB:boatname> Guppie </FB:boatname>
        <FB:location> Fish Hoek</FB:location>
    </FB:boat>
    ...
</sailing>
```

The namespace FB has been declared and initialised to 'http://www.FancyBoats.com'. Namespaces are URIs. URIs are generic identifiers like URLs.

XQuery

XQuery is a language for finding and extracting elements and attributes from XML documents. The way SQL is to relational databases, XQuery is the query language for XML documents. For example, to display all the names of the customers in the XML above, our XQuery will look as follows:

for $x in /Customers/Customer

return $x/Name

Exercise 2

In chapter 3, Introduction to SQL, we introduced the EMP table. Represent the records in the table in XML.

Chapter 8. Data Normalisation

Table of contents

Objectives

At the end of this chapter you should be able to:

- Describe the process, strengths and weaknesses of data normalisation, and demonstrate an understanding of when and to what extent the technique should be applied in practice.
- Explain and apply the concepts of functional dependency and determinants through the understanding and construction of determinacy diagrams.

- Describe and apply understanding of three normal forms for relations.
- Convert un-normalised data into first normal form relations, so that data items contain only single, simple values.
- Derive second normal form relations by eliminating part-key dependencies.
- Derive third normal form relations by removing transitive dependencies.

Introduction

In parallel with this chapter, you should read Chapter 13 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

Normalisation stands on its own as a well-founded approach to database design. In addition, normalisation links closely with the material covered in the previous two chapters on entity-relationship modelling. However, the additional flexibility of normalised designs comes at a price — a well-normalised design tends to perform poorly when subjected to large volumes of transactions. For this reason, there are trade-offs to be made between the extent to which a design is normalised and the performance response of the implemented system. The information in this chapter has to be applied carefully, in light of the information given in a later chapter on database design relating to de-normalisation and physical design.

Why should we attempt to normalise data? Un-normalised data often contains undesirable redundancy (and its associated 'costs' in storage, time and multiple updates), and different degrees of normalisation (i.e. different normal forms) can guarantee that certain creation, update and deletion anomalies can be avoided.

Context

This chapter covers the well-known approach to database design known as data normalisation. It introduces a bottom-up technique for the development of flexible database applications. This bottom-up approach complements the top-down entity-relationship technique presented in the first database design chapter, as the two approaches can be used to cross-check the extent to which the overall design satisfies the requirements of the application. By themselves, database designs arrived at through the normalisation process, while providing great flexibility, tend to perform very slowly. The complementary bottom-up and top-down methodologies, in practice, often reveal different information, and can be applied using different fact-finding techniques. For these reasons (of efficiency and the benefits of multiple viewpoints to get a better final design), a balanced approach to database design will use both approaches.

Determinacy diagrams

Determinants and determinacy diagrams

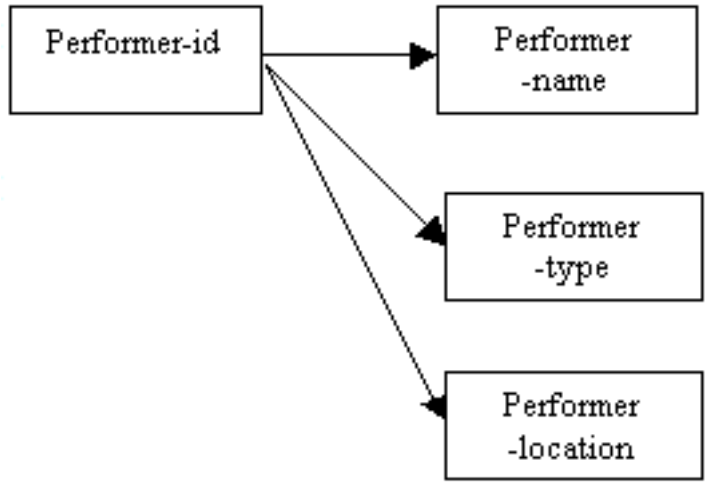
Diagrams can be used to indicate the dependencies between different attributes of an entity. We saw in the earlier chapter on entity-relationship modelling that one or more attributes could be identified as candidate keys before making a final selection of a primary key. When a primary key has been chosen, we may find that some attributes do not depend on the key, or some attributes depend only on part of the key. Determinacy diagrams offer the opportunity to examine the dependencies between attributes and the primary key in a visual representation.

Important

Determinant

When the value of one attribute allows us to identify the value of another attribute in the same relation, this first attribute is called a determinant. The determinant of a value might not be the primary key. This is true for groups of attributes as well, so if A is the determinant of B, A and B may either be single attributes, or more than one attribute.

In the diagram below, it can be seen that the name of a performer depends entirely on the performer-id (we know that this is a one-to-one relationship). We can say that performer-id functionally determines the performer-name, and this is shown by the arrow. In addition, the type and location of any particular performer are also determined by the performer-id.



It might be the case that there are performers who share the same family name

(for example, a family of actors). Each member of the family who is an actor will have a unique performer-id (as the attribute performer-id is the primary key), but there may be more than one person with that particular name. The performer-name would not make a suitable choice for primary key for this reason. The performer-id uniquely determines the performer-name, but a performer-name may indicate more than one performer-id.

In a similar way, there may be more than one performer of a particular type; the performer-id will identify the performer-type for that specific individual. It is likely that any one location may have more than one performer based there; the location of any particular performer can be determined by means of the performer-id as the primary key. There are several possibilities for considering how the fee to a performer for a booking at a venue might be calculated, and these might include:

- flat rate fee for all performers for all venues
- fee negotiated with performer
- fee depends on performer's location
- fee depends on location of venue
- fee depends on performer type
- fee depends on date of booking
- fee depends on a combination of factors (e.g. performer and agent)

The method by which the fee is calculated will affect the way the data is modelled; this is because the value of the fee can be linked to a number of other attributes, and might not be determined by the performer-id alone as the primary key. The determinacy diagrams may be different depending on the particular method of calculating the fee.

If we consider some of the possibilities outlined above, we can identify the dependencies that affect the fee and create a determinacy diagram.

Direct dependencies

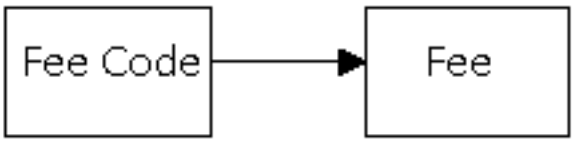
An example to illustrate direct dependencies might be: flat rate fee for all performers for all venues.

In this case, the fee could be regarded as another attribute of each performer, or could be linked to a performance (the number of performances determining the total amount earned). The fee could be regarded as an entity in its own right. We would need to take into account what would happen if the fees were to change. Would all fees change to the same new value? What would determine whether one performer earned a different fee from another? The answers to these questions would reveal underlying dependencies between the data.

If we assume that all performers are paid the same fee, and when the fee is changed it affects all performers in exactly the same way, we can identify the fee as a separate entity.

The value of the fee would then depend on the fee code. The fee is directly dependent on the fee code.

(Note that we would not want to insert the exact fee as a value for all performers because of the implications of updating the value when the fee changes.)



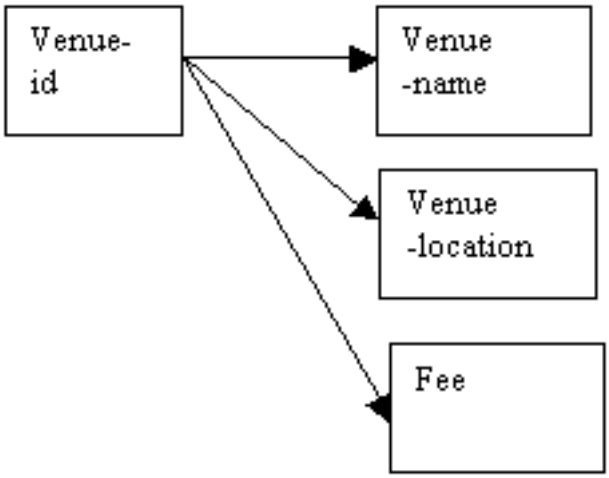
Transitive (indirect) dependencies

An example to illustrate transitive (also known as indirect) dependencies might be: fee depends on location of venue.

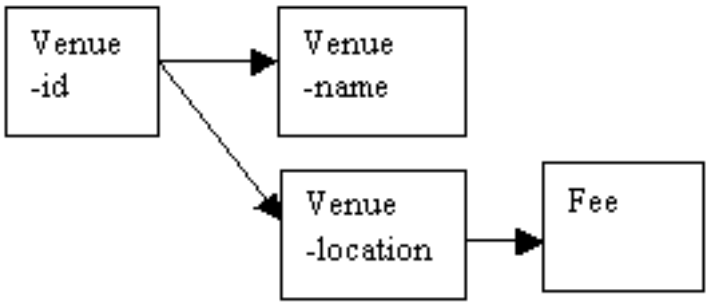
Where the value of the fee depends on the location of the venue, it is not possible to decide in advance what fee will be paid to a performer until details of the venue are known. This means that a booking must be made by an agent for a performer at a venue in order for the fee to be determined.

It will be necessary to find out whether the fee is determined by the specific venue, or whether all venues in the same location also attract the same fee.

If each venue has its own fee, then the fee will be determined by the venue-id, in the same way that other attributes of a particular venue, such as the name and location, are identified by venue-id as the key. This is a direct dependency.



On the other hand, if the fee applies to all venues in the same area, venues must be identified as belonging to specific areas in which a given fee applies. This is an indirect dependency, also known as a transitive dependency.



Important

Transitive (indirect) dependency

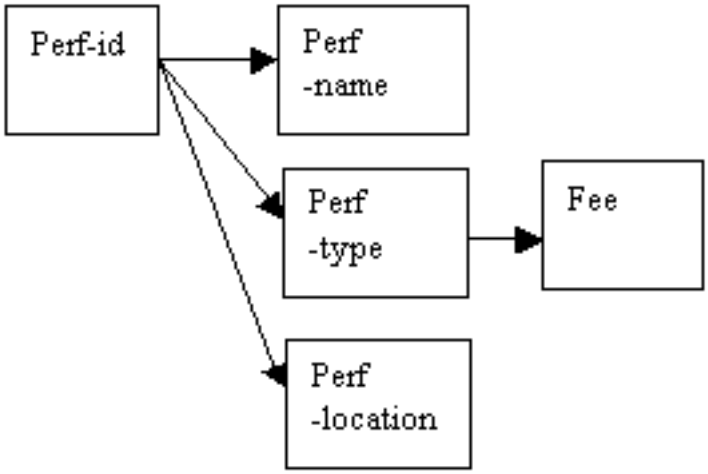
Sometimes the value of an attribute is not determined directly from the primary key, but through the value of another attribute which is determined by the primary key; this relationship is known as a transitive dependency.

Another example of a transitive dependency

Consider the following attributes: fee depends on performer type.

Here the fee depends on whether the performer is an actor, dancer, singer or some other type of performer. The different types of performer need to be

identified, and a fee specified in each case. The value of the fee does not depend directly on the performer-id, but is linked to the type of performer. This is another example of an indirect (or transitive) dependency.



Composite determinants and partial dependencies

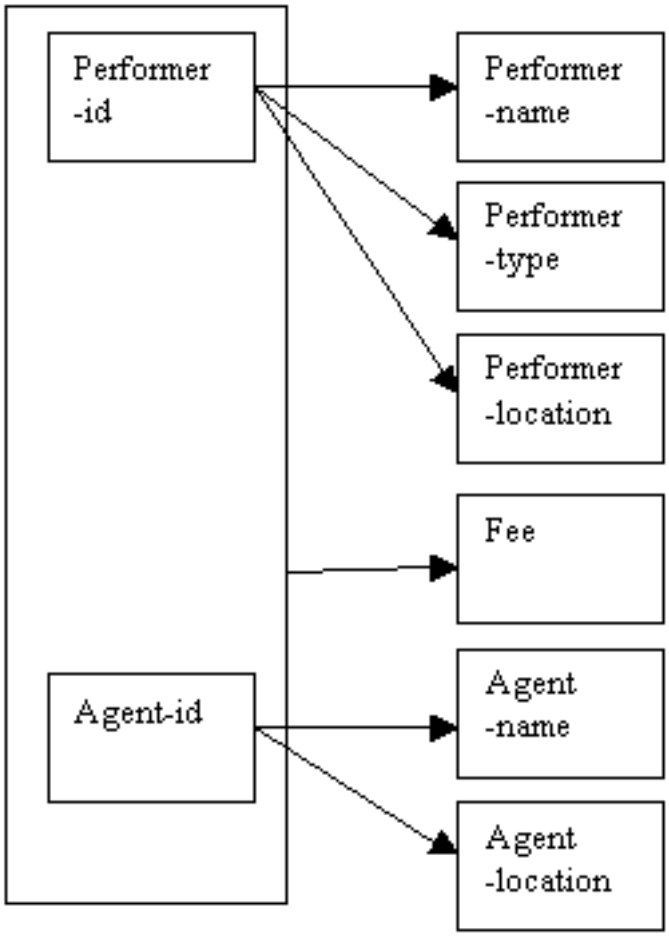
Sometimes the determinant is not a single attribute, but made up of two or more attributes. Consider the following: fee depends on a combination of factors (e.g. performer and agent).

Important

Composite determinant

If more than one value is required to determine the value of another attribute, the combination of values is known as a composite determinant.

If the fee is determined by more than one factor, both these elements must be taken into account. This is shown in the determinacy diagram on the right by the arrow including both the performer-id and the agent-id as the determinant items on which the fee depends. The attributes performer-id and agent-id are known as composite determinants.



Where every attribute in a primary key is required as a composite determinant for an attribute, the attribute is said to be fully functionally dependent on the key.

Note that the attributes that depend only on performer-id (such as the name, type and location of each performer), or agent-id (such as the agent and location of each agent) are shown linked directly to the appropriate key. If we take performer-id and agent-id as the key, we can say that the performer and agent details are partially dependent on the key. Partial dependency is when an attribute is functionally dependent on a proper subset of the key.

Important

Partial dependency

If the value of an attribute does not depend on an entire composite determinant, but only part of it, that relationship is known as a partial dependency.

Multiple determinants

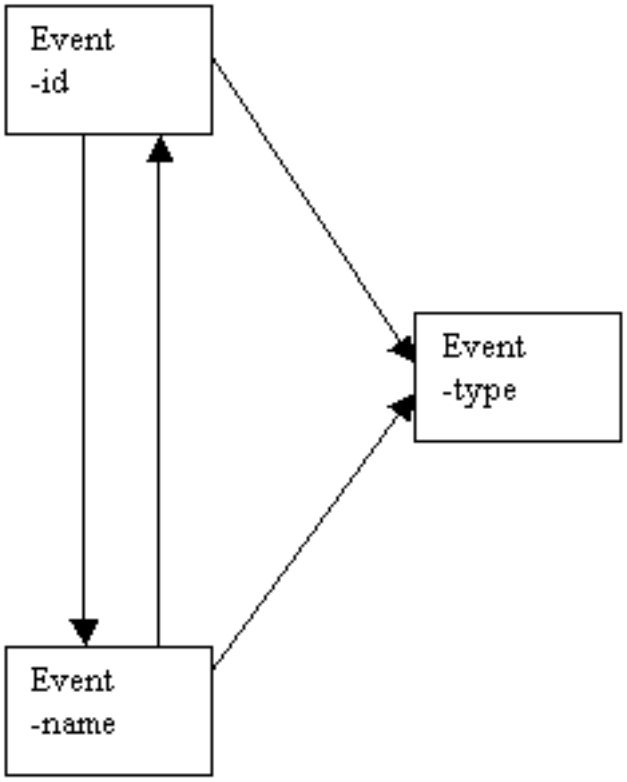
It is possible that there may be more than one attribute that can act as a determinant for other attributes. This is a slightly different situation from that of candidate keys, as not all determinants are necessarily candidate keys. If we wish to describe an event, we may find that there is a special relationship between the attributes event-id and event-name; each event will have a unique identification number, and also an unique name. The relationship between the event-id and the event-name is one-to-one. The better choice of primary key for the event would be event-id, which is a unique identification number.

The attribute event-name, while unique to each event, would not make such a good choice as the key because there can be problems in obtaining an exact match (e.g. “Quicktime”, “Quick time” and “Quick Time” would be regarded as different names).

We can show dependencies between the attributes event-id, event-name and event-type on a determinacy diagram.

Each event would have values for the attributes event-id, event-name and event-type.

In the determinacy diagram below, we can see that event-id is a determinant for the other two attributes, event-name and event-type.

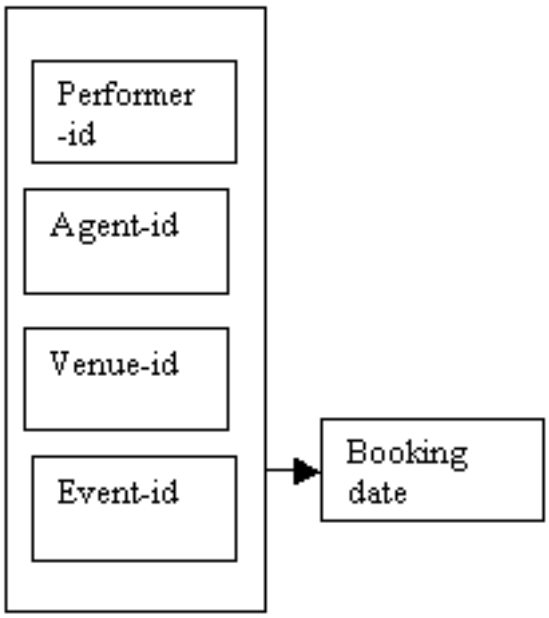


The determinacy diagram shows that the attribute event-name is also a determinant for the other two attributes, event-id and event-type. This is because there is a one-to-one relationship between event-id and event-name.

Overlapping determinants

There are sometimes cases where there is more than one combination of attributes that uniquely identifies a particular record. This means that the determinants have attributes in common. In certain circumstances, there may be a special relationship between the attributes, so that each uniquely determines the value of the other.

An example of this may be where each module in a degree programme has a unique module code and a unique module name. It would be possible to use either the module code or the module name as the determinant. In addition, the module code determines the module name, and the module name determines the module code.



In the context of our example relating to performers, agents, venues and events, we will also need to be able to identify bookings. We find that each booking can be identified by a different combination of attributes.

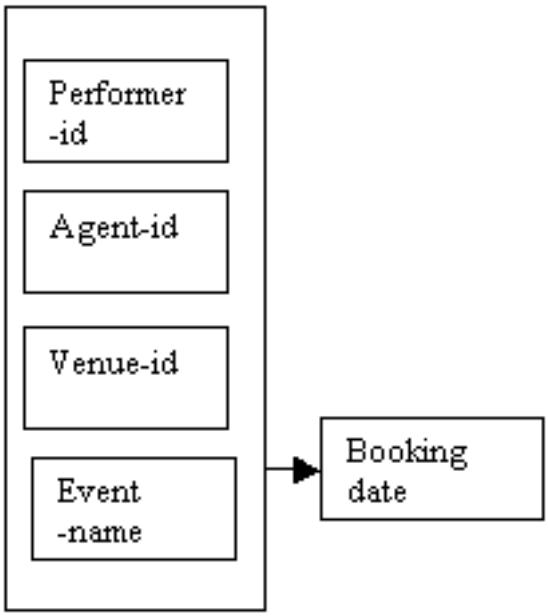
When a booking is made, the performer-id, agent-id, venue-id and event-id are all required in order to specify a particular event occurring on a given date. This also needs to be represented using a determinacy diagram.

Each booking can be identified by the primary key, which is shown on the right as a combination of the attributes performer-id, agent-id, venue-id and event-id.

Note that in this instance, the arrow (coming from the outer box) indicates that all four key attributes are used to identify the booking date.

We know that each event can be identified either by the event-id or the event-name; this means that we could have an alternative representation in the determinacy diagram, substituting the attribute event-name for event-id as part of the combined key.

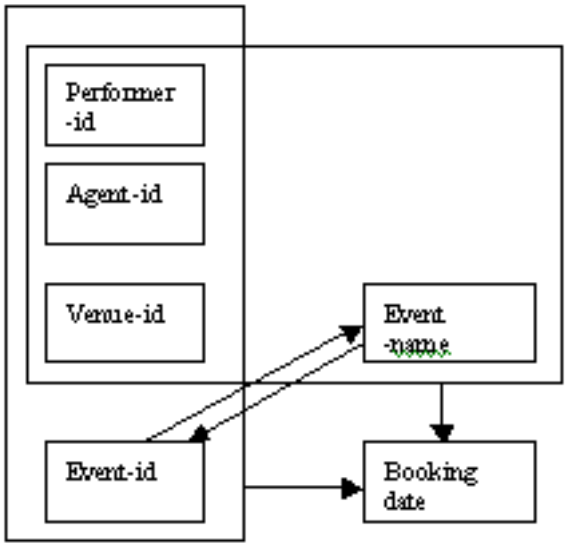
An alternative primary key for each booking would be a combination of performer-id, agent-id, venue-id and event-name.



Here again, the arrow (coming from the outer box) indicates that all four key attributes are used to identify the booking date.

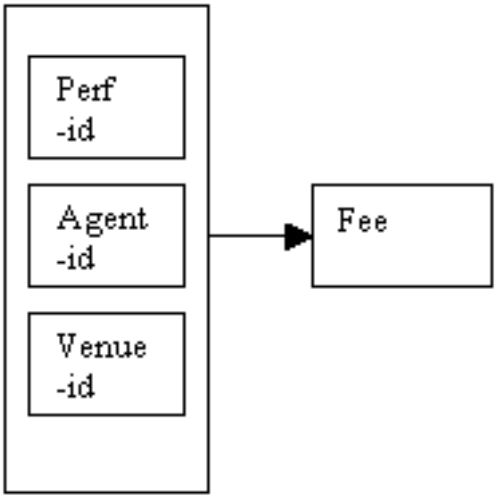
Here we have an overlapping key. The attribute event-name is a determinant, although it is not a candidate key for its own data. We would not want to use the event-name as a primary key, as it can present a problem in identifying the relevant tuple if the spelling is not exactly the same as in the relation.

The determinacy diagram also shows the relationship between the attributes event-id and event-name.



Exploring the determinant of 'fee' further

Consider the following determinacy diagram for attribute 'fee':

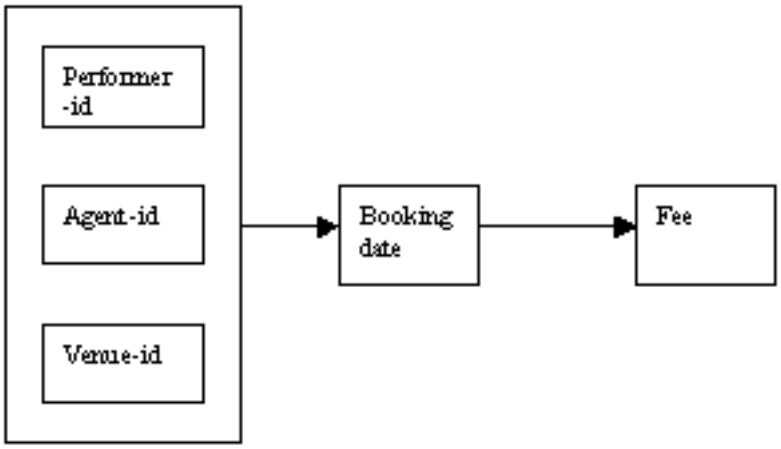


If a performer negotiates the same fee for all bookings, the fee depends on the performer-id, as each performer will have their own fee. This is a direct dependency.

Where the value of the fee depends the date of the booking, the value of the fee cannot be known until details of the booking are available.

This means that a booking must be made by an agent for a performer at a venue in order for the fee to be determined. It may be that a higher fee is paid in the summer months than at other times of the year.

The booking date will be determined by the composite determinant made up from the agent-id, performer-id and venue-id (as all three are involved). The booking date itself then determines the fee. There is therefore an indirect (or transitive) dependency between the composite key and the fee.



Finding keys using functional dependency

Functional dependency (FDs) helps to find keys for relations. To identify all candidate keys, check whether each determinant uniquely identifies tuples in the relation. Let's define another important concept called attribute closure.

Attribute closure

The closure of X, written X+, is all the attributes functionally determined by X. That is, X+ gives all the values that follow uniquely from X. Attribute closure is used to find keys and to see if a functional dependency is true or false.

To find the closure of X+, follow the following steps:

- ans = X
- For every Y→Z such that Y ans, add Z to ans
- Repeat until no more changes to X+ are possible

For example, given a relation R, such that

R(S, C, P, M, G, L, T)

FDs {SC → PMG, SL → C, CT → L, TL → C, SP → C

Can we answer the following two questions?

Is SL a key for R?

- Start with ans = {SL}
- Using 2nd FD, SL functionally determines C, so we add C to the ans, ans = {SLC}
- Using 1st FD, SC functionally determines PMG, so we add PMG to the ans, ans = {SLCPMG}
- No more attributes can be added because no subset of the ans functionally determines other attributes, so (SL)+ is SLCPMG

Is SL a key for R? No, because the closure of SL is not equal to all the attributes in R

Does SL → PG?

Yes, because PG is in (SL)+

Normalisation

In the context of databases, normalisation is a process that ensures the data is structured in such a way that attributes are grouped with the primary key that provides unique identification. This means that some attributes, which may not depend directly on the primary key, may be extracted to form a new relation.

There are a number of reasons for performing normalisation; normalised data is resilient against anomalies that may occur in updating values by insertion, amendment or deletion, and other inconsistencies, and makes better use of storage space.

The process of normalisation does not alter the values associated with the attributes of an entity; rather, it develops a structure based upon the logical connections and linkages that exist between the data.

Important

Normalisation

When a solution to a database problem is required, normalisation is the process which is used to ensure that data is structured in a logical and robust format. The most common transformations are from un-normalised data, through first and second, to third normal form. More advanced transformations are possible, including Boyce-Codd, fourth and fifth normal forms.

If we consider the data before it has undergone the normalisation process, we regard it as un-normalised.

Un-normalised data

In the table below we have details of performers, their agents, performance venues and booking dates in an un-normalised format. In this particular example, the fee paid to the performer depends on the performer-type (for example, the fee to all actors is 85).

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
101	Baron	Singer	75	York	1295	Burton	Luton	59	Atlas	Tokyo	959	Show Time	Musical	25-Nov-1999
											907	Elgar 1	Concert	
105	Steed	Dancer	60	Berlin	1435	Nunn	Boston	35	Polis	Athens	921	Silver Shoe	Ballet	07-Jan-2002
					1504	Lee	Taipei	54	Nation	Lisbon	942	White Lace	Ballet	10-Feb-2002
108	Jones	Actor	85	Bombay	1682	Tsang	Beijing	79	Festive	Rome	901	The Dark	Drama	29-Jul-2003
											913	What Now?	Drama	
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
					1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
					1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999
118	Markov	Dancer	60	Moscow										
											934	Angels	Opera	
126	Stokes	Comed -ian	90	Athens	1509	Patel	York	59	Atlas	Tokyo	945	Trick-Treat	Variety show	02-Sep-2001
129	Chong	Actor	85	Beijing	1478	Burns	Leeds	79	Festive	Rome	926	Next Year	Drama	22-Jun-2000
											938	New Dawn	Drama	
134	Brass	Singer	75	London	1504	Lee	Taipei	28	Gratton	Boston	981	Birdsong	Musical	18-Sep-2001
					1377	Webb	Sydney							

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
138	Ng	Singer	75	Penang	1509	Patel	York	84	State	Kiev	957	Quicktime	Musical	18-Aug-1999
								82	Tower	Lima				
140	Strong	Magic -ian	72	Rome	1478	Burns	Leeds	17	Silbury	Tunis	963	Vanish!	Magic show	18-Aug-1999
								92	Palace	Milan				
					1190	Patel	Hue							
								62	Shaw	Oxford				
141	Gomez	Music -ian	92	Lisbon	1478	Burns	Leeds	84	State	Kiev	941	Mahler 1	Concert	21-Jul-2000
											964	The Friends	Drama	
					1802	Chapel	Bristol							
143	Tan	Singer	75	Chicago	1504	Lee	Taipei	79	Festive	Rome	927	Chanson	Opera	21-Nov-2002
											971	Card Trick	Magic show	
147	Qureshi	Actor	85	London	1076	Eccles	Oxford	17	Silbury	Tunis	952	Gold Days	Drama	30-Apr-2000
					1409	Arkley	York	79	Festive	Rome	988	Secret Tape	Drama	17-Apr-2000
149	Tan	Actor	85	Taipei										
150	Pointer	Magic -ian	72	Paris										
152	Peel	Dancer	60	London	1428	Vernon	Cairo	59	Atlas	Tokyo	978	Swift Step	Dance	01-Oct-2001

To accommodate the size of the table, some headings have been shortened as shown below:

- P-id: performer-id
- Perf-name: performer-name
- Perf-type: performer-type
- Perf-Loc'n: performer-location
- A-id: agent-id
- Agent-Loc'n: agent-location
- V-id: venue-id
- Venue-Loc'n: venue-location
- E-id: event-id

Problems with un-normalised data

We can see from the table that some performers have more than one booking, whereas others have only a single booking, and some have none at all.

It is also shown in the table that agents are able to make bookings for different performers at different venues, but some agents have made no bookings, some venues have not been booked, and some events have not been scheduled.

The content of the table means that there is an inconsistent format, with multiple values for agents and venues associated with a single entry for some performers. The table as it stands would not be suitable for direct conversion into a relation.

Multiple venue bookings for Eagles

The performer Eagles (performer-id 112) has bookings at more than one venue, giving multiple rather than single entries for venue details.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
					1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
					1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999

More than one venue for performer 112 Eagles

Multiple agent bookings for Eagles

The performer Eagles (performer-id 112) has bookings made by more than one agent, and therefore there are multiple entries for agent details, rather than a single entry.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
					1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
					1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999

More than one agent for performer 112 Eagles

Multiple event details for Eagles

The performer Eagles (performer-id 112) has bookings for more than one event, so that there are multiple entries for event details, rather than just one entry.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
					1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
					1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999

More than one event for performer 112 Eagles

Translating the table of un-normalised data into a relation, in what is called first normal form, will mean that the data contained in the table in represented in a more structured way. A relation in first normal form has only single entries for each attribute for every tuple. We shall now investigate how to perform this translation.

First normal form

The initial stage in the normalisation process is to convert a table of un-normalised data into a relation in first normal form. This means that we must extract the repeating groups of data that may appear in some rows of the table, and replace them with tuples where each attribute has only one value associated with it (at most).

Important

First normal form (1NF)

A relation is in first normal form if there is only one value at the intersection of each row and column.

Repeating groups in an un-normalised table of data are converted to first normal form by replacing them with tuples where each attribute has a single entry.

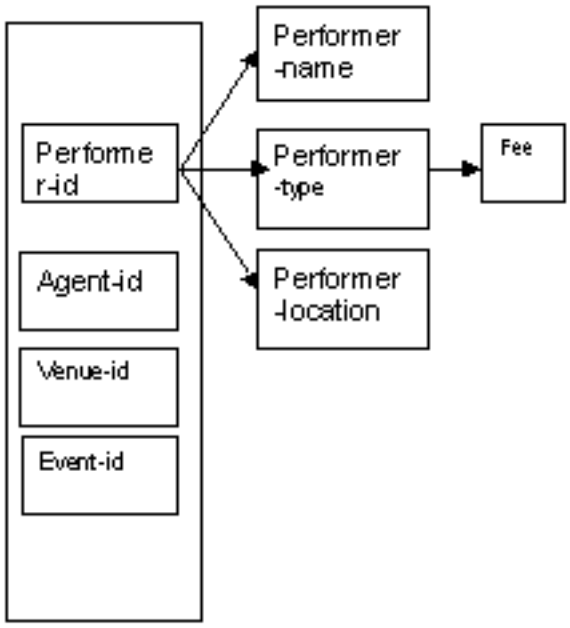
In order to convert an un-normalised relation into first normal form, we must identify the key attribute(s) involved. We can see from the table of un-normalised data that each performer has a code (performer-id), each agent is identified by an agent-id, each venue is determined by a venue-id and each event has an event-id.

Performer details

The details associated with each performer depend on the performer-id as the primary key.

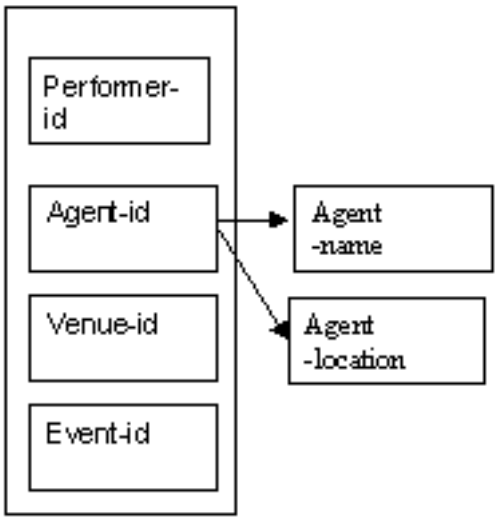
Note that the arrows coming directly from performer-id indicate that the performer attributes depend only on the key attribute performer-id, and not agent-id, venue-id or event-id.

We know that the fee in this case depends on the type of performer, and not directly on the primary key. This is shown in the diagram by the link between performer-type and fee.



Agent details

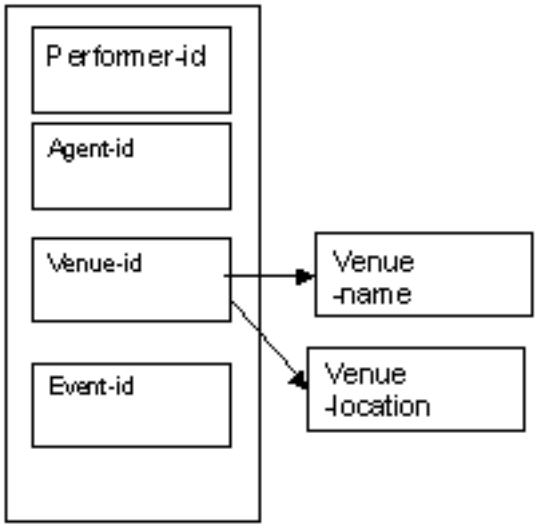
The information about each agent depends on the agent-id as the primary key.



Note that the arrow from agent-id indicates that the agent attributes depend only on agent-id as the key attribute, and not performer-id, venue-id or event-id.

Venue details

The primary key, venue-id, determines the name and location of each venue.



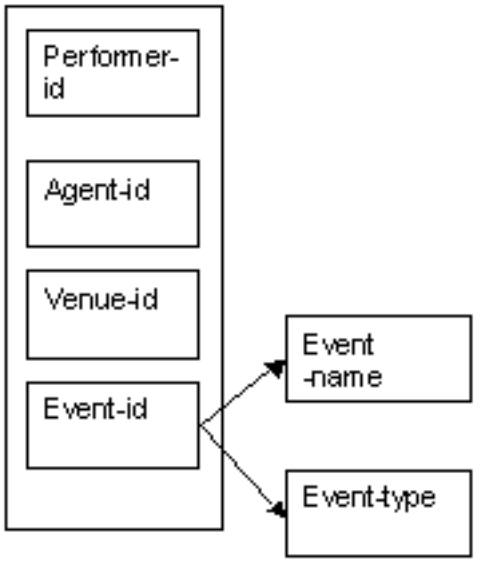
Note that the venue-name depends only on the venue-id as shown by the arrow in the diagram. The attributes performer-id, agent-id and event-id do not determine the venue-name.

Event details

We can consider the representation of events from two angles. We have two attributes which can be used as determinants: event-id and event-name. We can examine each in turn using a determinacy diagram, and then show the relationships between all three attributes (event-id, event-name and event-type) on a single determinacy diagram.

Event-id as the determinant

The primary key, event-id, determines the name and type of each event. There is a one-to-one relationship between event-id and event-name; either could be used to identify the other.



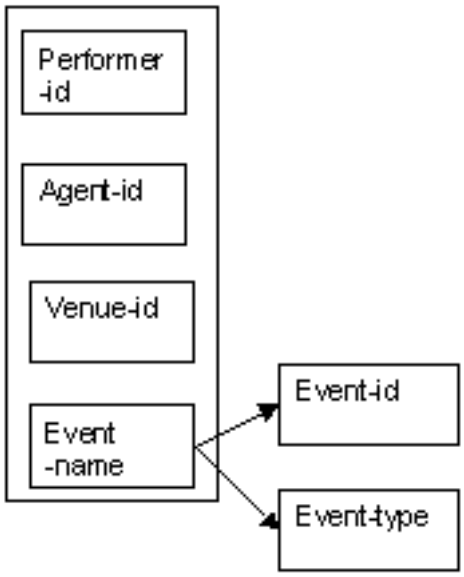
Note that the event-name depends only on the event-id as shown by the arrow in the diagram. The attributes performer-id, agent-id and venue-id do not determine the event-name.

Event-name as the determinant

There is a special relationship between the attributes event-id and event-name; each event-id and each event-name is unique.

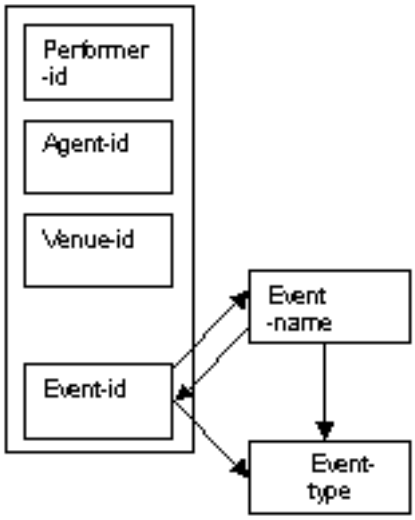
This means that we could use either the event-id or the event-name as the determinant for locating details about an event.

The determinacy diagram below shows the event-name being used as the determinant, although we would not want to use it as the primary key, as names can be difficult to get exactly right.



Event-id and event-name as determinants

We can show the special relationship between event-id and event-name by arrows illustrating the link in each direction.

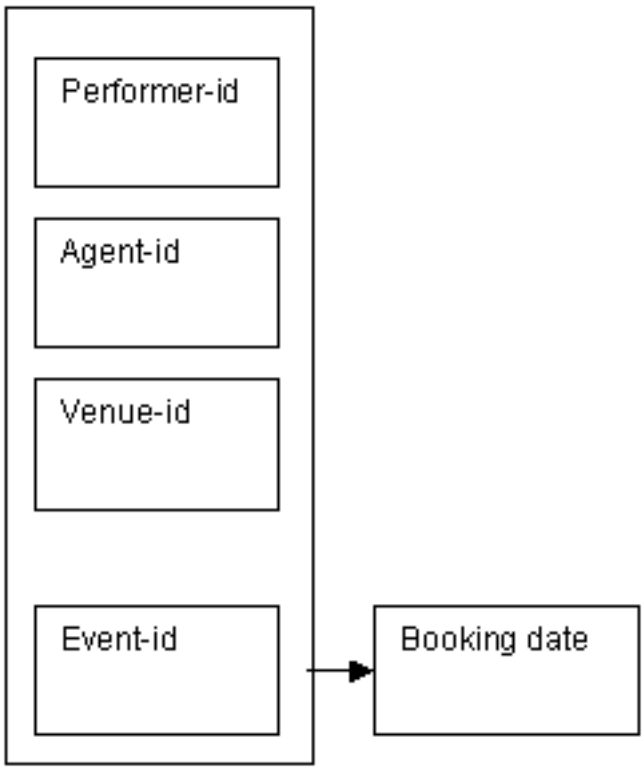


As either event-id or event-name can determine the event-type, there are links between event-id and event-type, and also between event-name and event-type.

Booking detail

In addition to the performers, agents and venues, we need to be able to identify the bookings that have been made. When a booking is made, the performer-id, agent-id, venue-id and event-id are all required in order to specify a particular event occurring on a given date. This also needs to be represented using a determinacy diagram.

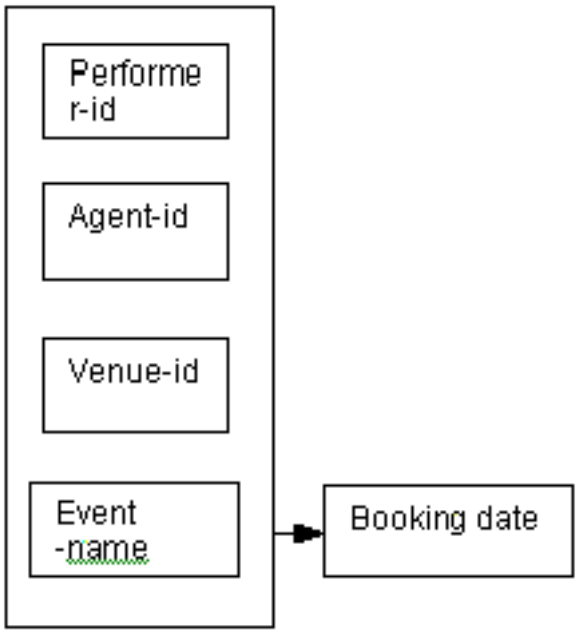
Each booking can be identified by the primary key, which is shown on the right as a combination of the attributes performer-id, agent-id, venue-id and event-id.



Note that in this instance, the arrow (coming from the outer box) indicates that all four key attributes are used to identify the booking date.

We know that each event can be identified either by the event-id or the event-name; this means that we could have an alternative representation in the determinacy diagram, substituting the attribute event-name for event-id as part of the combined key.

An alternative primary key for each booking would be a combination of performer-id, agent-id, venue-id and event-name.



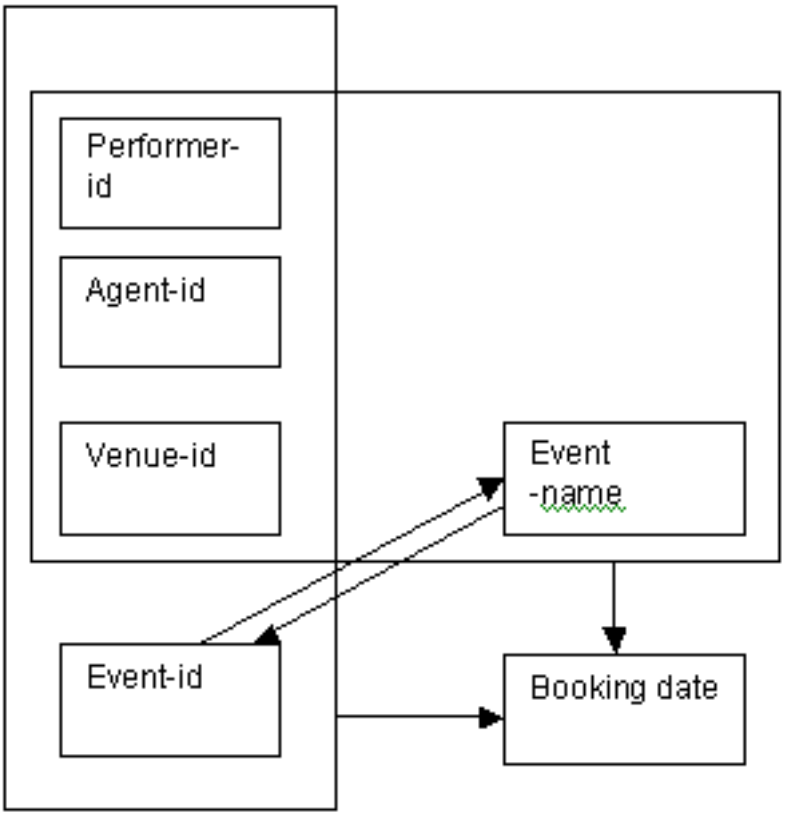
Here again, the arrow (coming from the outer box) indicates that all four key attributes are used to identify the booking date.

Here we have an overlapping key. The attribute event-name is a determinant, although it is not a candidate key for its own data. We would not want to use the event-name as a primary key, as it can present a problem in identifying the relevant tuple if the spelling is not exactly the same as in the relation.

We can show the overlapping nature of the keys for the booking details in a determinacy diagram.

The determinacy diagram below shows that the booking date could be located through a primary key constructed from the attributes performer-id, agent-id, venue-id and event-id, or by means of a primary key combining the attributes performer-id, agent-id, venue-id and event-name.

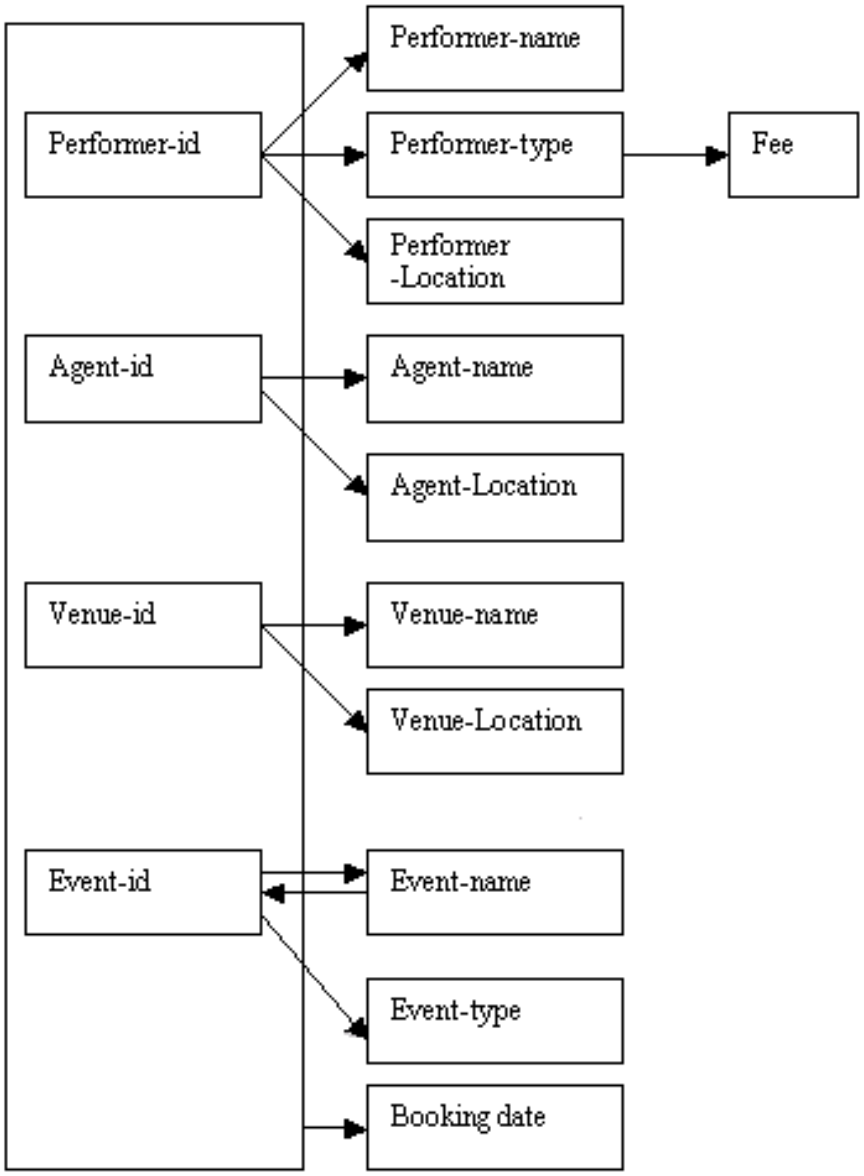
The determinacy diagram also shows the relationship between the attributes event-id and event-name.



It is not common to find overlapping keys; it is more usual to have a unique identifier which distinguishes between different items (for example, the performer-id will distinguish between different performers who may happen to have the same name). At this point in the normalisation process, overlapping keys do not present a problem, but they will be dealt with at a later stage. We will use the event-id in preference to the event-name for the time being, but we will need to remember the special relationship that exists between these two attributes.

Determinacy diagram for first normal form

The information represented in these four categories (performer, agent, venue and booking) can be displayed in a single diagram for first normal form (1NF):

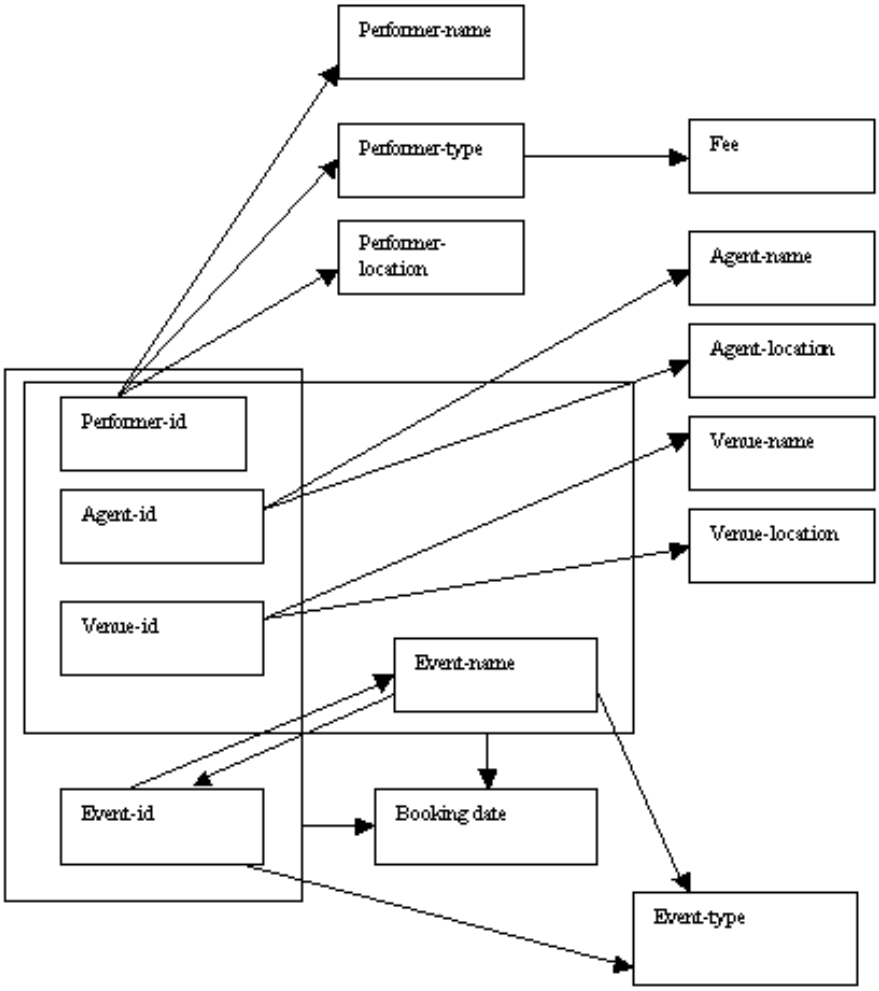


The combined determinacy diagram (above) for first normal form shows that:

- The performer attributes (name, type, location and fee) depend only on the key performer-id.
- The agent attributes (name and location) depend only on the key agent-id.

- The venue attributes (name and location) depend only on the key venue-id.
- The event attributes (name and type) depend only on the key event-id (we will examine the relationship between event-id and event-name later).
- The booking details depend on all four key attributes: performer-id, agent-id ,venue-id and event-id.

The full determinacy diagram for first normal form, showing the overlapping keys, is shown below:



The result of converting an un-normalised table of data into first normal form is to remove repeating values, so that each line in the table has the same format,

with only one value in each column for each row. This means that there will be only one value for each attribute for each tuple in a relation in first normal form.

Where more than one booking has been made for a performer, each booking is now given as a separate entry.

The original table of data has been converted into a relation in first normal form, as shown below. The relation has the same structure as the determinacy diagram, both being in first normal form, and exhibiting the following characteristics:

- All performers have a performer-id as the primary key.
- Details about agents can be determined from the primary key agent-id.
- Any venue can be identified by the venue-id as the primary key.
- All events can be determined by event-id as the primary key.
- Where a booking has been made, the key attributes performer-id, agent-id, venue-id and event-id all have values, which combine to identify each particular booking as a composite (or compound) primary key.

We can now convert our table of un-normalised data into a relation in first normal form (1NF). Note that there is at most a single value at the intersection of each row and column. This process is sometimes known as 'flattening' the table.

Table of relation in first normal form (1NF)

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
101	Baron	Singer	75	York	1295	Burton	Luton	59	Atlas	Tokyo	959	Show Time	Musical	25-Nov-1999
											907	Elgar 1	Concert	
105	Steed	Dancer	60	Berlin	1435	Nunn	Boston	35	Polis	Athens	921	Silver Shoe	Ballet	07-Jan-2002
105	Steed	Dancer	60	Berlin	1504	Lee	Taipei	54	Nation	Lisbon	942	White Lace	Ballet	10-Feb-2002
108	Jones	Actor	85	Bombay	1682	Tsang	Beijing	79	Festive	Rome	901	The Dark	Drama	29-Jul-2003
											913	What Now?	Drama	
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
112	Eagles	Actor	85	Leeds	1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
112	Eagles	Actor	85	Leeds	1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999
118	Markov	Dancer	60	Moscow										
											934	Angels	Opera	
126	Stokes	Comed -ian	90	Athens	1509	Patel	York	59	Atlas	Tokyo	945	Trick-Treat	Variety Show	02-Sep-2001
129	Chong	Actor	85	Beijing	1478	Burns	Leeds	79	Festive	Rome	926	Next Year	Drama	22-Jun-2000
											938	New Dawn	Drama	
134	Brass	Singer	75	London	1504	Lee	Taipei	28	Gratton	Boston	981	Birdsong	Musical	18-Sep-2001
					1377	Webb	Sydney							

P	Perf	Perf	Fee	Perf	A	Agent	Agent	V	Venue	Venue	E	Event	Event	Booking
140	Strong	Magic -ian	72	Rome	1478	Burns	Leeds	17	Silbury	Tunis	963	Vanish!	Magic show	18-Aug-1999
								92	Palace	Milan				
					1190	Patel	Hue							
								62	Shaw	Oxford				
141	Gomez	Music -ian	92	Lisbon	1478	Burns	Leeds	84	State	Kiev	941	Mahler 1	Concert	21-Jul-2000
											964	The Friends	Drama	
					1802	Chapel	Bristol							
143	Tan	Singer	75	Chicago	1504	Lee	Taipei	79	Festive	Rome	927	Chanson	Opera	21-Nov-2002
											971	Card Trick	Magic show	
147	Qureshi	Actor	85	London	1076	Eccles	Oxford	17	Silbury	Tunis	952	Gold Days	Drama	30-Apr-2000
147	Qureshi	Actor	85	London	1409	Arkley	York	79	Festive	Rome	988	Secret Tape	Drama	17-Apr-2000
149	Tan	Actor	85	Taipei										
150	Pointer	Magic -ian	72	Paris										
152	Peel	Dancer	60	London	1428	Vernon	Cairo	59	Atlas	Tokyo	978	Swift Step	Dance	01-Oct-2001

We can see that the relation in first normal form will still exhibit some problems when we try to insert new tuples, update existing values or delete existing tuples. This is because there is no primary key for the whole table, although each major component has its own key (performer, agent, venue, event and booking).

Insertion anomalies of first normal form

There is a problem in selecting a suitable key for the table in its current format.

If we wish to insert details for a new performer, agent, venue or booking, we need to be able to identify the key attribute and determine a value for the key for the new record, for it to be entered as a tuple in the relation.

There is no clear candidate for a key for the whole relation in first normal form. We cannot use the performer-id as a key, because not every record in the table has a performer specified. The following examples illustrate this: the venue 62 Shaw has no performer, no event and no agent; the agent 1377 Webb has made no bookings for performers, venues or events; and the event 938 New Dawn has no performer, agent or venue. A null value cannot be allowed in a key field (for reasons of entity integrity, as discussed in Chapter 2).

If we made up a fictitious performer-id value to use as the key when we wanted to insert a new agent, a new venue or a new event, we would then generate another set of problems, such as apparent double bookings.

We need to consider the possibilities for a key for the whole relation in first normal form, and identify any problems that might arise with each option. The use of the following attributes as the primary key will be considered in turn:

- Performer-id

- Agent-id
- Venue-id
- Event-id
- Performer-id, agent-id, venue-id and event-id combined

Would the attribute performer-id make a suitable key for the relation in 1NF?

The attribute performer-id is the primary key for performers, but it cannot be used as the key for the whole relation in first normal form as there are some cases where there is no relevant value, as shown in the following examples:

No performer-id for Shaw

The venue Shaw (venue-id 62) has not been used for any bookings, and therefore has no performer-id associated with it that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
								62	Shaw	Oxford				

No performer-id for venue 62

No performer-id for Webb

The agent Webb (agent-id 1377) has made no bookings for performers, and thus there is no appropriate performer-id that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
					1377	Webb	Sydney							

No performer-id for agent 1377

No performer-id for New Dawn

There are no bookings for the event New Dawn (event-id 938), and therefore there is no associated performer-id that could be used as a key.

Would the attribute agent-id make a suitable key for the relation in 1NF?

While it is the primary key for agents, the attribute agent-id would not make a good choice as the key for the whole relation in first normal form as here, too, there are times where there is no value present. This is illustrated below.

No agent-id for Shaw

No bookings have been made for the venue Shaw (venue-id 62), and therefore no agent-id is available to be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
								62	Shaw	Oxford				

No event-id for shaw

No agent-id for Tan

The actor Tan (performer-id 149) has no bookings and therefore no agent-id is available to be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
149	Tan	Actor	85	Taipei										

There is no agent-id for performer 149 Tan

Note that the performer-id as primary key for performers distinguishes between 149 Tan the actor, and 143 Tan the singer (who does have a booking).

No agent-id for New Dawn

There are no bookings for the event New Dawn (event-id 938), and therefore there is no agent-id that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
											938	New Dawn	Drama	

There is no agent-id for event 938 New Dawn

We can conclude that the attribute agent-id would not make a suitable key for the relation in first normal form.

Would the attribute venue-id make a suitable key for the relation in 1NF?

The attribute venue-id is the primary key for all venues, but it cannot be employed as the key for the whole relation in first normal form as there are instances where no value has been allocated, for example:

No venue-id for Tan

The actor Tan (performer-id 149) has no bookings at a venue and therefore there is no venue-id that can be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
149	Tan	Actor	85	Taipei				↑						

There is no venue-id for performer 149 Tan

No venue-id for Webb

The agent Webb (agent-id 1377) has made no bookings, and is therefore not associated with any venue-id that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
					1377	Webb	Sydney	↑						

No venue-id is associated with agent 1377 Webb

No venue-id for New Dawn

There are no bookings for the event New Dawn (event-id 938), and therefore there is no venue-id that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
								↑			938	New Dawn	Drama	

No venue-id is associated with event 938 New Dawn

We can conclude that the attribute venue-id would not make a suitable key for the relation in first normal form.

Would the attribute event-id make a suitable key for the relation in 1NF?

The attribute event-id is the primary key for events (although the event-name could also be used as the primary key). The examples below demonstrate that the event-id cannot be used as the key for the whole relation in first normal form, as there are cases where there is no value for the event-id.

No event-id for Tan

The actor Tan (performer-id 149) has no bookings at an event and therefore there is no event-id that can be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
149	Tan	Actor	85	Taipei										

There is no event-id for performer 149 Tan

No event-id for Shaw

The venue Shaw (venue-id 62) has not been used for any bookings, and therefore there is no event-id associated with it that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
								62	Shaw	Oxford				

No event-id for venue 62

No event-id for Webb

The agent Webb (agent-id 1377) has made no bookings, and thus there is no appropriate event-id that could be used as a key.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
					1377	Webb	Sydney							

No event-id for agent 1377

We can conclude that the attribute event-id would not make a suitable key for the relation in first normal form.

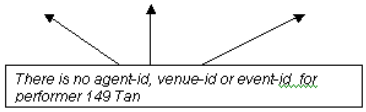
Would the combined attributes performer-id, agent-id, venue-id and event-id make a suitable key for the relation in 1NF?

The combined attributes performer-id, agent-id, venue-id and event-id serve as the primary key for all bookings, but this combination cannot be employed as the key for the whole relation in first normal form as there are entries where the key would be incomplete, for example:

No agent-id, venue-id or event-id for Tan

The actor Tan (performer-id 149) has no bookings made by an agent at a venue for an event and therefore there is no complete combined key value.

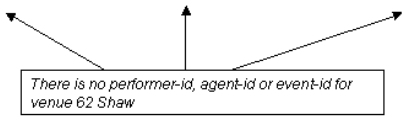
P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
149	Tan	Actor	85	Taipei										



No performer-id, agent-id or event-id for Shaw

No bookings have been made for the venue Shaw (venue-id 62), and therefore no complete combined key is available, as there is no performer, agent or event associated with the venue.

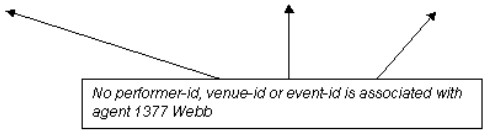
P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
								62	Shaw	Oxford				



No performer-id, venue-id or event-id for Webb

The agent Webb (agent-id 1377) has made no bookings, and there is therefore an incomplete combined key value for Webb (no performer, venue or event).

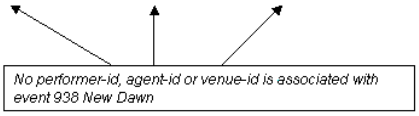
P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
					1377	Webb	Sydney							



No performer-id, agent-id or venue-id for New Dawn

The event New Dawn has not been booked, and therefore there is no complete combined key available as there is no performer, agent or venue associated with the event.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
											938	New Dawn	Drama	



We can conclude that the combination of the attributes performer-id, agent-id, venue-id and event-id would not make a suitable key for the relation in first normal form.

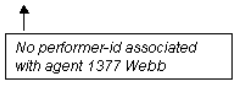
There is no obvious choice for a primary key. The attributes that we might expect to be able to use as a key (such as performer-id, agent-id, venue-id and event-id) are unsuitable because a value is not always available, and it is not possible to have a key field with a null (or empty) value (because of the requirements of entity integrity).

Arbitrary selection of a primary key for relation in 1NF

If we take an alternative approach and arbitrarily select the performer-id as the key field, this will also lead to problems.

We would not be able to insert details about new agents who have yet to make a booking, as they will not have a performer-id associated with them. Neither would it be possible to retain the tuple on agent Webb (agent-id 1377), who has yet to make a booking.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
					1377	Webb	Sydney							



We would not be able to insert details about new venues that have not yet been used for a booking, as they too will not have a performer-id associated with them. In this instance, it would not be possible to retain the tuple on the venue Shaw (venue-id 62).

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
								62	Shaw	Oxford				

No performer-id associated with venue

We would not be able to insert details about new events that had not yet been booked, as any such event will not have a performer-id associated with it. This means that we would not be able to retain the tuple on the event New Dawn, as it has not been used for a booking.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
											938	New Dawn	Drama	

No performer-id is associated with event 938

We can see that there is no single attribute, or combination of attributes, that could be used successfully to identify any record in the table; this implies that there will be difficulties when it comes to inserting new data as well as manipulating data already in the table.

We will see that the problem of not being able to find a key for the relation in first normal form will lead us into the creation of an improved structure for representing data, so that there will be no ambiguity or loss of information.

Amendment anomalies of first normal form

There is a problem in updating values in a table in first normal form. If there is more than one entry in the relation (for example, a performer who has several bookings), any change to that individual's details must be reflected in all such entries, otherwise the data will become inconsistent.

Problems if performer changes location

What would happen if a performer moved to another location, or changed name through marriage (or both)? In first normal form, the full details for a performer are repeated every time a booking is made, and each such entry would need to be updated to reflect the change in name or location. The performer 112 Eagles already has three bookings; if there is any change to the performer details, all three entries would need to be updated. If this is not done, and a further booking is made with the updated performer details, the data in the relation will become inconsistent.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
112	Eagles	Actor	85	Leeds	1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
112	Eagles	Actor	85	Leeds	1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999

Multiple entries would need to be changed to update details about 112 Eagles

Problems if agent changes location

The agent Lee (agent-id 1504) has made bookings for more than one performer, at more than one location, so if agent Lee were to move to another location it would be necessary to change details of the agent location in more than one place.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
105	Steed	Dancer	60	Berlin	1504	Lee	Taipei	54	Nation	Lisbon	942	White Lace	Ballet	10-Feb-2002
112	Eagles	Actor	85	Leeds	1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999

Multiple entries would need to be amended to update details for agent 1504 Lee

Problems if agent venue details change

The venue Atlas (venue-id 59) has been booked for more than one performer, and by more than one agent; this means that there are several entries relating to this venue. Any change to the details of the venue (perhaps a change of name following a change of ownership) would need to be made to every entry that included the venue Atlas, in order to avoid inconsistencies in the data.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
101	Baron	Singer	75	York	1295	Burton	Luton	59	Atlas	Tokyo	959	Show Time	Musical	25-Nov-1999
126	Stokes	Comed -ian	90	Athens	1509	Patel	York	59	Atlas	Tokyo	945	Trick-Treat	Variety Show	02-Sep-2001
152	Peel	Dancer	60	London	1428	Vernon	Cairo	59	Atlas	Tokyo	978	Swift Step	Dance	01-Oct-2001

More than one entry would need to be updated if details about venue 59 Atlas were changed

Problems if event details change

If one of the events were to be changed, this could affect a number of tuples

in the relation in first normal form. If the drama 952 Gold Days were to be rewritten to include songs, it would then need to be reclassified as a musical, and this information would need to be updated for every booking for that event. Even if the new musical production were allocated a new event-id, the change would still need to be reflected in every booking of the event.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
112	Eagles	Actor	85	Leeds	1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999
147	Qureshi	Actor	85	London	1076	Eccles	Oxford	17	Silbury	Tunis	952	Gold Days	Drama	30-Apr-2000

More than one entry would need to be updated if details about event 952 Gold Days were changed

Deletion anomalies of first normal form

Problems if an actor is deleted

What if we were to delete the record for the actor Eagles (performer-id 112)? In this case, Eagles has three bookings, at the venues Silbury (venue-id 17), Royale (venue-id 46) and Vostok (venue-id 75). Eagles is the only performer to have a booking at venues Royale and Vostok. The agent Ellis (agent-id 1522), who made the booking for Eagles at the venue Royale, has made no other bookings. The agent Stritch (agent-id 1460), who booked Eagles into the venue Silbury, has made no other bookings, although the venue has been booked by other agents for other performers.

The events for which Eagles has been booked include two bookings for 952 Gold Days (one by agent 1522 Ellis for venue 46 Royale, the other by agent 1504 Lee for venue 75 Vostok), and a booking for event 926 Next Year (made by agent 1460 Stritch for venue 17 Silbury). As both events have also been booked for other performers, we would not lose details of the events themselves if Eagles is deleted from the relation. If Eagles had been the only performer for either one of these events, the result would have been the loss of these details when Eagles had been deleted.

If the details for performer Eagles are deleted, not only will we lose the data about agents Ellis and Stritch, but we will also lose details of the venues Royale and Vostok. The performer Eagles has three bookings, which involve two events, Gold Days (which Eagles performs twice), and Next Year. As both these events are also performed by other individuals, the deletion of data relating to Eagles means that in this case we will not lose data about these two events. If, however, Eagles had been the only performer booked for either of these events, the event details would have been lost after the deletion of the performer Eagles.

It is worth noting that if the details for Eagles are removed from the relation, all three occurrences would have to be removed; there would be problems of data integrity and consistency if some were omitted.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
112	Eagles	Actor	85	Leeds	1522	Ellis	Madrid	46	Royale	Cairo	952	Gold Days	Drama	05-May-1999
112	Eagles	Actor	85	Leeds	1504	Lee	Taipei	75	Vostok	Kiev	952	Gold Days	Drama	16-Mar-1999

Data about agent 1522 Ellis would be lost if 112 Eagles were deleted

Data about venues 46 Royale and 75 Vostok would be lost if 112 Eagles were deleted

Problems if a performer is deleted

What if we were to delete the record for 152 Peel, the dancer? This may happen if Peel retires as a dancer.

The problem would be that not only would we remove the data related to Peel (which is our intention), but we would also unintentionally lose the data associated with the agent Vernon, as this is the only booking Vernon has made. We would also lose information stored about the event 978 Swift Step, as this is the only booking made that involves this event. Note that we would not lose details relating to the venue 59 Atlas, as this venue has also been booked for other performers.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
152	Peel	Dancer	60	London	1428	Vernon	Cairo	59	Atlas	Tokyo	978	Swift Step	Dance	01-Oct-2001

Data about agent 1428 Vernon would be lost if 152 Peel were deleted

Data about event 978 Swift Step would be lost if 152 Peel were deleted

Problems if an event is deleted

What would happen if the event 926 Next Year were to be withdrawn, and all tuples containing that event deleted?

The event Next Year is involved in two bookings, one for performer 112 Eagles, and another for performer 129 Chong.

The booking for Eagles was made by agent 1504 Lee for venue 17 Silbury. Eagles has other bookings, agent Lee has made bookings for other performers, and the venue Silbury has been booked for other events, so the deletion of this tuple will

not cause a loss of data about performers, agents or venues.

The other booking for event 926 New Year for performer Chong was made by agent 1478 Burns at venue 79 Festive. The agent Burns and the venue Festive are also involved in other bookings, but this was the only booking for performer Chong. If this tuple is deleted, we will lose all details concerning the performer 129 Chong.

P -id	Perf -name	Perf Type	Fee	Perf Loc'n	A -id	Agent name	Agent Loc'n	V -id	Venue name	Venue Loc'n	E -id	Event -name	Event -type	Booking date
112	Eagles	Actor	85	Leeds	1460	Stritch	Rome	17	Silbury	Tunis	926	Next Year	Drama	13-Aug-2000
129	Chong	Actor	85	Beijing	1478	Burns	Leeds	79	Festive	Rome	926	Next Year	Drama	22-Jun-2000

Data about performer 129 Chong would be lost if event 926 Next Year were deleted

These examples show that we need to store information about performers, agents, venues and events independently of each other, so that we do not risk losing data. The solution is to convert the relation in first normal form into a number of relations in second normal form.

Second normal form

The process of converting a relation from first normal form into second normal form is the identification of the primary keys, and the grouping together of attributes that relate to the key. This means that attributes that depend on different keys will now appear in a separate relation, where each attribute depends only on the key, whether directly or indirectly. The purpose of converting the relation into second normal form is to resolve many of the problems identified with first normal form.

Important

Second normal form (2NF)

For a relation to be in second normal form, all attributes must be fully functionally dependent on the primary key. Data items which are only partial dependencies (as they are not fully functionally dependent on the primary key) need to be extracted to form new relations.

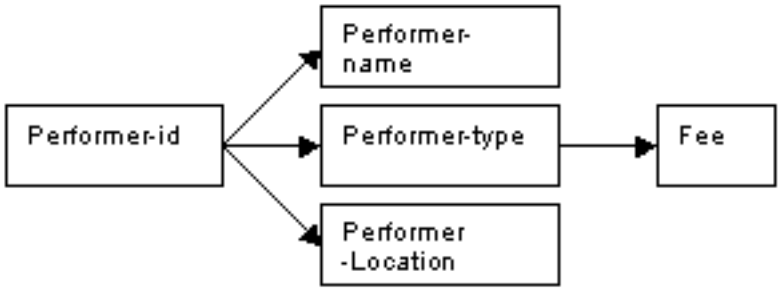
For our performer case study, the single relation in first normal form (1NF) is transformed into four relations in second normal form (working from the 1NF determinacy diagram): performers, agents, venues and bookings.

Performer details

All data relating to performers is now grouped separately from agents, venues, events and bookings. The determinacy diagram for performer details gives us a performer relation in second normal form. The primary key for the performer relation is performer-id, and the other attributes are names, performer-type, fee and location.

The creation of an independent new relation for performers has the following benefits, which resolve the problems encountered with the single relation in first normal form:

- New performers can be inserted even if they have no bookings.
- A single amendment will be sufficient to update performer details even if several bookings are involved.
- The deletion of a performer record will not result in the loss of details concerning agents, venues or events, as performers, agents, venues and events are now stored independently of each other.



Relation in second normal form: Performers

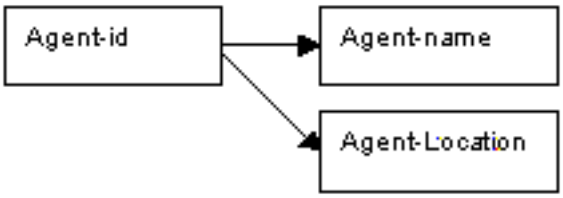
Performer-id	Performer-name	Performer-type	Fee	Performer-location
101	Baron	Singer	75	York
105	Steed	Dancer	60	Berlin
108	Jones	Actor	85	Bombay
112	Eagles	Actor	85	Leeds
118	Markov	Dancer	60	Moscow
126	Stokes	Comedian	90	Athens
129	Chong	Actor	85	Beijing
134	Brass	Singer	75	London
138	Ng	Singer	75	Penang
140	Strong	Magician	72	Rome
141	Gomez	Musician	92	Lisbon
143	Tan	Singer	75	Chicago
147	Qureshi	Actor	85	London
149	Tan	Actor	85	Taipei
150	Pointer	Magician	72	Paris
152	Peel	Dancer	80	London

Agent details

The information concerning agents is now stored separately from that of performers, venues and bookings. The determinacy diagram for agents gives us a relation for agents in second normal form. The primary key for the agents relation is agent-id, and the remaining attributes are name and location.

The new relation for agents has the following benefits, which resolve the problems encountered with the single relation in first normal form because the new relation is independent from performers, venues and bookings:

- New agents can be inserted even if they have made no bookings.
- A single change will be enough to update agent details, even if several bookings are involved.
- Agent details will now no longer be lost if a performer is deleted, as performers, agents, venues and events are now stored independently of each other.



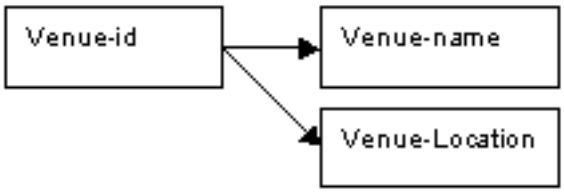
Relation in second normal form: Agents

Agent-id	Agent-name	Agent-location
1295	Burton	Luton
1435	Nunn	Boston
1504	Lee	Taipei
1682	Tsang	Beijing
1460	Stritch	Rome
1522	Ellis	Madrid
1478	Burns	Leeds
1377	Webb	Sydney
1509	Patel	York
1190	Patel	Hue
1802	Chapel	Bristol
1076	Eccles	Oxford
1409	Arkley	York
1428	Vernon	Cairo

Venue details

The creation of a new relation solely to store the details of venues has the following effects, which resolve the problems identified with the single relation in first normal form:

- Details of a new venue can be inserted, whether or not it has been booked.
- If the name of the venue is changed, the alteration only needs to be made once, in the venue relation, not for every booking of that venue.
- If details of a performer are deleted, and the performer had the only booking at a particular venue, details of the venue will not be lost.



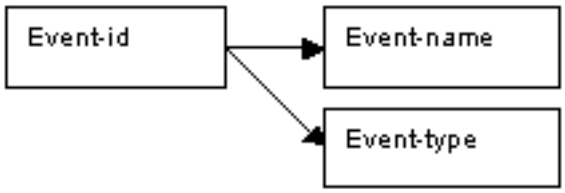
Relation in second normal form: Venues

Venue-id	Venue-name	Venue-location
59	Atlas	Tokyo
35	Polis	Athens
54	Nation	Lisbon
79	Festive	Rome
46	Royale	Cairo
28	Gratton	Boston
75	Vostok	Kiev
84	State	Kiev
82	Tower	Lima
17	Silbury	Tunis
92	Palace	Milan
62	Shaw	Oxford

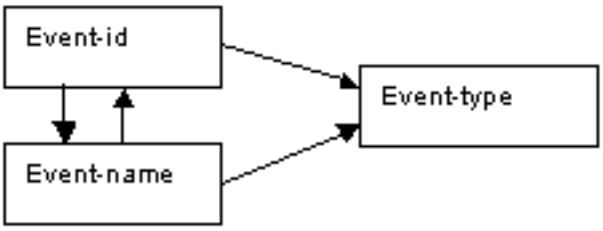
Event details

- A new relation is created to hold details of individual events.
- Details of a new event can be inserted, whether or not it has been booked.
- If the name of the event is changed, the alteration only needs to be made once, in the event relation, not for every booking of that event.
- If details of a performer are deleted, and the performer had the only booking of a particular event, details of the event will not be lost.

The determinacy diagram could be represented as follows:



An alternative representation of the determinacy diagram illustrates that the attribute event-name is also a determinant, although it is not a candidate key:

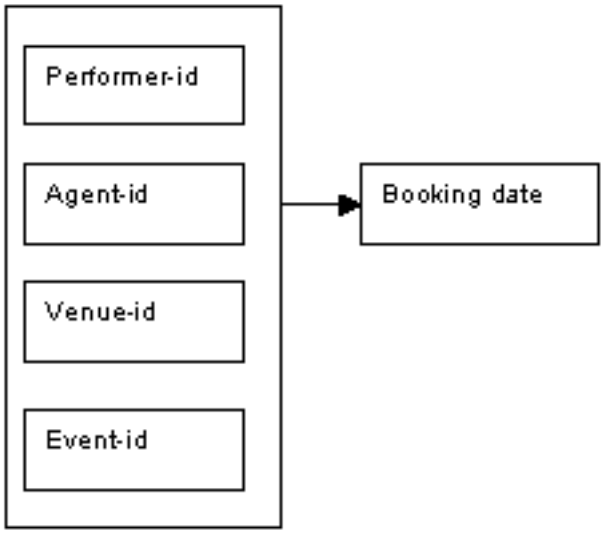


Relation in second normal form: Events

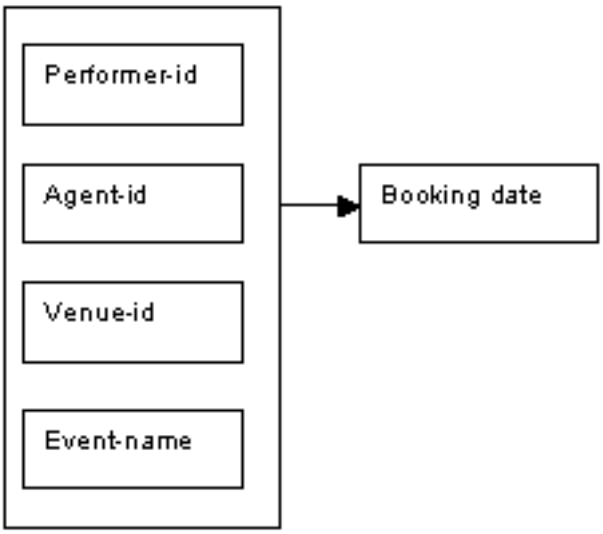
Event-id	Event-name	Event-type
901	The Dark	Drama
907	Elgar 1	Concert
913	What Now?	Drama
921	Silver Shoe	Ballet
926	Next Year	Drama
927	Chanson	Opera
934	Angels	Opera
938	New Dawn	Drama
941	Mahler 1	Concert
942	White Lace	Ballet
945	Trick-Treat	Variety show
952	Gold Days	Drama
957	Quicktime	Musical
959	Show Time	Musical
963	Vanish!	Magic show
964	The Friends	Drama
971	Card Trick	Magic show
978	Swift Step	Dance
981	Birdsong	Musical
988	Secret Tape	Drama

Booking details

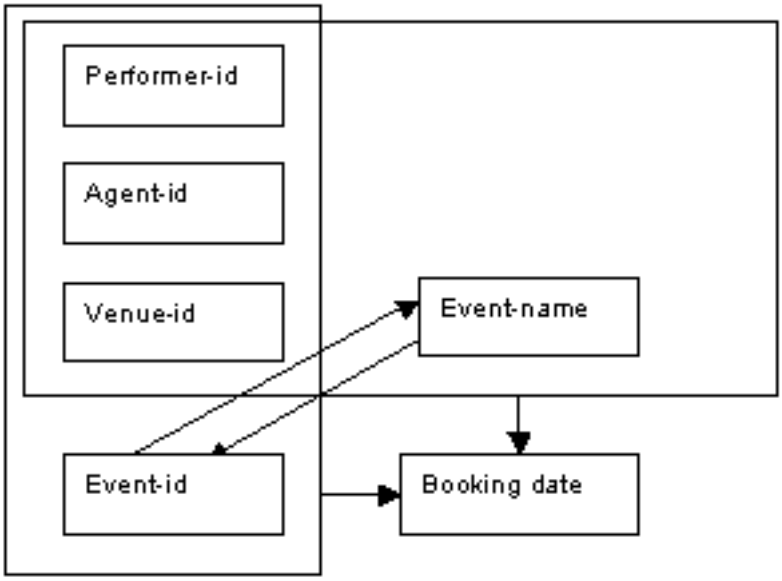
Every time a booking is made, the details are recorded in the relation called Bookings. There is no need to store all the details of the performer, agent, venue and event for each booking that is made, as this information can be acquired from the relevant relation for performers, agents, venues and event. The information that is needed for the Booking relation is the performer-id, agent-id, venue-id and event-id (these four attributes together form the key for this relation), and the booking date.



Another possible key for the Booking relation involves the attributes performer-id, agent-id, venue-id and event-name; as three of the four attributes in this key are the same as the first key described for this relation, we have an example of overlapping keys. Note that the overlapping keys are not resolved in the transformation from first normal form to second normal form, as event-id and event-name are part of each key. Conversion from first to second normal form extracts all non-key attributes which are only partially dependent on the key, and as such event-id and event-name remain as they are part of the key.



The determinacy diagram below shows the overlapping keys for the Bookings relation, and also illustrates the dependencies between the attributes event-id and event-name:



The details of the Bookings relation are shown below.

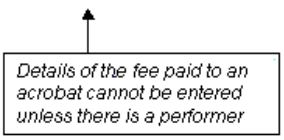
Relation in second normal form: Bookings

Performer -id	Agent -id	Venue -id	Event -id	Event -name	Booking date
101	1295	59	959	Show Time	25-Nov-1999
105	1435	35	921	Silver Shoe	07-Jan-2002
105	1504	54	942	White Lace	10-Feb-2002
108	1682	79	901	The Dark	29-Jul-2003
112	1460	17	926	Next Year	13-Aug-2000
112	1522	46	952	Gold Days	05-May-1999
112	1504	75	952	Gold Days	16-Mar-1999
126	1509	59	945	Trick-Treat	02-Sep-2001
129	1478	79	926	Next Year	22-Jun-2000
134	1504	28	981	Birdsong	18-Sep-2001
138	1509	84	957	Quicktime	18-Aug-1999
140	1478	17	963	Vanish!	18-Aug-1999
141	1478	84	941	Mahler 1	21-Jul-2000
143	1504	79	927	Chanson	21-Nov-2002
147	1076	17	952	Gold Days	30-Apr-2000
147	1409	79	988	Secret Tape	17-Apr-2000
152	1428	59	978	Swift Step	01-Oct-2001

Insertion anomalies of second normal form

We cannot enter a fee for a type of performer unless there is a performer of that type already present in the relation in second normal form. The reason for this is that if there is no existing performer of that type, there will be no performer-id value available as a key. If we want to add that acrobats are paid 65 (in whatever currency), we cannot do so unless we are able to enter complete details for a specific individual. Note that this performer would not have to have a booking, but there must be at least one person associated with a performer-type before.

Performer-id	Performer-name	Performer-type	Fee	Performer-location
		Acrobat	65	



Amendment anomalies of second normal form

If performer Stokes (performer-id 126), who is the only comedian in the relation, retrains and changes career to become a magician, we will then lose the information that comedians are paid a fee of 90 (in whatever currency is used). Stokes will then be paid 72, which is the fee for all magicians.

Performer-id	Performer-name	Performer-type	Fee	Performer-location
126	Stokes	Comedian	90	Athens

Performer-id	Performer-name	Performer-type	Fee	Performer-location
126	Stokes	Magician	72	Athens

Details of the fee paid to a comedian are lost if Stokes becomes a magician

We would also find an amendment anomaly if the fee paid to a particular type of performer changed. If all singers were granted a new rate, all tuples relating to singers would need to be updated, otherwise the data would become inconsistent.

Deletion anomalies of second normal form

If Gomez (performer-id 141), the only musician in the relation, decides to retire, we will lose the information regarding the fee of 92 paid to musicians.

Performer-id	Performer-name	Performer-type	Fee	Performer-location
141	Gomez	Musician	92	Lisbon

Details of the fee paid to musicians will be lost if Gomez retires

All these anomalies are caused by the fee paid to the performer being dependent on the performer-type, and not directly on the primary key performer-id. This indirect, or transitive, dependency can be resolved by transforming the relations in second normal form into third normal form, by extracting the attributes involved in the indirect dependency into a separate new relation.

Third normal form

The reason for converting a table from second normal form into third normal form is to ensure that data depends directly on the primary key, and not through some other relationship with another attribute (known as an indirect, or transitive, dependency).

Important

Third normal form (3NF)

A relation is in third normal form if there are no indirect (or transitive) dependencies between the attributes; for a relation to be in third normal form, all attributes must be directly dependent on the primary key.

An indirect dependency is resolved by creating a new relation for each entity; these new relations contain the transitively dependent attributes together with the primary key.

The conversion of a relation into third normal form will resolve anomalies identified in second normal form.

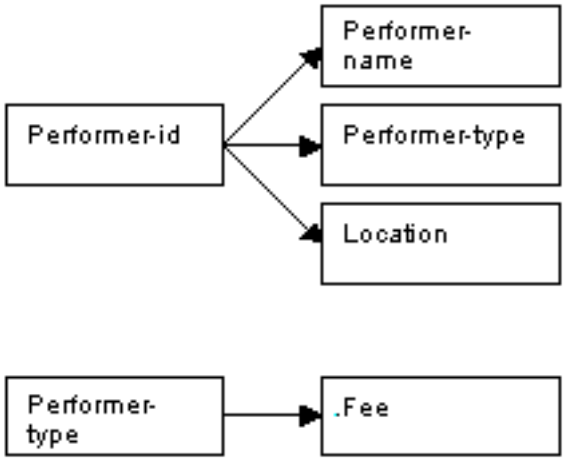
We now have six relations in third normal form: Performers, Fees, Agents, Venues, Events and Bookings.

Performer details

As before, the name and location of each performer depends on the performer-id. We noticed in second normal form that there were problems associated with having the fee contained within the performer relation, as the value of the fee depended on the performer-type and not on performer-id, demonstrating a transitive dependency.

One solution would be to create a new relation with performer-type as the key, and the fee as the other attribute; performer-type would also remain in the relation Performers, but the fee would be removed.

The relations for Performer and Fees follow the determinacy diagrams below.



Relation in third normal form: Performers

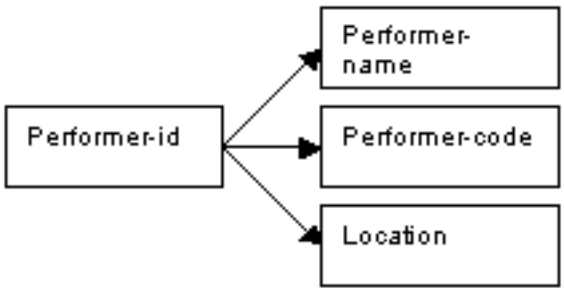
Performer-id	Performer-name	Performer-code	Performer-location
101	Baron	Singer	York
105	Steed	Singer	Berlin
108	Jones	Actor	Bombay
112	Eagles	Actor	Leeds
118	Markov	Dancer	Moscow
126	Stokes	Comedian	Athens
129	Chong	Actor	Beijing
134	Brass	Singer	London
138	Ng	Singer	Penang
140	Strong	Magician	Rome
141	Gomez	Musician	Lisbon
143	Tan	Singer	Chicago
147	Qureshi	Actor	London
149	Tan	Actor	Taipei
150	Pointer	Magician	Paris
152	Peel	Dancer	London

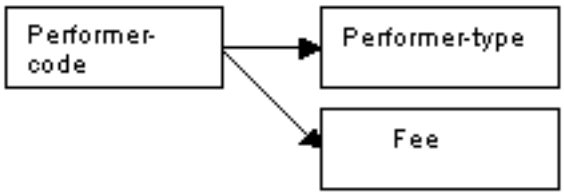
Relation in third normal form: Fees

Performer-type	Fee
Singer	75
Dancer	60
Actor	85
Comedian	90
Magician	84
Musician	92

A possible problem with this approach is the format of data entry of new performers; if "ACROBAT", "Acrobat" or "acrobat" are entered, they might not be recognised as the same performer-type. In addition, if an error is made and "arcobat" is entered, this may not be recognised. To deal with this problem, we have used a code for performer-type in the Performer relation. This code is then used as the key in the Fees relation, and the other attributes are performer-type and the fee, both of which depend on the performer-code as primary key. (We could have introduced the performer-code into the table of un-normalised data.)

The relations for Performer and Fees follow the determinacy diagrams below.





Relation in third normal form: Performers

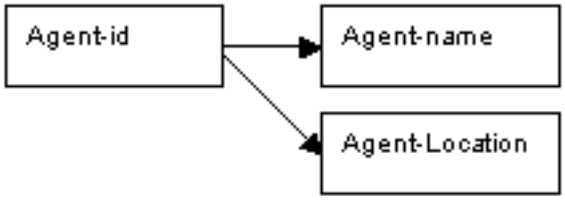
Performer-id	Performer-name	Performer-code	Performer-location
101	Baron	0348	York
105	Steed	0862	Berlin
108	Jones	0729	Bombay
112	Eagles	0729	Leeds
118	Markov	0862	Moscow
126	Stokes	0244	Athens
129	Chong	0729	Beijing
134	Brass	0348	London
138	Ng	0348	Penang
140	Strong	0360	Rome
141	Gomez	0915	Lisbon
143	Tan	0348	Chicago
147	Qureshi	0729	London
149	Tan	0729	Taipei
150	Pointer	0360	Paris
152	Peel	0862	London

Relation in third normal form: Fees

Performer-code	Performer-type	Fee
0348	Singer	75
0862	Dancer	60
0729	Actor	85
0244	Comedian	90
0360	Magician	84
0915	Musician	92

Agent details

There is no change to the determinacy diagram for Agents, as this is already in third normal form (there were no transitive dependencies). The relation follows the determinacy diagram below.

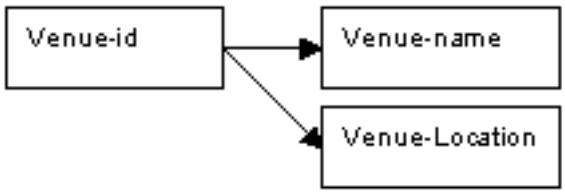


Relation in third normal form: Agents

Agent-id	Agent-name	Agent-location
1295	Burton	Luton
1435	Nunn	Boston
1504	Lee	Taipei
1682	Tsang	Beijing
1460	Stritch	Rome
1522	Ellis	Madrid
1478	Burns	Leeds
1377	Webb	Sydney
1509	Patel	York
1190	Patel	Hue
1802	Chapel	Bristol
1076	Eccles	Oxford
1409	Arkley	York
1428	Vernon	Cairo

Venue details

The data on Venues is already in third normal form as there were no transitive dependencies; there are therefore no changes to the determinacy diagram shown below, and the relation which follows.

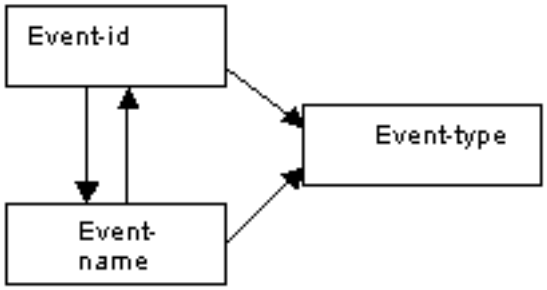


Relation in third normal form: Venues

Venue-id	Venue-name	Venue-location
59	Atlas	Tokyo
35	Polis	Athens
54	Nation	Lisbon
79	Festive	Rome
46	Royale	Cairo
28	Gratton	Boston
75	Vostok	Kiev
84	State	Kiev
82	Tower	Lima
17	Silbury	Tunis
92	Palace	Milan
62	Shaw	Oxford

Event details

The Events relation is already in third normal form as there are no transitive dependencies. There is the special relationship that exists between the attributes event-id and event-name, which does not present a problem within the Events relation itself, but creates difficulties in the Bookings relation because of the overlapping key which results.



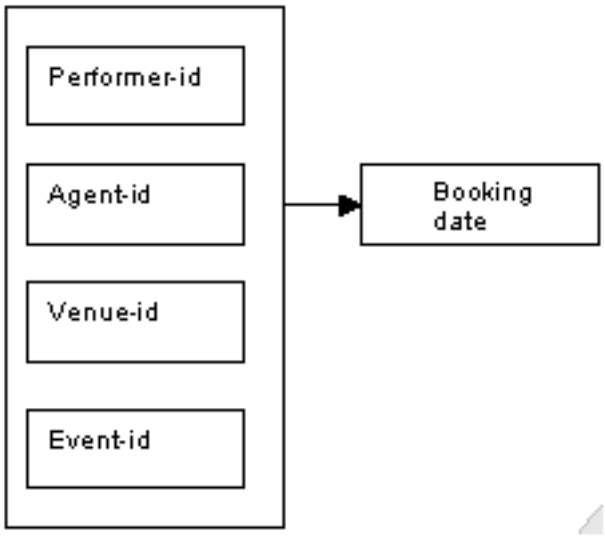
Relation in third normal form: Events

Event-id	Event-name	Event-type
901	The Dark	Drama
907	Elgar 1	Concert
913	What Now?	Drama
921	Silver Shoe	Ballet
926	Next Year	Drama
927	Chanson	Opera
934	Angels	Opera
938	New Dawn	Drama
941	Mahler 1	Concert
942	White Lace	Ballet
945	Trick-Treat	Variety show
952	Gold Days	Drama
957	Quicktime	Musical
959	Show Time	Musical
963	Vanish!	Magic show
964	The Friends	Drama
971	Card Trick	Magic show
978	Swift Step	Dance
981	Birdsong	Musical
988	Secret Tape	Drama

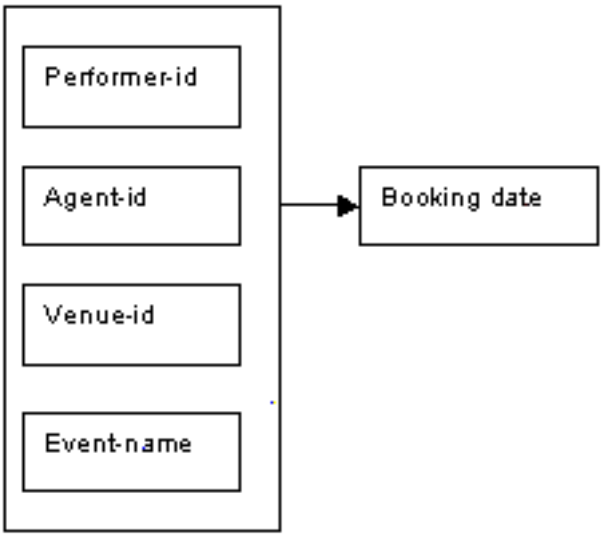
Booking details

The relation Bookings, with its composite determinants of performer-id, agent-id, venue-id and event-id, or performer-id, agent-id, venue-id and event-name, is already in third normal form as there are no transitive dependencies. The determinacy diagrams and the associated relation are illustrated below.

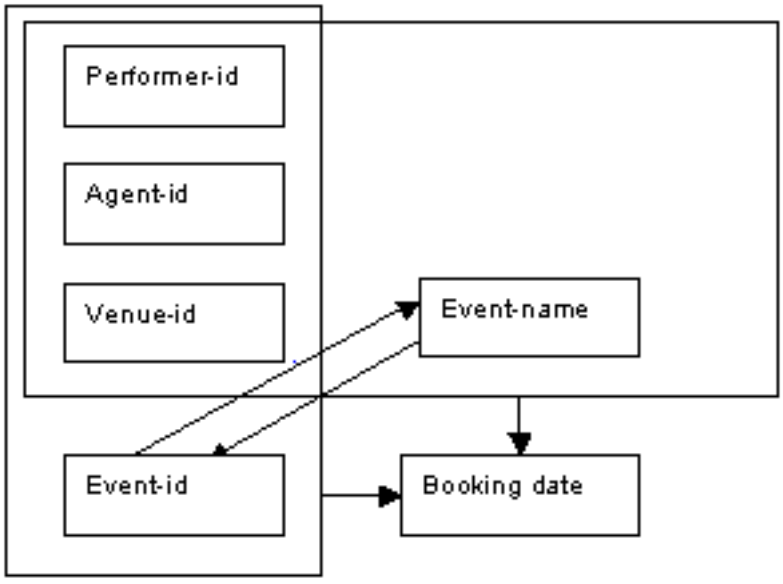
This determinacy diagram illustrates the combination of performer-id, agent-id, venue-id and event-id used as the determinant for the Bookings relation:



The next determinacy diagram shows the choice of performer-id, agent-id, venue-id and event-name as the determinants for the Bookings relation.



The determinacy diagram below combines the previous two determinacy diagrams to show the overlapping keys for the Bookings relation, and illustrates the dependencies between the attributes event-id and event-name.



The details of the Bookings relation are shown below.

Relation in third normal form: Bookings

Performer -id	Agent -id	Venue -id	Event -id	Event -name	Booking Date
101	1295	59	959	Show Time	25-Nov-1999
105	1435	35	921	Silver Shoe	07-Jan-2002
105	1504	54	942	White Lace	10-Feb-2002
108	1682	79	901	The Dark	29-Jul-2003
112	1460	17	926	Next Year	13-Aug-2000
112	1522	46	952	Gold Days	05-May-1999
112	1504	75	952	Gold Days	16-Mar-1999
126	1509	59	945	Trick-Treat	02-Sep-2001
129	1478	79	926	Next Year	22-Jun-2000
134	1504	28	981	Birdsong	18-Sep-2001
138	1509	84	957	Quicktime	18-Aug-1999
140	1478	17	963	Vanish!	18-Aug-1999
141	1478	84	941	Mahler 1	21-Jul-2000
143	1504	79	927	Chanson	21-Nov-2002
147	1076	17	952	Gold Days	30-Apr-2000
147	1409	79	988	Secret Tape	17-Apr-2000
152	1428	59	978	Swift Step	01-Oct-2001

Summary of the first three normal forms

We have seen how the original set of data items has been transformed through the initial process of identifying dependencies between data items, the formulation of successively higher normal-form collections of relations, each of which has represented an increasingly flexible design. The steps used to derive each successive normal form are summarised below:

- Identify data items which are the determinants of other data items, and through the removal of any repeating groups, form the data items into an initial first normal form relation.
- Identify any attributes that are not included in the primary key of the relation, which are not dependent on all of the primary key (this is sometimes called 'removing part-key dependencies'). It is also worth bearing in mind that this step does not arise for relations that have a single-attribute primary key.

- Identify any attributes that are not directly determined by the key (this is sometimes called 'removing transitive dependencies').

We shall see in a later chapter on database design that there is further work that can be done to normalise sets of relations, and alternative approaches to reaching third normal form (3NF). However, 3NF represents a point where we have gained a significant degree of flexibility in the design of a database application, and it is a point at which normalisation of many applications is considered to be complete.

Review questions

One of the biggest challenges when designing a database system is to obtain a correct and complete set of requirements from the prospective users of the system. Modern development methods place a strong emphasis on the need to develop prototypes of the system, so that these can be demonstrated to future users to clarify that what is being developed is what is actually required. Information gathering about the way in which an application is to work is a vital process which requires much attention to detail. This question provides an exercise in formulating the questions to be used in a data-analysis scenario. The importance of preparing for discussions about system requirements cannot be over-emphasised, as users often are short of time, have other commitments, and require guidance in describing the information required for a design.

Review question 1

Imagine that you have been commissioned by the owner of a small business to develop a database of the projects he is running. You know that the database is required to store details of the following:

1. The projects being undertaken, including expected start and finish dates.
2. Tasks required to complete each project.
3. Contract staff recruited to assist with the projects.
4. The budgets for projects.
5. The resources being used in projects and their costs.

Design a questionnaire you might use to assist you in obtaining the details of dependencies between data items when discussing the database with the business owner.

Review question 2

Given below is a possible series of answers to the questions in the previous question. Given these responses, formulate the data items mentioned into a first normal form relation.

1. How is each project identified?

   Each project is to be allocated a unique project number. This number need only be two digits long, as there will never be more that 99 projects to be stored at one time.

2. Is it required to store both expected and actual completed start and finish dates for projects?

   Yes, all four data items are required, and the same four data items are required for tasks as well.

3. How are tasks identified?

   They also have a unique task number, which again can be safely limited to two digits. So each task is identified by the combination of the project number of the project within which it occurs, and its own task number.

4. Do projects have many tasks?

   Yes, each project typically consists of about 10 tasks.

5. Can a task be split between more than one project?

   No, a task is always considered to take place within one project only.

6. Are employees assigned to projects, or to specific tasks within projects? How many tasks can an employee work on at one time?

   Employees are assigned to specific tasks within projects, so each employee can work on a number of tasks at one time. Furthermore, each task has an employee allocated to it who is specifically responsible for its successful completion. Each project has a project leader responsible for that project's successful completion.

7. What is required to be stored about contract staff pay?

   Each staff member is paid at a monthly rate, that rate being determined entirely by the highest qualification held by the staff member. We simply need to record the appropriate qualification for each staff member, and the monthly rate at which they are paid, plus the start and end dates of their current contact.

Review question 3

Remove any part-key dependencies from the relation produced in question 2 to produce a set of second normal form relations.

Review question 4

From the second normal form design in the previous question, produce a set of third normal form relations, by removing any indirect or transitive dependencies.

Review question 5

Explain the role of determinacy diagrams in database application development.

Review question 6

What is a repeating group? Why is it necessary to remove repeating groups in Relational database design?

Review question 7

Explain the term 'part-key dependency', and its role in normalisation.

Review question 8

What is the difference between second and third normal form relations?

Discussion topic

As mentioned at the start of the review questions, the process of eliciting information about the requirements of computer applications is an extremely important and potentially difficult one. Among the techniques that are commonly used to capture the requirements of users and other stakeholders in the system are:

- Interviews, which vary in different organisations and between individuals in the amount of planning and pre-determined questions
- Questionnaire surveys, in the following formats: written, e-mail or web-based
- Brainstorming
- Direct observation of users carrying out tasks

All of these techniques and more can play a useful role in capturing requirements, and each technique has particular strengths and weaknesses. You are encouraged to discuss with other students the strengths and weaknesses you consider each of the techniques listed above have in obtaining accurate and comprehensive information about the requirements for a new computer application. You should include in the discussion any experiences you have had yourself of good or bad practice in the process of requirements capture.

Application and further work

You are encouraged to consider the strengths and weaknesses of the application developed in the review questions.

Firstly, identify the additional flexibility gained by each successive stage of the normalisation process. That is, clarify the sorts of data manipulation that can be carried out in the more normalised versions of the design, compared to the un-normalised design.

Secondly, consider to what extent this extra flexibility is likely to be useful to the business owner, and whether it is worth the overhead of managing the additional tables.

Chapter 9. Advanced Data Normalisation

Table of contents

Objectives

At the end of this chapter you should be able to:

- Convert a set of relations to Boyce-Codd normal form.
- Describe the concept of multi-valued dependency, and be able to convert a set of relations to fourth normal form.
- Avoid a number of problems associated with decomposing relations for normalisation.

- Describe how denormalisation can be used to improve the performance response of a database application.

Context

This chapter relates closely to the previous two on database design. It finalises the material on normalisation, demonstrates how a fully normalised design can equally be represented as an entity-relationship model, and addresses the impact that a target DBMS will have on the design process. The issues relating to appropriate choices of DBMS-specific parameters, to ensure the efficient running of the application, relate strongly to the material covered in the chapters on indexing and physical storage. Information in all three chapters can be used in order to develop applications which provide satisfactory response times and make effective use of DBMS resources.

Recap

Introduction

In parallel with this chapter, you should read Chapter 14 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

In this concluding database design unit, we bring together a number of advanced aspects of database application design. It begins by extending the coverage of data normalisation in an earlier chapter, describing Boyce-Codd normal form (a refinement of the original third normal form) and providing a different view of how to generate a set of relations in third normal form. The chapter then looks at a number of important issues to be considered when decomposing relations during the process of normalisation. Finally, the important topic of physical database design is included, which shows the impact that DBMS-specific parameters can have in the development of an application. Many of these considerations have a direct impact on both the flexibility and the performance response of the application.

Before starting work on this chapter

This chapter addresses a number of advanced issues relating to data normalisation. It is very important that you fully understand all the concepts and techniques introduced in the previous chapter, Data Normalisation.

You should not attempt this chapter until you are confident in your understanding and application of the following concepts:

- **Functional dependency**

- Fully functional dependency
- Partial functional dependency
- Direct dependency
- Transitive (indirect) dependency
- **Determinants**
- Determinant
- Determinacy diagrams
- **Normal forms**
- Un-normalised form (UNF)
- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)

Summary of the first three normal forms

The following is a brief summary of the first three normal forms:

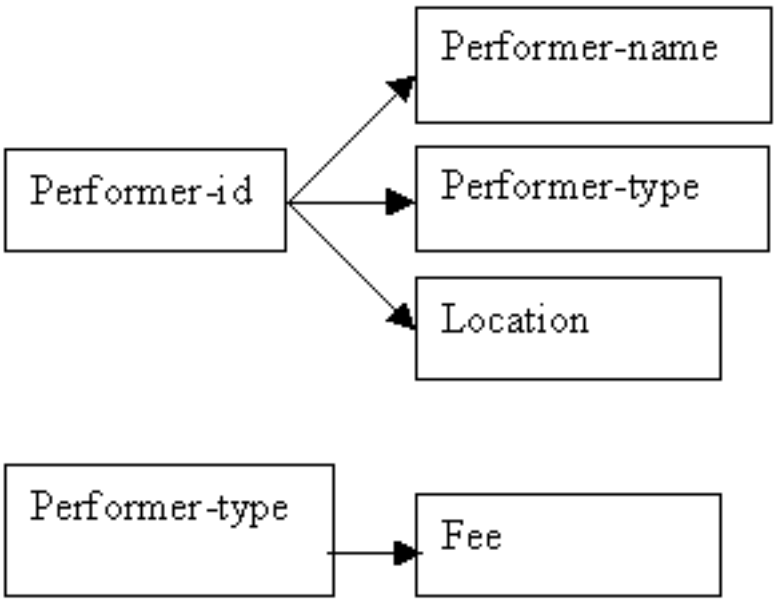
- 1NF — Identify the determinants of data items, and through the removal of any repeating groups, arrange the data items into an initial first normal form relation.
- 2NF — Remove part-key dependencies from the relations in first normal form. I.e. for non-key attributes, remove those attributes that are not fully functionally dependent on the whole of the primary key (and form new entities where these attributes are fully functionally dependent on the whole primary key).
- 3NF — Remove any transitive (indirect) dependencies from the set of relations in second normal form (to produce a set of relations where all attributes are directly dependent on the primary key).

Third normal form determinacy diagrams and relations of Performer

In this chapter we shall be extending the work from the previous chapter on the data for the Performer case study. As a starting point, we shall first present the determinacy diagrams and the third normal form relations developed for this case study.

Case study

Determinacy diagram: Performers and Fees



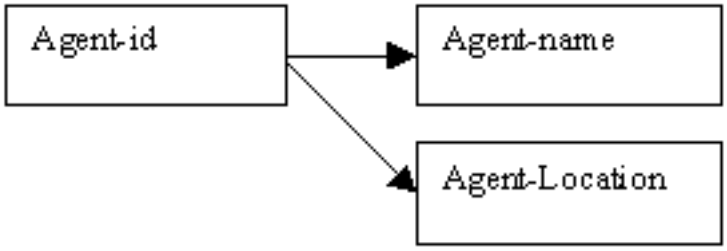
Relation in third normal form: Performers

Performer-id	Performer-name	Performer-code	Performer-location
101	Baron	Singer	York
105	Steed	Singer	Berlin
108	Jones	Actor	Bombay
112	Eagles	Actor	Leeds
118	Markov	Dancer	Moscow
126	Stokes	Comedian	Athens
129	Chong	Actor	Beijing
134	Brass	Singer	London
138	Ng	Singer	Penang
140	Strong	Magician	Rome
141	Gomez	Musician	Lisbon
143	Tan	Singer	Chicago
147	Qureshi	Actor	London
149	Tan	Actor	Taipei
150	Pointer	Magician	Paris
152	Peel	Dancer	London

Relation in third normal form: Fees

Performer-type	Fee
Singer	75
Dancer	60
Actor	85
Comedian	90
Magician	84
Musician	92

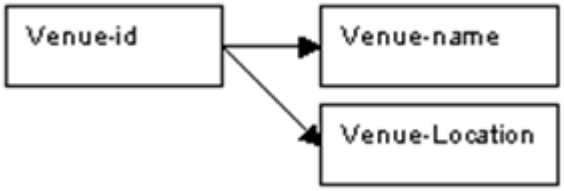
Determinacy diagram: Agents



Relation in third normal form: Agents

Agent-id	Agent-name	Agent-location
1295	Burton	Luton
1435	Nunn	Boston
1504	Lee	Taipei
1682	Tsang	Beijing
1460	Stritch	Rome
1522	Ellis	Madrid
1478	Burns	Leeds
1377	Webb	Sydney
1509	Patel	York
1190	Patel	Hue
1802	Chapel	Bristol
1076	Eccles	Oxford
1409	Arkley	York
1428	Vernon	Cairo

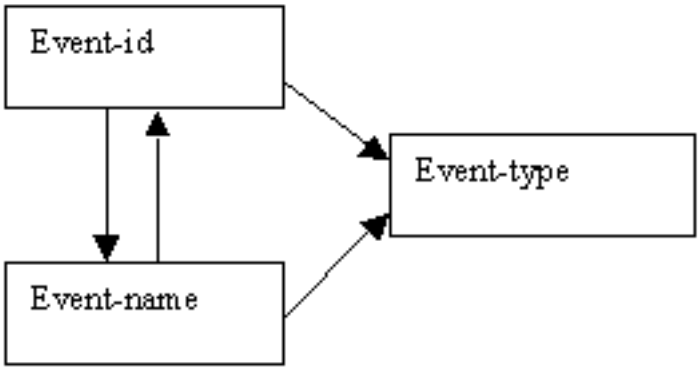
Determinacy diagram: Venues



Relation in third normal form: Venues

Venue-id	Venue-name	Venue-location
59	Atlas	Tokyo
35	Polis	Athens
54	Nation	Lisbon
79	Festive	Rome
46	Royale	Cairo
28	Gratton	Boston
75	Vostok	Kiev
84	State	Kiev
82	Tower	Lima
17	Silbury	Tunis
92	Palace	Milan
62	Shaw	Oxford

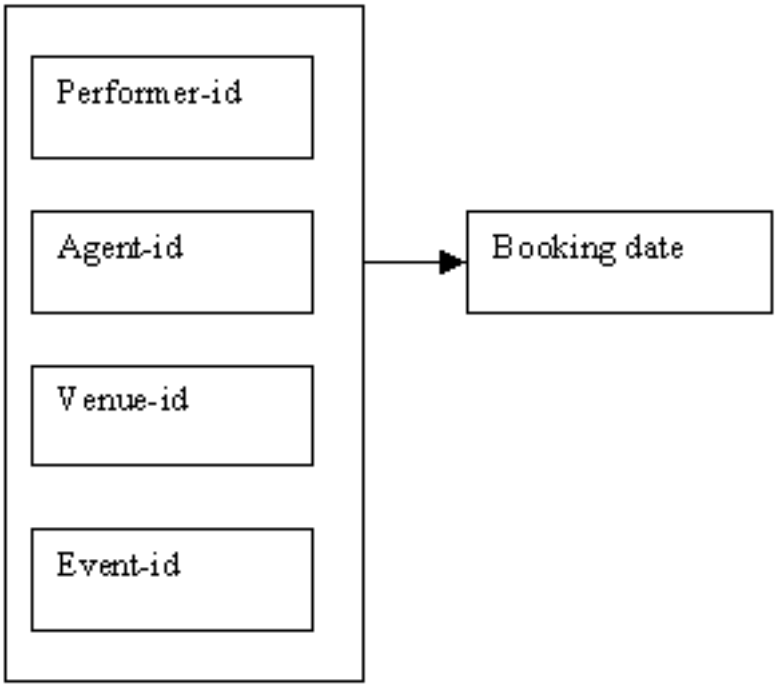
Determinacy diagram: Events

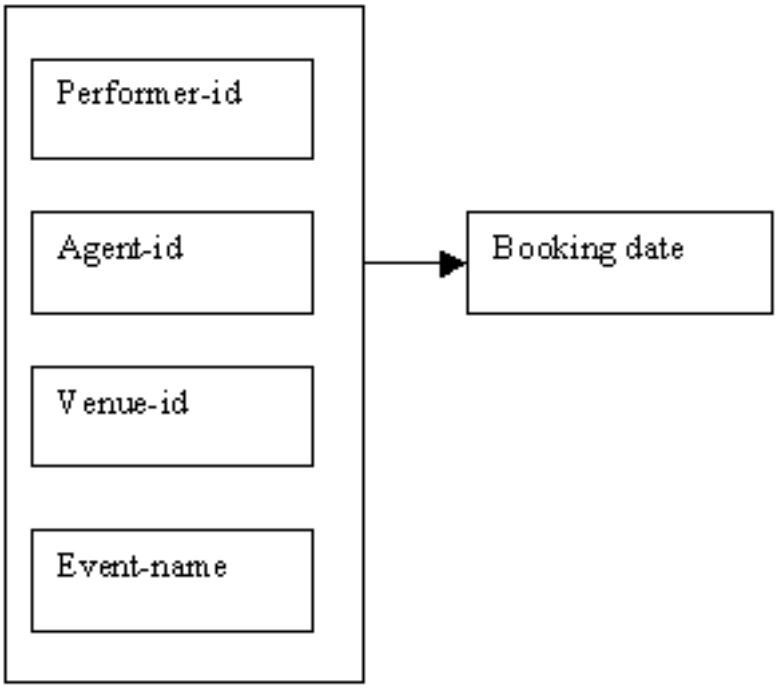


Relation in third normal form: Events

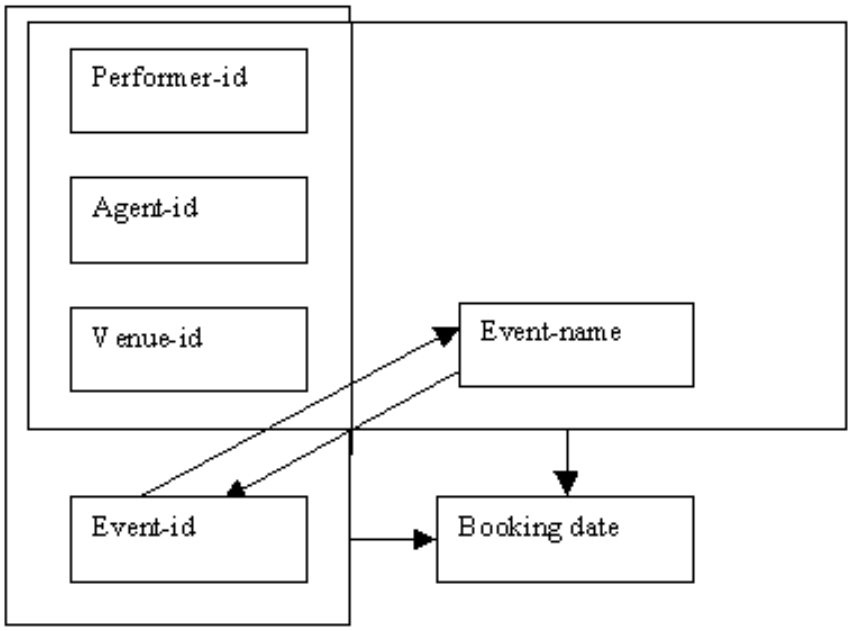
Event-id	Event-name	Event-type
901	The Dark	Drama
907	Elgar 1	Concert
913	What Now?	Drama
921	Silver Shoe	Ballet
926	Next Year	Drama
927	Chanson	Opera
934	Angels	Opera
938	New Dawn	Drama
941	Mahler 1	Concert
942	White Lace	Ballet
945	Trick-Treat	Variety show
952	Gold Days	Drama
957	Quicktime	Musical
959	Show Time	Musical
963	Vanish!	Magic show
964	The Friends	Drama
971	Card Trick	Magic show
978	Swift Step	Dance
981	Birdsong	Musical
988	Secret Tape	Drama

Determinacy diagrams: Bookings


Performer-id
Agent-id
Venue-id
Event-id
Booking date


Performer-id
Agent-id
Venue-id
Event-name
Booking date

The determinacy diagram below combines the previous two determinacy diagrams to show the overlapping keys for the Bookings relation, and illustrates the dependencies between the attributes event-id and event-name:



The details of the Bookings relation are shown below:

Relation in third normal form: Bookings

Performer-id	Agent-id	Venue-id	Event-id	Event-name	BookingDate
101	1295	59	959	Show Time	25-Nov-1999
105	1435	35	921	Silver Shoe	07-Jan-2002
105	1504	54	942	White Lace	10-Feb-2002
108	1682	79	901	The Dark	29-Jul-2003
112	1460	17	926	Next Year	13-Aug-2000
112	1522	46	952	Gold Days	05-May-1999
112	1504	75	952	Gold Days	16-Mar-1999
126	1509	59	945	Trick-Treat	02-Sep-2001
129	1478	79	926	Next Year	22-Jun-2000
134	1504	28	981	Birdsong	18-Sep-2001
138	1509	84	957	Quicktime	18-Aug-1999
140	1478	17	963	Vanish!	18-Aug-1999
141	1478	84	941	Mahler 1	21-Jul-2000
143	1504	79	927	Chanson	21-Nov-2002
147	1076	17	952	Gold Days	30-Apr-2000
147	1409	79	988	Secret Tape	17-Apr-2000
152	1428	59	978	Swift Step	01-Oct-2001

Motivation for normalising beyond third normal form

Why go beyond third normal form?

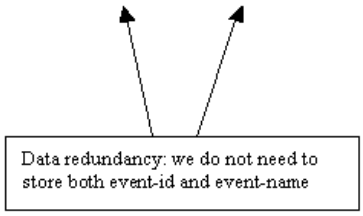
As we shall explore in this section, under certain circumstances there are anomalies that can occur for data that meets all the requirements for third normal form. Once these anomalies were identified and understood, database researchers developed the further normal forms we shall explore in this chapter.

Insertion anomalies of third normal form

There are no true insertion anomalies in the Bookings relation in third normal form; the details about each performer, agent, venue and event are also held in separate relations specifically for those entities, but there is data redundancy.

Relation in third normal form: Bookings

Performer-id	Agent-id	Venue-id	Event-id	Event-name	BookingDate
101	1295	59	959	Show Time	25-Nov-1999
105	1435	35	921	Silver Shoe	07-Jan-2002
105	1504	54	942	White Lace	10-Feb-2002
108	1682	79	901	The Dark	29-Jul-2003
112	1460	17	926	Next Year	13-Aug-2000
112	1522	46	952	Gold Days	05-May-1999
112	1504	75	952	Gold Days	16-Mar-1999
126	1509	59	945	Trick-Treat	02-Sep-2001
129	1478	79	926	Next Year	22-Jun-2000
134	1504	28	981	Birdsong	18-Sep-2001
138	1509	84	957	Quicktime	18-Aug-1999
140	1478	17	963	Vanish!	18-Aug-1999
141	1478	84	941	Mahler 1	21-Jul-2000
143	1504	79	927	Chanson	21-Nov-2002
147	1076	17	952	Gold Days	30-Apr-2000
147	1409	79	988	Secret Tape	17-Apr-2000
152	1428	59	978	Swift Step	01-Oct-2001



We can see that there is data redundancy in the Bookings relation, as every time a particular event is involved in a booking, both the event-id and the event-name need to be inserted into the Bookings relation.

Strictly speaking, we do not need to have both event-id and event-name in the Bookings relation, as each determines the other. If a mistake were to be made while inserting a new tuple, so that the event-id and the event-name did not match, this would cause problems of inconsistency within the database. The solution is to decide on one of the two determinants from the Events relation as part of the composite key for the Bookings relation.

We have noted that the event-id and the event-name determine each other within the Events relation, and this in turn creates overlapping keys in the Bookings relation. If the relationship between event-id and event-name were to break down, and a new event happened to have the same name as another event with a different event-id, this could create problems in the Bookings relation.

Performer Scenario 2

We can refer to a slightly altered database design as 'Performer Scenario 2', in

order to demonstrate the effects of overlapping keys.

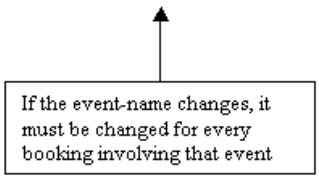
An issue that we need to examine is in the context of this slightly different database. In the example we having been using, the Events relation contains event-id, event-name and event-type. We can see that the performer-type in the Performers relation matches the event-type in the Events relation (e.g. actors performing in dramas, singers performing in musicals). If we now consider that the database holds only event-id and event-name as details about each event, this would affect the structure of the database.

Now, if we were to attempt to insert details about an event which had not yet been booked, we would not be able to do so as we would have an incomplete key in the Bookings relation. An event which has not been booked would have an event-id and an event-name, but no other attributes would have a value as there has been no booking.

Amendment anomalies of third normal form

If there were a change to the name of a particular event, this would need to be reflected in every booking involving that event. Some events may be booked many times, and if the change to the name of an event is not updated in each case, we would again find problems with maintaining consistent information in the database.

Performer-id	Agent-id	Venue-id	Event-id	Event-name	Booking Date
112	1522	46	952	Gold Days	05-May-1999
112	1504	75	952	Gold Days	16-Mar-1999
147	1076	17	952	Gold Days	30-Apr-2000



Here too, the solution is to identify either event-id or event-name as the determinant from the Events relation, so that the other of these two attributes is stored once only in the Events relation.

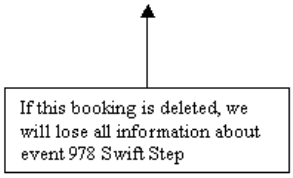
Deletion anomalies of third normal form

There are no deletion anomalies in the example we have been using. If we consider Scenario 2, however, we will find that deletion anomalies do exist.

Performer Scenario 2

We know that in Performer Scenario 2, there is no separate Events relation. If a booking is cancelled, we will want to delete the relevant tuple from the Bookings relation. This means that if we delete a tuple which contained details of an event that had no other booking, we would lose all information about that event.

Performer-id	Agent-id	Venue-id	Event-id	Event-name	BookingDate
152	1428	59	978	Swift Step	01-Oct-2001



Boyce-Codd and fourth normal form

Beyond third normal form

In this section we introduce two new normal forms that are more 'strict' than third normal form. For historical reasons, the simple numbering of first, second and third deviates before getting to fourth. The two new normal forms are called:

- Boyce-Codd normal form
- Fourth normal form

Boyce-Codd normal form

When it comes to identifying the booking, there is an ambiguity, as the booking details could be identified by more than one combination of attributes.

As it is possible to identify details of each event either by the event-id or by the event-name, there are two possible groupings of attributes that could be used to identify a booking: performer-id, agent-id, venue-id and event-id, or performer-id, agent-id, venue-id and event-name.

Important

Boyce-Codd normal form (BCNF)

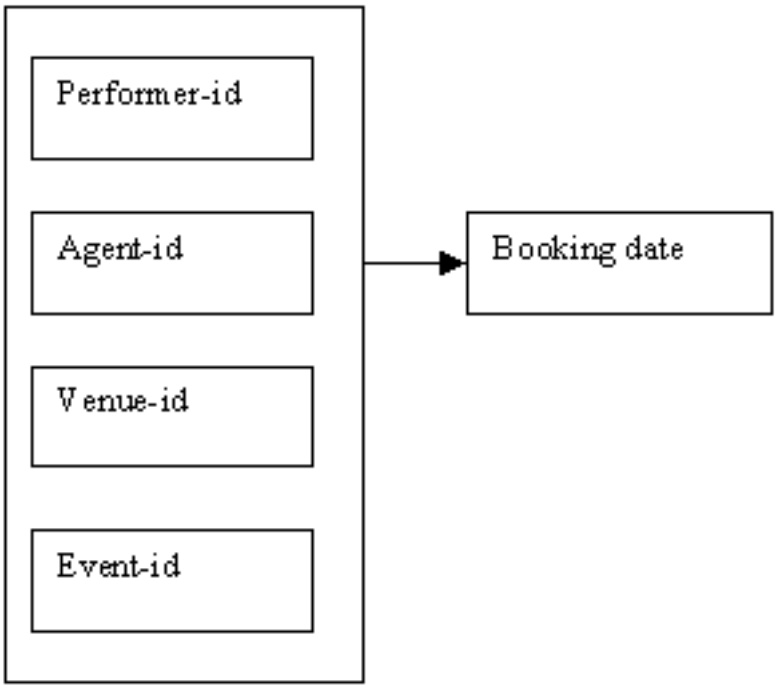
A relation is in Boyce-Codd normal form if all attributes which are determinants are also candidate keys.

Boyce-Codd normal form is stronger than third normal form, and is sometimes known as strong third normal form.

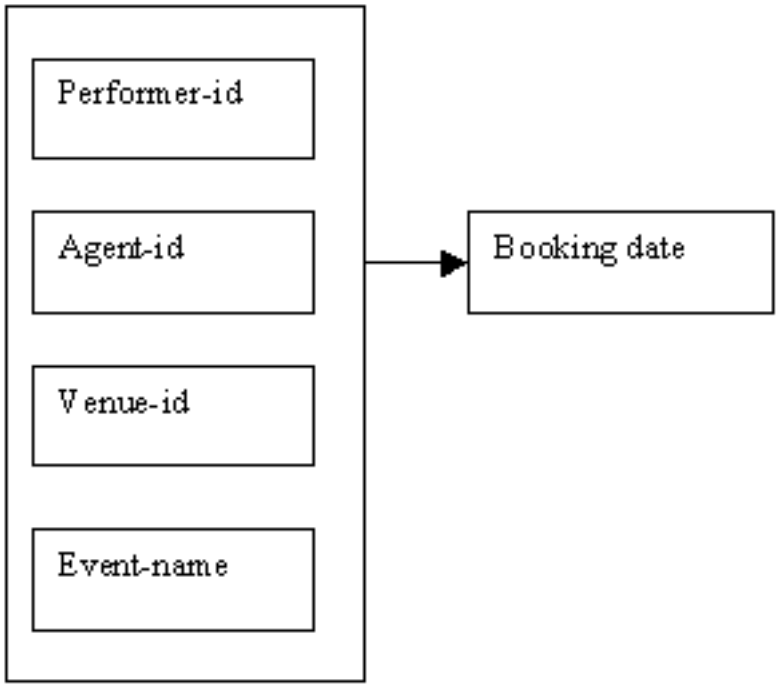
Transformation into Boyce-Codd normal form deals with the problem of overlapping keys.

An indirect dependency is resolved by creating a new relation for each entity; these new relations contain the transitively dependent attributes together with the primary key.

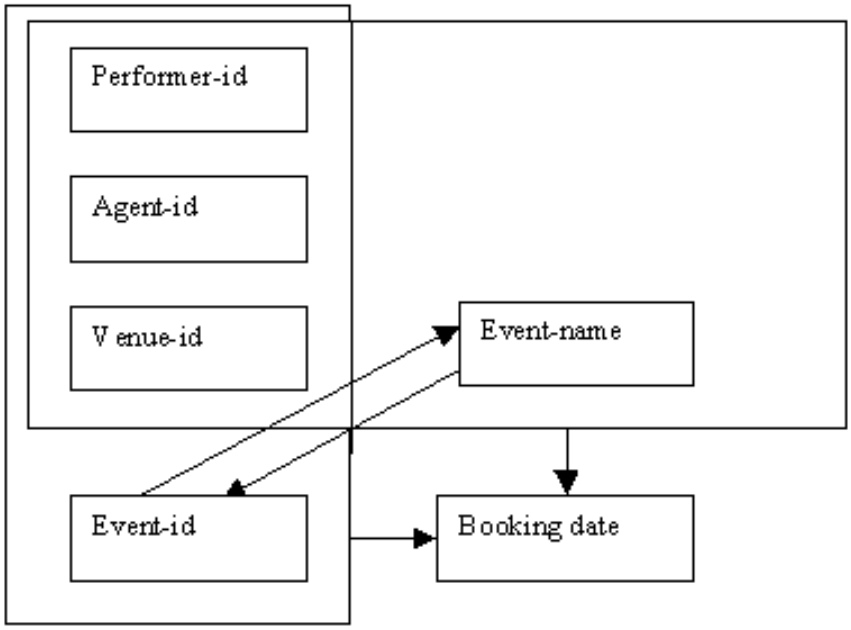
We know that we can identify a booking by means of the attributes performer-id, agent-id, venue-id and event-id, as shown in the determinacy diagram below.



We also know that we can identify a booking by using the attributes performer-id, agent-id, venue-id and event-name, shown in the next determinacy diagram.



When we combine the two determinacy diagrams shown above, we can see that we have an example of overlapping keys:



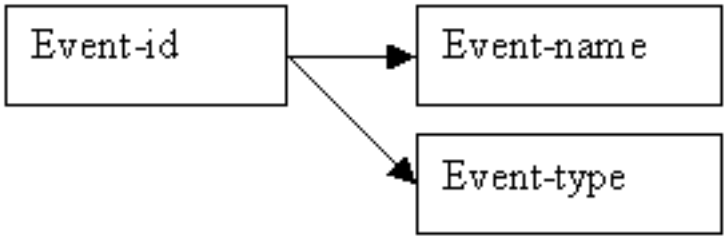
The details of the Bookings relation are shown later in this section.

Although overlapping keys are rare in practice (and some examples may appear rather contrived), we need to be aware that they can occur, and how to deal with them. The solution is simple: we need to decide on a single choice of attributes so that we have only one primary key. We know that event-name would not be an ideal choice of primary key. This is because it can be difficult to get names exactly right (e.g. "Quicktime" is not identical to "Quick Time"), and it may be coincidence rather than a rule that there is a one-to-one relationship between event-id and event-name (the relationship might break down). The choice of attribute to appear in the primary key is therefore event-id rather than event-name.

In Boyce-Codd normal form, we have six relations: Performers, Fees, Agents, Venues, Events and Bookings. The structure of the determinacy diagrams and content of the relations for Performers, Fees, Agents and Venues remain unchanged from third normal forms, and are not repeated here. There are changes to the Events and Bookings relations, which are illustrated below. A summary of the determinacy diagrams and the relations for this example are given in the 'Summary of normalisation rules' section of this chapter, together with an entity-relationship diagram.

Event details The choice of event-id as the primary key for the Bookings relation means that we can show the simpler representation of the determinacy diagram for event details, as we no longer have to consider the attribute event-

name as a possible key.

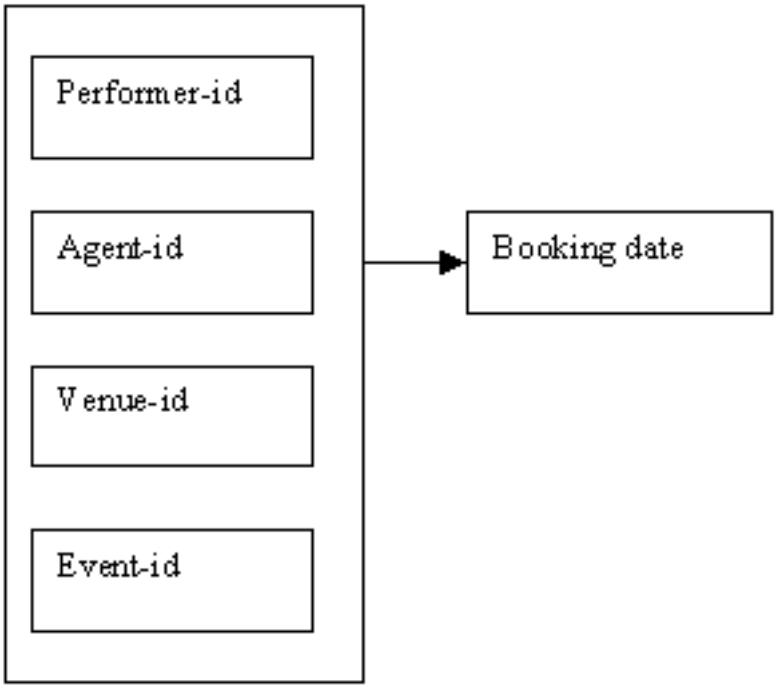


Event-id	Event-name	Event-type
901	The Dark	Drama
907	Elgar 1	Concert
913	What Now?	Drama
921	Silver Shoe	Ballet
926	Next Year	Drama
927	Chanson	Opera
934	Angels	Opera
938	New Dawn	Drama
941	Mahler 1	Concert
942	White Lace	Ballet
945	Trick-Treat	Variety show
952	Gold Days	Drama
957	Quicktime	Musical
959	Show Time	Musical
963	Vanish!	Magic show
964	The Friends	Drama
971	Card Trick	Magic show
978	Swift Step	Dance
981	Birdsong	Musical
988	Secret Tape	Drama

Note that in Scenario 2, where there was no separate Events relation, it would now be necessary to create an Events relation in order to transform the Bookings relation from third normal form into Boyce-Codd normal form.

Booking details Now that we have decided that event-id is the more suitable attribute for use as part of the key for the Bookings relation, we no longer need to store the event-name, which is already held in the Events relation. The problem of the overlapping keys has now been resolved, and the key for the Bookings relation is the combination of the attributes performer-id, agent-id,

venue-id and event-id.



The Bookings relation no longer needs to hold the attribute event-name, as this is already held in the Events relation.

Relation in Boyce-Codd normal form: Bookings

Performer-id	Agent-id	Venue-id	Event-id	BookingDate
101	1295	59	959	25-Nov-1999
105	1435	35	921	07-Jan-2002
105	1504	54	942	10-Feb-2002
108	1682	79	901	29-Jul-2003
112	1460	17	926	13-Aug-2000
112	1522	46	952	05-May-1999
112	1504	75	952	16-Mar-1999
126	1509	59	945	02-Sep-2001
129	1478	79	926	22-Jun-2000
134	1504	28	981	18-Sep-2001
138	1509	84	957	18-Aug-1999
140	1478	17	963	18-Aug-1999
141	1478	84	941	21-Jul-2000
143	1504	79	927	21-Nov-2002
147	1076	17	952	30-Apr-2000
147	1409	79	988	17-Apr-2000
152	1428	59	978	01-Oct-2001

Exercise 1

Define Boyce-Codd normal form.

Fourth normal form

The normalisations process so far has produced a set of five relations, which are robust against insertion, amendment and deletion anomalies. If at this stage it were decided to introduce further details into the relations, it would still be possible to do so. Database designers and developers would be well advised to start again with the normalisations process if changes are proposed to the data. However, for this example, we will introduce some new information that only affects the performers.

We are now required to add further details to the Performers relation, to show their knowledge and skills in two other areas: languages and hobbies.

Important

Fourth normal form (4NF) A relation is in fourth normal form if there are no multi-valued dependencies between the attributes.

Multi-valued dependencies occur where there are a number of attributes that depend only on the primary key, but exist independently of each other.

The representation of languages spoken and hobbies would be a simple enough requirement if each performer spoke exactly one language and had only one hobby. However, our performers are multi-talented, and some speak many languages, and others have several hobbies. Furthermore, the languages and hobbies are not related to each other. This presents us with a problem: how can we represent this in the relation? We know we cannot have a group of items under the headings Languages and Hobbies, as this would contravene the rules for first normal form.

The relation below is an attempt at representing some of this information, using a small number of performers as an example.

Relation: Some-Performers-Example 1

Relation: Some-Performers-Example 1

Perf-id	Perf-name	Perf-code	Performer-location	Languages	Hobbies
101	Baron	0348	York	French	
101	Baron	0348	York	English	
101	Baron	0348	York	Italian	
101	Baron	0348	York		Art
105	Steed	0862	Berlin	German	
105	Steed	0862	Berlin	English	
105	Steed	0862	Berlin	Mandarin	
105	Steed	0862	Berlin		Music
105	Steed	0862	Berlin		Poetry
108	Jones	0729	Bombay	Cantonese	

If we look at this relation, while it conforms to the rules for first normal form (there are no repeating groups), there is still some ambiguity in its meaning. If we look at Baron's hobbies, we can see that 'art' has been identified, but that there is no entry for the attribute 'language'. Does this mean that Baron does not speak any other languages? We know this is not true, because there are other entries that demonstrate that Baron speaks three languages. If we take the alternative view, and look at another entry for Baron, we can see that Baron speaks Italian, but from this entry it could appear that Baron has no hobbies. This approach is not the solution to the problem.

Another attempted solution pairs languages and hobbies together, but sometimes there is a language but no hobby (or the other way around).

Relation: Some-Performers-Example 2

Relation: Some-Performers-Example 2

Perf-id	Perf-name	Perf-code	Performer-location	Languages	Hobbies
101	Baron	0348	York	French	Art
101	Baron	0348	York	English	
101	Baron	0348	York	Italian	
105	Steed	0862	Berlin	German	Music
105	Steed	0862	Berlin	English	Poetry
105	Steed	0862	Berlin	Mandarin	
108	Jones	0729	Bombay	Cantonese	

In this new approach, we have entered each hobby against a language. However, we are still faced with problems. If Steed decides to give up poetry as a hobby, we will lose the information that Steed speaks English. If Baron's French gets 'rusty' and is deleted from the relation, we will lose the information that Baron's hobby is art.

Relation: Some-Performers-Example 3

Relation: Some-Performers-Example 3

Perf-id	Perf-name	Perf-code	Performer-location	Languages	Hobbies
101	Baron	0348	York	French	Art
101	Baron	0348	York	English	Art
101	Baron	0348	York	Italian	Art
105	Steed	0862	Berlin	German	Music
105	Steed	0862	Berlin	German	Poetry
105	Steed	0862	Berlin	English	Music
105	Steed	0862	Berlin	English	Poetry
105	Steed	0862	Berlin	Mandarin	Music
105	Steed	0862	Berlin	Mandarin	Poetry
108	Jones	0729	Bombay	Cantonese	

In this next attempt, all languages are paired with all hobbies; this means that there is a great amount of redundancy, as basic data about the performers is repeated each time. We have a problem with Jones, who does not appear to have a hobby, which questions whether this entry is valid. In addition, if Steed learns a new language, it would be necessary to repeat this new language paired with Steed's existing hobbies. This option is also an unsatisfactory method of solution.

The solution to this problem is to divide the information that we are trying to represent into a group of new relations; one containing the basic performer information as before, another showing details of languages spoken, and a third maintaining a record of hobbies.

This transformation deals with the problems of multi-valued facts and associated redundancies in the data; we can now convert the relation into three relations

in fourth normal form.

Naturally, the new relations would hold data for all the performers, although only an extract from each relation is given here.

Relation in fourth normal form: Some-Performers

Perf-id	Perf-name	Perf-code	Performer-location
101	Baron	0348	York
105	Steed	0862	Berlin
108	Jones	0729	Bombay

Relation in fourth normal form: Some-Performers-Languages

Perf-id	Languages
101	French
101	English
101	Italian
105	German
105	English
105	Mandarin
108	Cantonese

Relation in fourth normal form: Some-Performers-Hobbies

Perf-id	Hobbies
101	Art
105	Music
105	Poetry

Exercise 2

Define fourth normal form.

Summary of normalisation rules

The rules used for converting a group of data items which are un-normalised into a collection of normalised relations are summarised in the table below. Remember that in some cases we might choose not to normalise the data completely, if this would lead to inefficiency in the database.

The conversion from fourth normal form to fifth normal form is included for completeness. We will not be examining the definition of fifth normal form in detail; it is concerned with avoiding unnecessary duplication of tuples when new relations are created by joining together existing relations. The cause of this problem is the existence of interdependent multi-valued data items.

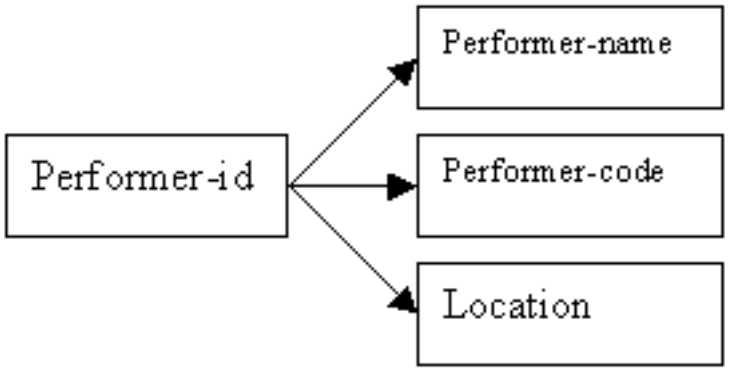
Starting with normal form	Convert to normal form	Abbreviation	Use the rule
Un-normalized data	First	1NF	remove repeating groups of data within a single tuple into new tuples with only one data value for each attribute
First	Second	2NF	Extract non-key partial dependencies (items not fully functionally dependent on the key) into a separate relation
Second	Third	3NF	remove transitive dependencies into a separate relation
Third	Boyce-Codd	BCNF	remove overlapping keys by identifying a single primary key and holding other values in a separate relation
Third Or Boyce-Codd	Fourth	4NF	remove multi-valued dependencies by extracting independent data into a separate relation
Fourth	Fifth	5NF	remove join dependencies caused by interdependent data

Fully normalised relations

We now have five relations which are fully normalised, and can be represented by means of determinacy diagrams, relations, and an entity-relationship diagram. Each relation has a corresponding entity in the entity-relationship diagram.

Performer details

All performers appear in the Performers relation. The primary key is performer-id, and the other attributes are performer-name, performer-code (which identifies the performer-type) and performer-location.

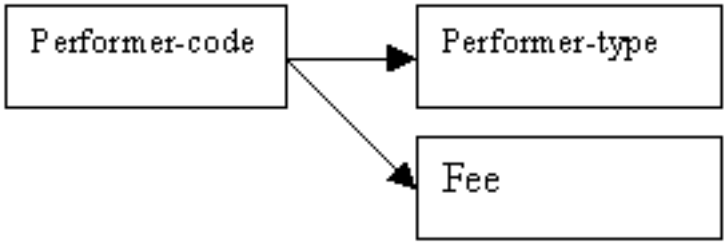


Fully normalised relation: Performers

Performer-id	Performer-name	Performer-code	Performer-location
101	Baron	0348	York
105	Steed	0862	Berlin
108	Jones	0729	Bombay
112	Eagles	0729	Leeds
118	Markov	0862	Moscow
126	Stokes	0244	Athens
129	Chong	0729	Beijing
134	Brass	0348	London
138	Ng	0348	Penang
140	Strong	0360	Rome
141	Gomez	0915	Lisbon
143	Tan	0348	Chicago
147	Qureshi	0729	London
149	Tan	0729	Taipei
150	Pointer	0360	Paris
152	Peel	0862	London

Fee details

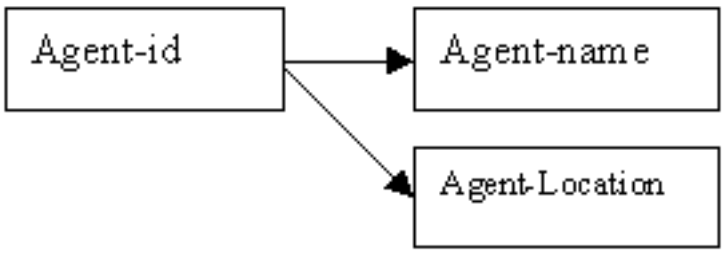
Each performer is paid a fee depending on the performer-type. The rates of pay for each performer-type are stored in the Fees relation, together with a performer-code, which is the primary key.



Performer-code	Performer-type	Fee
0348	Singer	75
0862	Dancer	60
0729	Actor	85
0244	Comedian	90
0360	Magician	84
0915	Musician	92

Agent details

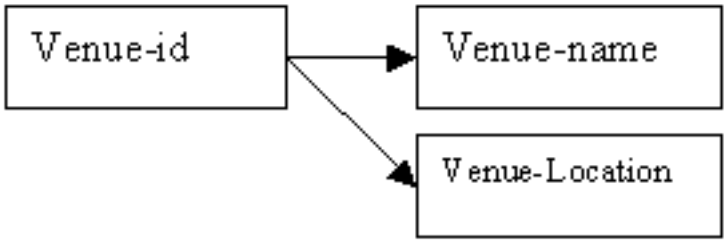
All agents are recorded in the Agents relation, where the primary key is agent-id, and the remaining attributes are agent-name and agent-location.



Agent-id	Agent-name	Agent-location
1295	Burton	Luton
1435	Nunn	Boston
1504	Lee	Taipei
1682	Tsang	Beijing
1460	Stritch	Rome
1522	Ellis	Madrid
1478	Burns	Leeds
1377	Webb	Sydney
1509	Patel	York
1190	Patel	Hue
1802	Chapel	Bristol
1076	Eccles	Oxford
1409	Arkley	York
1428	Vernon	Cairo

Venue details

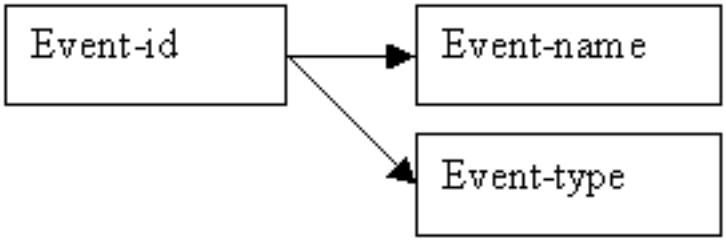
There are a number of venues available for bookings, and these are stored in the Venues relation. The primary key is venue-id, and the other attributes are venue-name and venue-location.



Venue-id	Venue-name	Venue-location
59	Atlas	Tokyo
35	Polis	Athens
54	Nation	Lisbon
79	Festive	Rome
46	Royale	Cairo
28	Gratton	Boston
75	Vostok	Kiev
84	State	Kiev
82	Tower	Lima
17	Silbury	Tunis
92	Palace	Milan
62	Shaw	Oxford

Event details

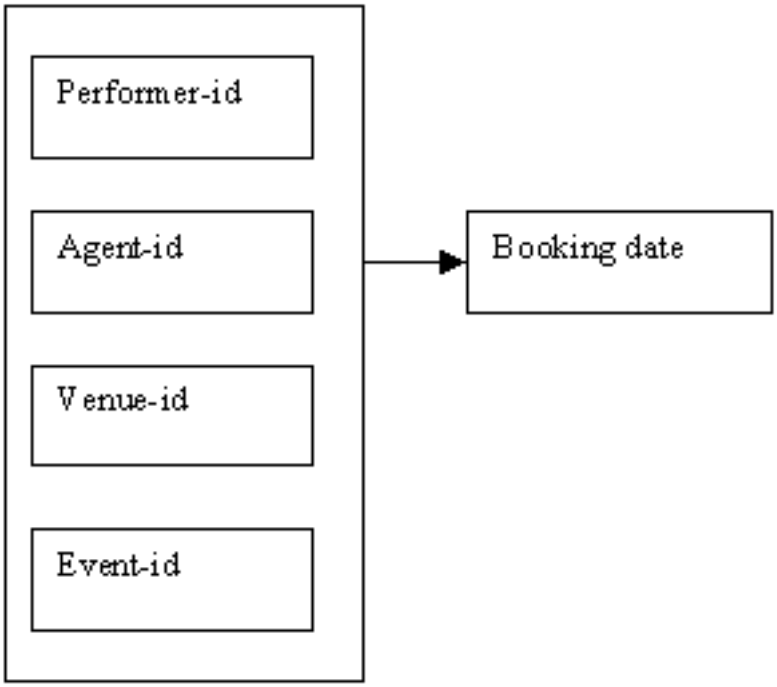
All events which can be booked are listed in the Events relation; the primary key is event-id, and the other attributes are event-name and event-type.



Event-id	Event-name	Event-type
901	The Dark	Drama
907	Elgar 1	Concert
913	What Now?	Drama
921	Silver Shoe	Ballet
926	Next Year	Drama
927	Chanson	Opera
934	Angels	Opera
938	New Dawn	Drama
941	Mahler 1	Concert
942	White Lace	Ballet
945	Trick-Treat	Variety show
952	Gold Days	Drama
957	Quicktime	Musical
959	Show Time	Musical
963	Vanish!	Magic show
964	The Friends	Drama
971	Card Trick	Magic show
978	Swift Step	Dance
981	Birdsong	Musical
988	Secret Tape	Drama

Booking details

Every booking made by an agent, for a performer, at a venue, for an event, is stored in the Bookings relation. The primary key is a combination of performer-id, agent-id, venue-id and event-id; the remaining attribute is booking date.


Performer-id
Agent-id
Venue-id
Event-id
Booking date

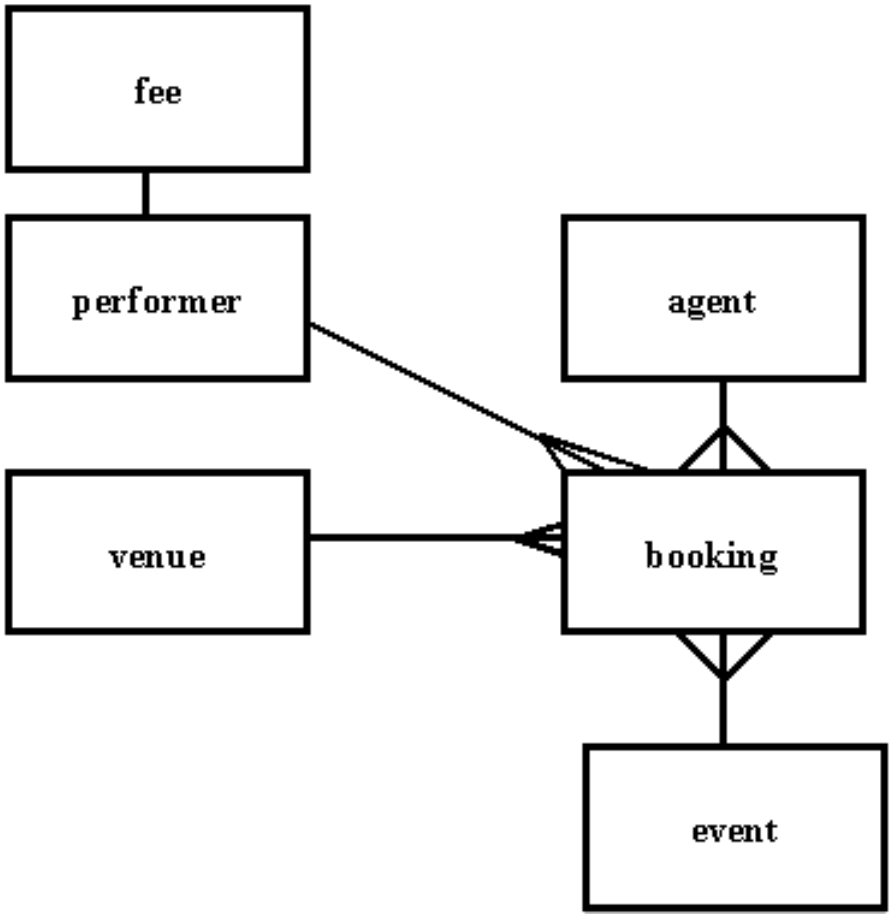
Performer-id	Agent-id	Venue-id	Event-id	Booking date
101	1295	59	959	25-Nov-1999
105	1435	35	921	07-Jan-2002
105	1504	54	942	10-Feb-2002
108	1682	79	901	29-Jul-2003
112	1460	17	926	13-Aug-2000
112	1522	46	952	05-May-1999
112	1504	75	952	16-Mar-1999
126	1509	59	945	02-Sep-2001
129	1478	79	926	22-Jun-2000
134	1504	28	981	18-Sep-2001
138	1509	84	957	18-Aug-1999
140	1478	17	963	18-Aug-1999
141	1478	84	941	21-Jul-2000
143	1504	79	927	21-Nov-2002
147	1076	17	952	30-Apr-2000
147	1409	79	988	17-Apr-2000
152	1428	59	978	01-Oct-2001

Note that this assumes there can only be one booking involving a particular combination of performer, agent, venue and event. This means that we cannot have multiple bookings made involving the same performer, agent, venue and event, as the primary key would be the same for each booking and we would therefore lose unique identification of bookings.

In order to accommodate multiple bookings involving the same entities, we could include the booking date as part of the key, but then we would not be able to distinguish between morning and evening performances on the same date (unless we included time as well as date).

Entity-relationship diagram

The determinacy diagrams and relations, which are now fully normalised, can also be viewed as entities linked by relationships using the data modelling technique described in the chapter on entity-relationship modelling. Each determinacy diagram represents a relation, which in turn corresponds to an entity, as can be seen in the entity-relationship diagram below. The relationships that exist between each entity are summarised below the diagram.



This entity relationship diagram represents the following:

- Each performer may have many bookings, so the relationship between performer and booking is one-to-many.
- A performer earns a fee, the value of which depends on the performer type, so the relationship between performer and fee is one-to-one (a performer can only be of one type).
- An agent may make a number of bookings, so the relationship between agent and booking is one-to-many.
- Any venue may have been booked several times, which makes the relationship between venue and booking one-to-many.
- Each event may be involved in a number of bookings, so this relationship is also one-to-many.

- The relationships that exist between performers, agents, venues and events are shown by their connections through the bookings.

Exercise 3

Why have so many normal forms?

Further issues in decomposing relations

When moving to a higher normal form, we often have a choice about the way in which we can decompose a relation into a number of other relations. This section examines problems that can arise if the wrong decomposition is chosen.

As an example, supposing within a government department responsible for intelligence gathering, we wish to record details of employees in the department, and the levels of their security clearance, which describe the levels of access employees have to secret information. The table might contain the following attributes:

Relation EMPLOYEE (Employee, Security_code, Level)

Where Employee is the primary key and provides some convenient means of identifying each employee, Security_code identifies the security clearance of that employee, and Level identifies the level of access to secret information possessed by anyone having that Security_code.

The determinants in this relation are:

Employee determines Security_code

Security_code determines Level

So we have two functional dependencies, respectively Security_code is functionally dependent on Employee, and Level is functionally dependent on Security_code. We also have a transitive, or indirect dependency, of Level on Employee; that is, an employee's level of security clearance does depend on who that employee is, but only via the value of their Security_code.

This relation is in second normal form; i.e. it contains no repeating groups and no part-key dependencies, but it does contain a transitive dependency.

In order to convert relation EMPLOYEE to third normal form, we need to decompose it to remove the transitive dependency of Level on Employee. Until we make this decomposition, we have the following insert, update and deletion anomalies:

- We cannot create a new Security_code until we have an Employee to whom we wish to allocate it.

- If we change the Level of a Security_code, i.e. change the Level of information employees who hold that code can access, then in relation EMPLOYEE, we would have to propagate the update throughout all the employees who hold that Level.
- If we remove the last Employee holding a particular Security_code, we loose the information about the Level of clearance assigned to that Security_code.

To perform the decomposition, suppose we split relation EMPLOYEE into two relations as follows:

Decomposition A

Relation EMPLOYEE-CLEARANCE (Employee, Level)

Relation SECURITY_LEVEL (Security_code, Level)

There are problems with this decomposition. Supposing we wish to change the security clearance for a given Employee. We can change the value of Level in relation SECURITY_CLEARANCE, but unfortunately, this update is not independent of the data held in relation SECURITY_LEVEL. In order for the change to have taken place in the SECURITY-CLEARANCE relation, one of two things must have arisen. Either the Employee in question has changed his/her Security_code, in which case no update need be made to relation SECURITY_LEVEL, or the Level associated with the Security_code possessed by the Employee has changed, in which case relation SECURITY_LEVEL must be changed to reflect this.

The problem has arisen because the two relations in decomposition A are not independent of one another. There is in fact a functional dependency between them: the fact that Security_code is functionally dependent on Employee. In decomposition A, instead of storing in the same relation those data items that are functionally dependent on one another, we have split the functional dependency of Security_code on Employee across the two relations, preserving the transitive dependency of Level on Employee in relation SECURITY_CLEARANCE. The problems this gives is that we cannot then make updates to one of these relations without considering whether updates are required to the other. As a further example, if we make updates to relation SECURITY_LEVEL, changing the Level of access associated with each Security_code, we must make sure that these updates are propagated to relation SECURITY_CLEARANCE, i.e. that the employees who possess the altered security codes have their Level attribute updated to reflect the changes in the SECURITY_LEVEL relation.

Resolution of the problem

The solution to this problem is to ensure that when making decompositions, we preserve the functional dependencies of data items within the resulting rela-

tions, rather than splitting them between the different relations. For the above example, the correct decomposition would therefore be as follows:

Decomposition B

Relation EMPLOYEE_CODE (Employee, Security_code)

Relation ACCESS_LEVEL (Security_code, Level)

This decomposition allows us to manipulate the level of security granted to an individual employee (in relation EMPLOYEE_CODE) independently of that which specifies in general the level of access associated with security codes (maintained in relation ACCESS_LEVEL).

Denormalisation and over-normalisation

Denormalisation

As the normalisation process progresses, the number of relations required to represent the data of the application being normalised increases. This can lead to performance problems when it comes to performing queries and updates on the implemented system, because the increased number of tables require multiple JOINs to combine data from different tables. These performance problems can be a major issue in larger applications (by larger we mean both in terms of numbers of tables and quantity of data).

To avoid these performance problems, it is often decided not to normalise an application all the way to fourth normal form, or, in the case of an existing application which is performing slowly, to denormalise an existing application. The process of denormalisation consists of reversing (or in the case of a new application, not carrying out in the first place) the steps to fully normalise an application. Whether this is appropriate for any given application depends critically on two factors:

- Whether the size of the application is sufficient that it will run slowly on the hardware/software platform being used.
- Whether failing to carry out certain steps in the normalisation process will compromise the requirements of the applications users.

Supposing, for example, we have a relation in which we wish to store the details of companies, the departments making up the companies and the locations of the departments. We might describe such a relation as follows:

Relation COMPANY (Company, Department, Location)

A row in the relation indicates that a particular Department of a specific Company is based at a particular Location. The primary key of relation COMPANY is the attribute Company. Assuming that Location depends directly on the Department of any particular Company, there is a transitive dependency between

Company and Location, via Department. To convert relation COMPANY to third normal form, we would decompose it into:

Relation DEPARTMENT (Company, Department)

Relation LOCATION (Department, Location)

This would avoid the insert, update and deletion anomalies associated with second normal form relations, giving us the ability to manipulate the information about which departments make up a particular company quite independently of the information about where particular departments are located. This is the additional flexibility provided by taking the step of converting the application to third normal form. However, if we wish to store information about a large number of companies and departments, and we will not need to manipulate the location information about departments independently of the information about which departments make up a company, then we may choose not to proceed to third normal form, but to leave relation COMPANY in second normal form. Thus we'd retain the Company, Department and Location attributes in one relation, where they can be queried and manipulated together, without the need for JOINs.

If we choose to leave relation COMPANY in second normal form, what we will have lost in terms of flexibility is as follows:

- The ability to create new departments at specific locations without allocating them to a specific company.
- The ability to create new departments at specific locations without allocating them to a specific company.

Note that in this particular case, the update anomaly does not arise, as each department is assumed to appear only once in relation COMPANY. Users of the application may feel that the flexibility provided by third normal form is simply not required in this application, in which case we can opt for the second normal form design, with its improved performance.

The same arguments apply when considering whether to take any steps that lead to a higher normal form; there is always a trade-off similar to the above, between the increased flexibility of a more normalised design versus a faster-running application in a less normalised design, due to the smaller number of relations. When Relational database systems first arrived in the early '80s, their performance was generally slow, and this had an influence in slowing down their adoption by some companies. Since that time, a huge amount of research and development has gone into improving the performance of Relational systems. This development work, plus the considerable improvements in the processing speed of hardware, tends to suggest that the need to denormalise applications should be reduced; however, it remains an important option for application designers seeking to develop well-tuned applications.

Over-normalisation

A further technique for improving the performance response of database applications, is that of over-normalisation. This technique is so-called because it results in a further decomposition of the relations of an application, but for different reasons than that of the usual normalisation process. In normalisation, we are seeking to satisfy user requirements for improved application flexibility, and to eliminate data redundancy. In contrast, the decompositions made during over-normalisation are generally done so to improve application performance.

As an example, we shall take the case of a company possessing a large table of customer information, supposing the table contains several thousand rows, and that the customers are more or less equally divided into those based in the home country of the company and overseas.

There are essentially two approaches to over-normalisation of a table — we can divide it up either horizontally or vertically.

Splitting a table horizontally

The most common approach is to split a table horizontally. In the case of the large customer table, we might split it into two tables, one for home-based customers, and the other for overseas customers.

Splitting a table vertically

The alternative approach of vertical partitioning might be used if the columns of a table fell naturally into two or more logical subsets of information; for example, if several columns of the customer table contained data specific to credit-limit assessment, whereas others contained more general contact and customer-profiling information. If this were the case, we might split the table vertically, one partition containing credit-limit assessment information, and the other containing the more general customer details. It is important when performing vertical partitioning in this way that the primary key of the entity involved, in this case, Customer, is retained in both partitions, enabling all of the data for the same entity instance (here for a specific customer) to be re-assembled through a JOIN.

Example of over-normalisation

So, for the process of over-normalisation, tables can be split into a number of horizontal or vertical fragments. For example, if the customers in the above example contained a 'region' attribute, which indicated in which region of the world they are based, then rather than a simple dual split into home-based and overseas customers, we might create a separate partition for each regional grouping of customers.

There are a number of reasons why relations may be fragmented in this way, most of which are directly concerned with improving performance. These objectives are briefly examined below:

- Splitting a large table into a number of smaller tables often reduces the number of rows or columns that need to be scanned by specific query or update transactions for an application. This is particularly true when the partitions created are a good match to different business functions of the application - for example, in the customer table example above, if customers in different regions of the world undergo different types of query and update processing.
- The smaller tables retrieved by queries restricted to one, or even a few, of a number of partitions will take up less space in main memory than the large number of rows fetched by a query on a large table. This often means that the small quantity of data is able to remain in memory, available for further processing if required, rather than being swapped back to disk, as would be likely to happen with a larger data set.

Just as a good match to business functions for the chosen partitions means the over-normalisation process will work well in improving performance, a poor match to transaction requirements could lead to a poorer performance, because the over-normalised design could lead to an increased number of JOINs.

Review questions

Review question 1

Consider the following scenario.

A database is being designed to store details of a hospital's clinics and the doctors who work in them. Each doctor is associated with just one hospital. Each clinic has a two-to-three-hour session during which a doctor who is a specialist in a particular field of medicine, sees patients with problems in that specialist area; for example, a diabetic clinic would be run by a doctor who is a specialist in diabetes. The same clinic may occur in a number of different hospitals; for example, several hospitals may run a diabetes clinic. Doctors may hold a number of different clinics. Clinic within the same hospital are always held by the same doctor. The relation is therefore of the following form:

Relation CLINIC (Hospital, Clinic, Doctor)

A row in the relation signifies that a particular clinic is held by a particular doctor in a specific hospital.

1. What are the determinants in the above relation?
2. Demonstrate that relation CLINIC is not in BCNF. Which normal form is it in?

3. Explain any insertion, update or deletion problems that might arise with relation CLINIC. How might these be resolved?

Review question 2

A library holds a database of the loans and services used by its members. Each member may borrow up to 10 books at a time, and may reserve sessions making use of library facilities, such as time on an Internet PC, a Multimedia PC, booking a ticket for a performance at the library theatre, etc.

Describe how the above scenario could be handled in the process of normalisation.

Review question 3

It is required to develop a database which can provide information about soccer teams: the number of games they have played, won, drawn and lost, and their current position in the league.

Write down your thoughts on the issues involved in supporting the requirement to provide the current league position, and how this is best satisfied.

Review question 4

If BCNF is a stronger definition of 3NF, and provides a more concise definition of the normalisation process, why is it worth understanding the step-by-step processes of moving from an un-normalised design to 3NF? Why is BCNF a stronger normal form than 3NF?

Review question 5

A binary relation is a relation containing just two attributes. Is it true that any binary relation must be in BCNF?

Review question 6

What is the difference between a repeating group and a multi-valued dependency?

Review question 7

True or false: Splitting a functional dependency between two relations when decomposing to a higher normal form is to be preferred to splitting a transitive dependency. Give reasons to justify your assertion.

Review question 8

Explain the difference between denormalisation and over-normalisation. What do the two techniques have in common, and what differences do they have?

Review question 9

A chemical plant produces chemical products in batches. Raw materials are fed through various chemical processes called a production run, which turns the raw materials into a final product. Each batch has a unique number, as

does each product produced by the plant. We can also assume that product names are unique. Each production run results in the production of a quantity of a particular product. We assume that only one product is produced in any given production run, and so different production runs are required for different products. The design of a relation to store the details of production runs could look as follows:

Relation PRODUCTION_RUN (Product_no, Product_name, Batch_no, Quantity)

In which normal form is relation PRODUCTION_RUN? Explain the reasoning behind your assertion.

Resolve any anomalies that could arise in the manipulation of rows in the PRODUCTION_RUN relation.

Review question 10

A company wishes to store details of its employees, their qualifications and hobbies. Each employee has a number of qualifications, and independently of these, a number of hobbies.

Produce a normalised design for storing this information.

Review question 11

We saw earlier the issues surrounding storing some types of derived data; for example, the league position of soccer teams. Supposing we wish to store the number of points accumulated by such teams, given the rules that:

- A win is awarded 3 points
- A draw is awarded 1 point
- There are no points for a defeat

Consider any problems that might be associated with the storage of the number of points obtained by teams in such a league.

Discussion topic

Normalisation has been the second major technique we have examined for use in the design of database applications, the other being entity-relationship modelling. You are encouraged to discuss your feelings about the relative usefulness of these two approaches with respect to the following:

- Learnability. Which of the two techniques have you found easier to learn? Do not settle for merely identifying which technique was easier to learn, but examine what it is about the techniques that makes one or other of them easier to learn.

- Usability. Which of the techniques have you found easier to use so far in the chapters you have worked through, and which would you expect to be more useful in commercial application development? Be sure to back your assertions with an explanation of why you believe them to be true.

Finally, consider what you believe the relative strengths and weaknesses of the two design approaches to be, and consider to what extent these are or are not complementary.

Chapter 10. Declarative Constraints and Database Triggers

Table of contents

  - Calling a procedure from within a function and vice versa
- Discussion topics
- Additional content and activities

Objectives

At the end of this chapter you should be able to:

- Know how to capture a range of business rules and store them in a database using declarative constraints.
- Describe the use of database triggers in providing an automatic response to the occurrence of specific database events.
- Discuss the advantages and drawbacks of the use of database triggers in application development.
- Explain how stored procedures can be used to implement processing logic at the database level.

Introduction

In parallel with this chapter, you should read Chapter 8 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter introduces you to some of the most advanced features of Relational databases and SQL, namely declarative constraints, database triggers and stored procedures. These features have been made available in popular DBMSs such as Oracle. They provide those DBMSs with greater flexibility and power in dealing with complexities in many demanding business applications.

The reason for studying these advanced database features is that we need to address a growing trend of providing mechanisms for the processing as well as storage of data in database systems. Declarative constraints are a means of recording some types of business rules within a database system, and by doing so, have them systematically applied across all the applications operating on the database. Database triggers and stored procedures are additional mechanisms provided in some of the most powerful DBMSs (e.g. Oracle) for storing and applying logic at the database rather than application level.

The contents of this chapter are closely related to some of the others in this module. The distribution of processing in an application is an area of design that has developed with the evolution of client-server computing. A database designer now has choices about whether to place some aspects of business logic at the server (where the database resides), by having them built into the database system and enforced at that level, or at the client where it is enforced at the

application level. This chapter extends the SQL constructs studied in the chapter on Advanced SQL and discusses how business rules can be captured at the database level. Because of the design decisions that need to be made about the placing of business logic, this chapter also relates to the two on database design, and the chapter on distributed databases and client-server applications.

Because different DBMSs may implement those advanced features in different ways, our study will be focused on the related functionality provided by Oracle. Oracle PL/SQL statements will be used to provide examples to enable detailed discussions. Other DBMSs should provide similar functionalities, but you should consult with the system's documentation should you come across any incompatibilities. All SQL statements are in capital letters.

Context

For any complex database application, it is likely that there will be two or more tables involved to store information. It is also likely that data within the same table or in different tables will have to maintain some kind of relationship to reflect the corresponding business logic. In addition, some attributes (columns) of a table may need to have certain conditions imposed on them, and these conditions, which are often used to capture necessary business rules, need to be satisfied at all times.

In order to accommodate these practical needs of database applications, the SQL standard provides mechanisms to maintain the integrity of databases; that is, the integrity of data within a single table or in different tables as a whole. Declarative constraints are one of such mechanisms. They are used to define, according to the business application rules, conditions on columns and tables. Once defined (i.e. declared), these conditions will be enforced by the DBMS automatically.

As can be seen from the above description, the types of declarative constraints that can be declared are predefined by the DBMS, which conforms to the SQL standard. Usually they are used to store and enforce the kinds of business rules which are generally needed across different applications. Although they are simple to use and maintain, they lack some necessary flexibility and may not always be able to satisfy some specific needs of individual applications. To compensate for this, some DBMSs (e.g. Oracle) provide another type of mechanism to ensure database integrity: database triggers and stored procedures.

A procedure is a set of SQL or PL/SQL (in the case of Oracle) statements used together to execute a particular function. Database triggers are a mechanism that allows a database designer to write procedures that are automatically executed whenever a predefined situation (an event) is raised by the execution of INSERT, UPDATE or DELETE statements on a table or view. Because the database designer is responsible for creating triggers and writing procedures, he/she has an overall control. This control can be used to capture and build

business logic into the database as necessary. As a result, this mechanism offers greater flexibility and fewer restrictions for the designer to develop complex database applications. In short, database triggers and procedures are not only able to enforce integrity constraints, but can also be used to write customised functions to satisfy individual applications' needs.

In the rest of this chapter, we are going to study in detail declarative constraints, database triggers and procedures. We will see how they are used in practical applications, what the advantages and drawbacks are, and what the solutions are to potential problems.

To facilitate detailed discussions, suppose we need to implement a database for a university. The basic requirements state that there are four entities: STUDENT, MODULE, LECTURER and DEPT. A student can attend as many modules as necessary, and a module must be attended by at least one student. A module must be taught by one and only one lecturer, but a lecturer may teach between one and four modules. A student should be enrolled to a department; a module should be offered by one and only one department; a lecturer should belong to one and only one department.

It is not difficult to see that we will need to implement five tables: four tables for the four entities and one table (called RECORD) for the many-to-many relationship between STUDENT and MODULE.

- STUDENT (SID, SNAME, DNAME, SLEVEL, SEMAIL).
- LECTURER (EID, LNAME, LEMAIL, DNAME).
- MODULE (CODE, TITLE, EID, DNAME).
- DEPT (DNAME, LOCATION).
- RECORD (SID, CODE, MARK).

In the STUDENT table, SID is the student's identity number and the primary key, SNAME is the student's name, DNAME is the department to which the student has enrolled, SLEVEL is the level the student is at, and SEMAIL is the student's email address. In the LECTURER table, EID is the employee identity number for the lecturer and the primary key, LNAME is the lecturer's name, LEMAIL is the lecturer's email address and ENAME is the name of the department. In the MODULE table, CODE is the code of the module and the primary key, TITLE is the title of the module, EID is the name of the lecturer taking the module and DNAME is the name of the department the module belongs to. The DEPT table has only two attributes, department name DNAME (primary key) and location of the department in the university. In the RECORD table, SID is the student number, CODE is the code of the module and MARK is the mark a student obtained from attending a module. The SID and CODE makes a primary key.

Declarative constraints

Constraints are a mechanism provided within the DDL SQL standard to maintain the consistency and integrity of a database and, at the same time, enforce certain business rules in the database application. There are five different types of declarative constraints in SQL that can be defined on a database column within a table, and they are as follows:

- PRIMARY KEY
- NOT NULL
- UNIQUE
- CHECK
- FOREIGN KEY

The PRIMARY KEY constraint

The PRIMARY KEY constraint is used to maintain the so-called entity integrity. When such a constraint is declared on a column of a table, the DBMS enforces the following rules:

- The column value must be unique within the table.
- The value must exist for any tuple (a record or a row of data) that is to be stored in the table. That is, the column cannot have a NULL value.

For the STUDENT table in our university database, for example, we have SID as the key attribute. As a normal business rule, all students must have a valid and unique ID number as soon as they are enrolled. Thus, the SID column must have a unique value and cannot be null. To enforce this business rule, we can have the PRIMARY KEY constraint declared on the column when creating the STUDENT table. One way to do this is:

CREATE TABLE STUDENT (

SID NUMBER(5) CONSTRAINT PK_STUDENT PRIMARY KEY,

SNAME VARCHAR2(30),

DNAME VARCHAR2(30),

SLEVEL NUMBER(1),

SEMAIL VARCHAR2(40));

In the above SQL statement, the constraint is declared by using the keywords CONSTRAINT and PRIMARY KEY. A column definition clause with such a constraint declaration is called a column constraint clause. “PK_STUDENT” is a user-defined name for the constraint. It is optional, but when defined, it

can help the database designer and user to pinpoint a violation of this constraint. The reason is that when this particular constraint is violated, the DBMS will generate an error/warning message which includes the constraint's name. A usual convention for defining a PRIMARY KEY constraint's name is "PK_Table_Name".

There is an alternative way to declare the PRIMARY KEY constraint:

CREATE TABLE STUDENT (

SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30),

SLEVEL NUMBER(1),

SEMAIL VARCHAR2(40),

CONSTRAINT PK_STUDENT PRIMARY KEY (SID));

OR

CREATE TABLE STUDENT (

SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30),

SLEVEL NUMBER(1),

SEMAIL VARCHAR2(40),

PRIMARY KEY (SID));

In this SQL statement, a separate clause (called table constraint clause) is used to define the constraint. The column name (e.g. SID) must be explicitly stated in the list (i.e. within the brackets ( )). If the table has a composite key, then the list will include all the key attributes. For example, to create the RECORD table, we have:

CREATE TABLE RECORD (

SID NUMBER(5),

CODE VARCHAR2(6),

MARK NUMBER(3),

CONSTRAINT PK_RECORD PRIMARY KEY (SID, CODE));

By enforcing the PRIMARY KEY constraint, the DBMS can prevent any attempt or mistake of inserting or updating a student record with a duplicate student number. It also ensures that every student on record has a valid ID

number. In the RECORD table, it ensures that each record has a unique combination of SID and CODE values, which means that a student will never be allowed to have two or more records for the same module.

It must be emphasised that a table can have at most one PRIMARY KEY constraint, and it is actually optional (a table does not have to have a PRIMARY KEY constraint). However, it is rare that a table be created without such a constraint, because tables usually do have a primary key.

Review question 1

1. What types of constraints can be declared in SQL?
2. What rules are enforced by the PRIMARY KEY constraint?
3. Is it true that a table must have at least one PRIMARY KEY constraint?

The NOT NULL constraint

The NOT NULL constraint is imposed on any column that must have a value. In the STUDENT table, for example, the attributes DNAME and SLEVEL can have this constraint declared on them to reflect the application requirement that whenever a student is enrolled, he/she must be assigned to a department and be at a certain level.

To declare the constraint on DNAME and SLEVEL, we can use the following SQL statement to create table STUDENT:

CREATE TABLE STUDENT (

SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30) CONSTRAINT NN_STUDENT_DNAME NOT NULL,

SLEVEL NUMBER(1) NOT NULL,

SEMAIL VARCHAR2(40),

CONSTRAINT PK_STUDENT PRIMARY KEY (SID));

You may have noticed that the constraint on DNAME has been given a user-defined name "NN_STUDENT_DNAME", while the one on SLEVEL has not. It is optional to name a NOT NULL constraint. Unlike the PRIMARY KEY constraint, it does not make much difference whether or not you choose to define a name for the constraint. In Oracle, when the NOT NULL constraint is violated, the system will generate an error message. However, this message will not include the name of the NOT NULL constraint, even if one is defined.

Also notice that when a constraint is not to be given a user-defined name, the keyword CONSTRAINT is not used. The same applies to other constraint definitions.

The UNIQUE constraint

The UNIQUE constraint is the same as the PRIMARY KEY constraint, except NULL values are allowed. In the STUDENT table, for example, the SEMAIL attribute should have this constraint. The reason is that according to the university's policy, a student may or may not be given an email account. However, when one is given, the email account name must be unique. By enforcing this constraint on SEMAIL, the DBMS can ensure that different students will not be allowed to have the same email addresses. For those who do not have an email account, the SEMAIL column can have NULL values.

To declare the UNIQUE constraint on SEMAIL, we can use the following SQL statement to create table STUDENT:

CREATE TABLE STUDENT (

SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30) NOT NULL,

SLEVEL NUMBER(1) NOT NULL,

SEMAIL VARCHAR2(40) CONSTRAINT UK_STUDENT_SEMAIL UNIQUE,

CONSTRAINT PK_STUDENT PRIMARY KEY (SID));

Again, an optional user-defined name "UK_STUDENT_SEAMIL" is given to the constraint. This is a good practice in Oracle, because when the UNIQUE constraint is violated, the system will generate an error message containing the name. Similar to the PRIMARY KEY constraint, the constraint's name helps pinpoint the violation. You can avoid giving the constraint a name and just use the UNIQUE keyword:

SEMAIL VARCHAR2(40) UNIQUE

Review question 2

Why is it a good practice to give a name to a declarative constraint?

The CHECK constraint

Declaration of a basic CHECK constraint

The CHECK constraint defines a discrete list of values that a column can have. This list of values may be literally expressed within the constraint declaration or may be defined using a mathematical expression. In the STUDENT table, for example, a student must be at a level between 0 and 3. To impose such a constraint, the CREATE statement for the STUDENT table will be as follows:

CREATE TABLE STUDENT (

SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30) NOT NULL,

SLEVEL NUMBER(1) NOT NULL CONSTRAINT CK_STUDENT_LEVEL CHECK ((SLEVEL>=0) AND (SLEVEL<=3)),

SEMAIL VARCHAR2(40) CONSTRAINT UK_STUDENT_SEMAIL UNIQUE,

CONSTRAINT PK_STUDENT PRIMARY KEY (SID));

Notice two things in the above CREATE statement. First, the CHECK constraint can be declared in a column constraint clause and concatenated (linked) with other NOT NULL, UNIQUE and/or PRIMARY KEY constraints. When a column constraint clause is concatenated, there is no separator between the different constraints, just a comma after the last constraint. Second, the check condition (e.g. (SLEVEL>=0) AND (SLEVEL<=3)) can include logical connectors such as AND and OR. Thus, it is possible to define a complex condition.

Alternatively, the CHECK constraint can be defined using a table constraint clause, such as:

CREATE TABLE STUDENT (

SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30) NOT NULL,

SLEVEL NUMBER(1) NOT NULL,

SEMAIL VARCHAR2(40) CONSTRAINT UK_STUDENT_SEMAIL UNIQUE,

CONSTRAINT PK_STUDENT PRIMARY KEY (SID),

CONSTRAINT CK_STUDENT_LEVEL CHECK ((SLEVEL>=0) AND (SLEVEL<=3)));

It is worth mentioning that when the CHECK constraint is applied to a list of literal values, the values are case sensitive. For example, if only students in the Department of Computing Science or Information Technology are allowed to be

in the database, a CHECK constraint is defined on DNAME in the following way:

……

DNAME VARCHAR2(30) NOT NULL, ……,

CHECK (DNAME IN ('Computing Science', 'Information Technology')), ……;

Any value that does not exactly match the specified values (including 'Computing science') will cause a violation.

Complex CHECK constraints

It is also possible to create a CHECK constraint that is constructed from multiple columns of the table. In this case, because it applies to more than one column, the constraint must be declared with a table constraint clause rather than a column constraint clause. For example, instead of declaring two CHECK constraints on SLEVEL and DNAME respectively, we can use a single constraint called CK_STUDENT_VALIDITY as follows:

CREATE TABLE STUDENT (SID NUMBER(5),

SNAME VARCHAR2(30),

DNAME VARCHAR2(30) NOT NULL,

SLEVEL NUMBER(1) NOT NULL,

SEMAIL VARCHAR2(40) CONSTRAINT UK_STUDENT_SEMAIL UNIQUE,

CONSTRAINT PK_STUDENT PRIMARY KEY (SID),

CONSTRAINT CK_STUDENT_VALIDITY CHECK (((SLEVEL>=0) AND (SLEVEL<=3))

AND (DNAME IN ('Computing Science', 'Information Technology'))));

This CREATE statement will create the same STUDENT table as the earlier statement that uses two separate CHECK constraints.

Review question 3

What is the purpose of using a CHECK constraint?

The FOREIGN KEY constraint

We saw in earlier chapters, when introducing the Relational model, that entities are often linked by a one-to-many relationship. For example, a department may contain many employees, so we say there is a one-to-many relationship between instances of the department entity and instances of the employee entity.

Entities related in this way are sometimes referred to as parents and children; in the example above, the parent entity would be the department table, and the employee entity would be the child table.

A foreign key is a column or a set of columns (attributes) that links each row in the child table containing the foreign key to the row of the parent table containing the matching key value. The FOREIGN KEY constraint enforces referential integrity, which means that, if the foreign key contains a value, that value must refer to an existing, valid row in the parent table.

In our university database, for example, SID and CODE are foreign keys in the RECORD table (notice that SID and CODE together form the primary key for RECORD as well), and RECORD has two parent tables STUDENT and MODULE. For a row in RECORD, there must be an existing student row with the same SID value in STUDENT, and a valid row in MODULE with the same CODE value. Otherwise, the referential integrity is broken. One important implication of this, is that when using FOREIGN KEY constraints, the parent tables must be created before the child tables, and the parent tables must be populated before the child tables, in order to avoid constraint violations. It is important to bear in mind this required order of doing things when undertaking practical work involving FOREIGN KEY constraints.

The following SQL statement can be used to declare the FOREIGN KEY constraints on SID and CODE when creating the RECORD table.

CREATE TABLE RECORD (

SID NUMBER(5),

CODE VARCHAR2(6),

MARK NUMBER(3),

CONSTRAINT PK_RECORD PRIMARY KEY (SID, CODE),

CONSTRAINT FK_RECORD_SID FOREIGN KEY (SID) REFERENCES STUDENT,

FOREIGN KEY (CODE) REFERENCES MODULE);

It can be seen from the above example that:

- The FOREIGN KEY constraint can be given an optional name. In the example, FK_RECORD_SID is the name for the constraint on SID. To define the name, the keyword CONSTRAINT must be used. Otherwise, it is omitted as in the case of declaring the constraint on CODE.
- The keywords FOREIGN KEY define which column (or columns) is the foreign key column to be constrained.
- The keyword REFERENCES indicates the parent table.

By declaring and enforcing the FOREIGN KEY constraint, the DBMS can ensure that the referential integrity is maintained in both the child table(s) and the parent table(s).

In the child table, the DBMS will not allow any INSERT or UPDATE operation that attempts to create a foreign key value without a matching candidate key value in the corresponding parent table (as indicated by the REFERENCES clause).

In the parent table, the DBMS ensures that appropriate actions are taken for any UPDATE or DELETE operation that attempts to change or delete a candidate key value that is being referenced by some rows in the child table. The kind of actions that can be taken are user definable. They are CASCADE, SET NULL, SET DEFAULT and NO ACTION.

CASCADE

This action can be triggered by either a DELETE or an UPDATE operation.

When a parent row is deleted, all its child rows are also deleted. This action can subsequently be applied to each child row deleted, because such rows may themselves have a candidate key that is used as a foreign key in some other tables. Thus, this action may be executed in a cascading manner.

The CASCADE option can be specified in SQL as follows:

CREATE TABLE RECORD (

SID NUMBER(5),

CODE VARCHAR2(6),

MARK NUMBER(3),

CONSTRAINT PK_RECORD PRIMARY KEY (SID, CODE),

FOREIGN KEY (SID) REFERENCES STUDENT ON DELETE CASCADE,

FOREIGN KEY (CODE) REFERENCES MODULE);

In this example, when a student row is deleted from the STUDENT table, all his/her records will also be removed from the RECORD table.

When the candidate key value is changed (by a UPDATE operation), the foreign key column in the child table is set to the same new value. Similar to CASCADE by DELETE, such update actions can be carried out in a cascading manner to the child tables of the child table and so on. For example, when a student's identity number (SID) is changed, all his/her records in the RECORD table should have the new SID value to replace the old. In Oracle, such an action can be defined by creating a trigger (to be discussed later).

SET NULL

When a row is deleted from the parent table, all its child rows will have their corresponding foreign key column set to NULL. This option is only valid if the foreign key column allows NULL value (i.e. it has neither the PRIMARY KEY constraint nor the NOT NULL constraint).

Similarly, when the candidate key value of the parent row is changed, all its child rows may have their corresponding foreign key column set to NULL. Again, this option is valid if and only if the foreign key column allows NULL value.

The SET NULL option can be specified in Oracle by creating corresponding triggers.

SET DEFAULT

By having this option, the operation of deleting the parent row or updating the candidate key value in the parent table will set the corresponding foreign key column in the child table to its default value. This option is only valid if the foreign key column has a DEFAULT value specified.

Again in Oracle, this option can be implemented using appropriate triggers.

NO ACTION

This is the option by default. If there is no other option specified, the DBMS will reject any DELETE or UPDATE in the parent table that may affect rows in the child tables. Any such illegal attempt (to break the referential integrity) will raise an error message in Oracle.

Review question 4

1. Does the keyword CONSTRAINT always need to be used in declaring a constraint?
2. What are the rules enforced by the FOREIGN KEY constraint?

Activity 1 - Creating tables with appropriate constraints

For the university database described in the Context section, we now want to use SQL to create five tables as specified below:

STUDENT

- SID: a five-digit number, which is also the primary key of the table.
- SNAME: a string of characters; maximum length is 30.
- SLEVEL: a single-digit integer; must have a value.
- SEMAIL: a string of characters; maximum length is 40; must be unique.
- DNAME: foreign key referring to the DEPT table; must have a value.

MODULE

- CODE: a string of 6 letters and/or numbers; primary key of the table.
- TITLE: a string of characters; maximum length is 45; must be unique.
- EID: foreign key referring to the LECTURER table; must have a value.
- DNAME: foreign key referring to the DEPT table; must have a value.

LECTURER

- EID: a six-digit number; primary key of the table.
- LNAME: a string of characters; maximum length is 30.
- LEMAIL: a string of characters; maximum length is 40; must be unique.
- DNAME: foreign key referring to the DEPT table; must have a value.

DEPT

- DNAME: a string of characters; maximum length is 30; primary key of the table.
- LOCATION: a string of characters; maximum length is 35; must have a value.

RECORD

- SID: foreign key referring to the STUDENT table; primary key attribute.
- CODE: foreign key referring to the MODULE table; primary key attribute.
- MARK: an integer.

Activity 2 - Inserting data into the tables and enforcing constraints

Having created the five tables, we can now insert data records into them. The records that are to be stored are listed below. Insert each of them and see what happens after the insertion of the highlighted rows, bearing in mind the constraints that some of the columns may have.

LECTURER

EID	LNAME	LEMAIL	DNAME
733911	Short		Computing Science
723816	Long		Computing Science
931285	Woods		Business

STUDENT

SID	SNAME	SLEVEL	SEMAIL	DNAME
99038	Tomkins	1		Computing Science
99710	Major	1	major@mdx.ac.uk	Business Studies
97329	Cook	3	Cook@mdx.ac.uk	Computing Science
98544	Blair	2		Computing Science
97626	Major	3	major@mdx.ac.uk	English
98914	Redwood	2		Business Studies
96777	Bond		bond@mdx.ac.uk	Computing Science

DNAME	LOCATION
Computing Science	Hendon
English	Cat Hill
Business Studies	Bounds Green

MODULE

CODE	TITLE	EID	DNAME
CS1234	Databases	723816	Computing Science
CS4222	System Design	733911	Computing Science
EG8361	Literature	237188	English
BS5544	Business Environment	931285	Business Studies

RECORD

SID	CODE	MARK
99038	CS1234	71
97329	CS1234	69
98544	CS4222	85
99710	BS5544	43
98914	CS1234	31
99038	CS4222	66
98544	CS1234	81
99038	BS5544	63

Activity 3 - Maintaining referential integrity

We have learned that by declaring and enforcing the FOREIGN KEY constraint, the DBMS can ensure that the referential integrity is maintained in both the child table(s) and the parent table(s).

In the child table, the DBMS will not allow any INSERT or UPDATE operation that attempts to create a foreign key value without a matching candidate key value in the corresponding parent table (as indicated by the REFERENCES clause). We have seen one of such examples in Activity 2.

In the parent table, the DBMS ensures that appropriate actions are taken for any UPDATE or DELETE operation that attempts to change or delete a candidate key value that is being referenced by some rows in the child table.

Now try to perform the following two operations on our university database and see what happens:

- Operation 1: Change the name of the Department of Computing Science to simply 'Computing' in the DEPT table.
- Operation 2: Delete all level 3 students from the STUDENT table.

Changing the definition of a table

Once created, a table's definition can still be changed using the ALTER TABLE command in SQL. Different DBMSs implement ALTER TABLE differently, providing more or less functionality than that specified in the SQL standard. In Oracle, the following operations can be carried out on a table using appropriate ALTER TABLE statements:

- Add a new column, including a constraint declaration for that column.
- Modify an existing column's type, with certain restrictions.
- Modify an existing column's constraint definition, with certain restrictions.
- Add a new constraint to an existing column.
- Drop (remove) an existing constraint from a column.

Add a new column

Suppose we now want to create a new column in the RECORD table to store the date on which the mark was obtained. The column is to be named EXAM_DATE, and can be added in the following way:

ALTER TABLE RECORD

ADD EXAM_DATE DATE;

If there is an application requirement stating that the exam date must not be earlier than 1st January 1998 in order for the mark to be valid, we can include a CHECK constraint as well when creating the new column. In this case, the following SQL statement is used instead of the previous one:

ALTER TABLE RECORD

ADD EXAM_DATE DATE CONSTRAINT CK_RECORD_DATE CHECK (TO_CHAR(EXAM_DATE, 'YYMMDD') >= '980101');

The constraint is given a name "CK_RECORD_DATE". The system function TO_CHAR is used to convert EXAM_DATE into a string so that it can be compared with '980101', representing 1st January 1998. Other constrains can be specified in the column constraint clause in a similar way.

Modify an existing column's type

Using the ALTER TABLE command, we can modify the type definition of a column with the following restrictions:

- If there is data (except NULL) present in the column, then the type of this column cannot be changed. The type definition can only be changed if the table is empty, or all values in the column concerned are NULL.
- If the type definition is changed on a column with a UNIQUE or PRIMARY KEY constraint, it may potentially become incompatible with the data type of a referencing FOREIGN KEY. Thus, the ALTER TABLE command should be used with caution.

The following SQL statement changes the data type of SID in the RECORD table to NUMBER(9) (the original type was NUMBER(5)):

ALTER TABLE RECORD

MODIFY SID NUMBER(9);

Notice that SID in RECORD is a foreign key referencing SID in STUDENT which still has the type NUMBER(5). However, because NUMBER(9) and NUMBER(5) are compatible, this ALTER TABLE operation is allowed. If we attempt to change SID to be of type VARCHAR2(5), it will be rejected because of incompatible data types.

Modify an existing column's constraint definition

There are a number of possibilities for modifying a constraint definition. If a column has the NOT NULL constraint, it can be changed to NULL to allow null values, and vice versa.

For example, the SEMAIL column in the STUDENT table did allow NULL values. If we wish to change this, the following SQL statement is used:

ALTER TABLE STUDENT MODIFY SEMAIL NOT NULL;

Notice that the above operation is valid if and only if the table is empty or the SEMAIL column does not have any NULL value. Otherwise, the operation is rejected.

The SEMAIL column can also be changed back to allow NULL values as following:

ALTER TABLE STUDENT MODIFY SEMAIL NULL;

For other existing constraints (i.e. UNIQUE, CHECK and FOREIGN KEY), if they need be changed, they have to be removed first (DROP) and then new ones added (ADD).

Add a new constraint

New constraints, such as UNIQUE, CHECK and FOREIGN KEY, can be added to a column.

In our university database, for example, we may add a UNIQUE constraint on SNAME in the STUDENT table. As a result, no students can have the same name in the database.

ALTER TABLE STUDENT

ADD CONSTRAINT UK_STUDENT_SNAME UNIQUE(SNAME);

In the RECORD table, if we want to ensure that MARK is always less than or equal to 100, we can add a CHECK constraint on MARK as follows:

ALTER TABLE RECORD ADD CONSTRAINT CK_RECORD_MARK CHECK (MARK <= 100);

In the LECTURER table, DNAME is a foreign key with a link to the DEPT table. If we did not declare a FOREIGN KEY constraint on DNAME when creating LECTURER, we can add it now using the following statement:

ALTER TABLE LECTURER

ADD CONSTRAINT FK_LECTURER_DNAME

FOREIGN KEY (DNAME) REFERENCES DEPT;

All the keywords in the statements are highlighted. Notice that we have given names to all the newly added constraints. They will be helpful when constraints have to be dropped.

Drop an existing constraint

The ALTER TABLE … DROP command is used to remove an existing constraint from a column. This operation is effectively to delete its definition from the data dictionary of the database.

Earlier, we added a UNIQUE constraint (named UK_STUDENT_SNAME) on SNAME in the STUDENT table. It prevents any students having the same name. Obviously it is not practically useful, because it is always possible that some students may happen to have the same name. In order to remove this constraint, we can use the following SQL statement:

ALTER TABLE STUDENT DROP CONSTRAINT UK_STUDENT_SNAME;

Notice that in order to drop a constraint, its name has to be specified in the DROP clause. There is no difficulty if the constraint has a user-defined name. However, if the user does not give a name to the constraint when it is declared, the DBMS will automatically assign a name to it. To remove such a constraint, the system-assigned name has to be found out first. It can be done in Oracle, but it causes some unnecessary trouble. This may be another incentive to define a name for a constraint when it is declared.

Review question 5

1. How does one change the definition of a constraint on a column?
2. How does one remove an existing constraint?

Activity 4 - Changing an existing column's constraint

In Activity 2, when we were trying to insert the following record:

96777	Bond		bond@mdx.ac.uk	Computing Science

into the STUDENT table, an error occurred. This was because there was a NOT NULL constraint declared on SLEVEL. As a result, the SLEVEL column cannot take NULL values. The insertion did not take place.

In this activity, we will change the constraint on SLEVEL from NOT NULL to NULL so that NULL value is allowed. Write an SQL statement to perform the change and then re-insert the record into the STUDENT table. It should now be held in the table.

Activity 5 - Adding a new constraint to a column

When the RECORD table was created, there was no constraint on MARK. As a normal business rule, however, we know that a student's mark should always be between 0 and 100. Thus, we can declare an appropriate constraint to enforce this rule. Since the RECORD table has already been created, we need to use the ALTER TABLE command to add the new constraint.

Write a proper SQL statement to perform the required operation. Having added the new constraint, try to increase all the marks in module CS1234 by 80 and see what happens.

Activity 6 - Modifying an existing FOREIGN KEY constraint

In Activity 3, we could not delete level 3 students from the STUDENT table, because they were still being referenced by some child rows via the foreign key SID in the RECORD table.

Now suppose that we want to relax the constraint a bit so that we may remove student rows from the STUDENT table together with their child rows. In this case, the FOREIGN KEY constraint on SID in RECORD needs be changed to include the option to allow cascade deletions.

Write proper SQL statements to perform the required modification. Remember that for existing constraints such as FOREIGN KEY, UNIQUE and CHECK, if they need be modified, they have to be removed first and then new ones added.

Now, having modified the constraint, perform the operation to delete all level 3 students from the STUDENT table. Then check both the STUDENT and RECORD tables to see what rows have been deleted.

Database triggers

A trigger defines an action the database should take when some database-related event occurs. Triggers may be used to:

- Supplement declarative constraints, to maintain database integrity.
- Enforce complex business rules.
- Audit changes to data.

Different DBMSs may implement the trigger mechanism differently. In this chapter, we use Oracle to discuss triggers. In Oracle, a trigger consists of a set of PL/SQL statements. The execution of triggers is transparent to the user. They are executed by the DBMS when specific types of data manipulation commands are performed on specific tables. Such commands include INSERT, UPDATE and DELETE.

Because of their flexibility, triggers may supplement database integrity constraints. However, they should not be used to replace them. When enforcing business rules in an application, you should first rely on the declarative constraints available in the DBMS (e.g. Oracle); only use triggers to enforce rules that cannot be coded through declarative constraints. This is because the enforcement of the declarative constraints is more efficient than the execution of user-created triggers.

It is worth mentioning that in order to create a trigger on a table, you must be able to alter that table and any other table that may subsequently be affected by the trigger's action. You need to ensure that you have sufficient privilege to do so.

Types of triggers

In Oracle, there are fourteen types of triggers that can be implemented using PL/SQL. Once again, note that other DBMSs may not have the same support, and that you should consult your system's documentation if you encounter any problems. The type of a trigger is defined by the following three features:

- Event
- Level
- Timing

Event

Refers to the triggering SQL statement; that is, INSERT, UPDATE or DELETE. A single trigger can be designed to fire on any combination of these SQL statements.

Level

Refers to statement-level versus row-level triggers. The level of a trigger denotes whether the trigger fires once per SQL statement or once for each row affected by the SQL statement.

Statement-level triggers execute once for each SQL statement. For example, if an UPDATE statement updates 300 rows in a table, the statement-level trigger of that table would only be executed once. Thus, these triggers are not often used for data-related activities. Instead, they are normally used to enforce additional security measures on the types of transactions that may be performed on a table.

Statement-level triggers are the default type of triggers created via the CREATE TRIGGER command.

Row-level triggers execute once for each row operated upon by a SQL statement. For example, if an UPDATE statement updates 300 rows in a table, the row-level trigger of that table would be executed 300 times. Also, row-level triggers have access to column values of the row currently being operated upon by the SQL statement. They can evaluate the contents of each column for that row. Thus, they are the most common type of triggers and are often used in data-auditing applications.

Row-level triggers are created using the FOR EACH ROW clause in the CREATE TRIGGER command.

It is important to know that a trigger can only be associated with one table, but a table can have a mixture of different types of triggers.

Timing

Timing denotes whether the trigger fires BEFORE or AFTER the statement-level or row-level execution. In other words, triggers can be set to occur immediately before or after those triggering events (i.e. INSERT, UPDATE and DELETE).

Within the trigger, one will be able to reference the old and new values involved in the transaction. 'Old' refers to the data as it existed prior to the transaction. UPDATE and DELETE operations usually reference such old values. 'New' values are the data values that the transaction creates (such as being INSERTed).

If one needs to set a column value in an inserted row via a trigger, then a BEFORE INSERT trigger is required in order to access the 'new' values. Using an AFTER INSERT trigger would not allow one to set the inserted value, since the row will already have been inserted into the table. For example, the BEFORE INSERT trigger can be used to check if the column values to be inserted are valid or not. If there is an invalid value (according to some pre-specified business

rules), the trigger can take action to modify it. Then only validated values will be inserted into the table.

AFTER row-level triggers are often used in auditing applications, since they do not fire until the row has been modified. Because the row has been successfully modified, it implies that it has satisfied the referential integrity constraints defined for that table.

In Oracle, there is a special BEFORE type of trigger called an INSTEAD OF trigger. Using an INSTEAD OF trigger, one can instruct Oracle what to do instead of executing the SQL statement that has activated the trigger. The code in the INSTEAD OF trigger is executed in place of the INSERT, UPDATE or DELETE triggering transaction.

Valid trigger types

To summarise, the fourteen types of triggers are listed below.

Triggered by INSERT:

- BEFORE INSERT statement-level.
- BEFORE INSERT row-level.
- AFTER INSERT statement-level.
- AFTER INSERT row-level.

Triggered by UPDATE:

- BEFORE UPDATE statement-level.
- BEFORE UPDATE row-level.
- AFTER UPDATE statement-level.
- AFTER UPDATE row-level.

Triggered by DELETE:

- BEFORE DELETE statement-level.
- BEFORE DELETE row-level.
- AFTER DELETE statement-level.
- AFTER DELETE row-level.

To replace the triggering event:

- INSTEAD OF statement-level.
- INSTEAD OF row-level.

The first twelve types of triggers are most commonly used, and are discussed in this chapter. The INSTEAD OF triggers are more complex than others, and interested students are advised to refer to books specifically dealing with Oracle PL/SQL programming.

Review question 6

What is a database trigger? Should triggers be used to replace declarative constraints and why?

Creating triggers

Instead of presenting a formal syntax for creating triggers, a number of examples are used to illustrate how different types of triggers are created.

Statement-level trigger

This type of trigger is created in the following way:

CREATE TRIGGER first_trigger_on_student

BEFORE INSERT ON STUDENT BEGIN

[the trigger body consisting of PL/SQL code;]

END;

The CREATE TRIGGER clause must define the trigger's name. In the example, it is called "first_trigger_on_student". In practice, the name must be something that can reflect what the trigger does. In this sense, "first_trigger_on_student" is not a good choice of name, but merely for convenience of illustrating the syntax.

In the next clause (after CREATE TRIGGER), the timing and triggering event must be specified. In our example, the trigger will fire BEFORE (timing) any INSERT (event) operation ON the STUDENT table. Obviously, timing can also be AFTER, and event be UPDATE or DELETE.

The last part of a trigger definition is the BEGIN/END block containing PL/SQL code. It specifies what action will be taken after the trigger is invoked.

In the above example, instead of defining a single triggering event (INSERT), a combination of the three events may be specified as follows:

CREATE OR REPLACE TRIGGER first_trigger_on_student

BEFORE INSERT OR UPDATE OR DELETE ON STUDENT BEGIN

[the trigger body consisting of PL/SQL code;]

END;

In this case, any of the INSERT UPDATE, and DELETE operations will activate the trigger. Also notice that instead of using a CREATE TRIGGER clause, we use CREATE OR REPLACE TRIGGER. Because the "first_trigger_on_student" trigger is already in existence, the keywords CREATE OR REPLACE are used. For defining new triggers, the keyword CREATE alone is sufficient

Option for the UPDATE event

If the timing and triggering event are simply defined as

BEFORE UPDATE ON STUDENT

then UPDATE on any column will fire the trigger. In Oracle, we have the option to specify a particular column whose update will activate the trigger. Updates on other columns will have no effects on the trigger.

For example, we can define a trigger specifically for UPDATE on DNAME in the STUDENT table (meaning whenever a student changes department, the trigger fires). This trigger is called "second_trigger_on_student".

CREATE TRIGGER second_trigger_on_student

BEFORE UPDATE OF DNAME ON STUDENT

BEGIN [the trigger body consisting of PL/SQL code;]

END;

Notice the option "OF column's name" is used for the UPDATE operation.

Row-level triggers

To define a row-level trigger, the FOR EACH ROW clause must be included in the CREATE TRIGGER statement. For example, if we want to have a trigger for each INSERT, UPDATE and DELETE operation on every row that is affected, we can create the trigger in the following way:

CREATE TRIGGER third_trigger_on_student

AFTER INSERT OR UPDATE OR DELETE ON STUDENT

FOR EACH ROW

BEGIN

[the trigger body consisting of PL/SQL code;]

END;

The trigger will fire whenever a row has been inserted, updated or deleted. At the row-level, we can also add the option for the UPDATE event.

CREATE OR REPLACE TRIGGER third_trigger_on_student

AFTER INSERT OR UPDATE OF DNAME OR DELETE ON STUDENT

FOR EACH ROW

BEGIN

[the trigger body consisting of PL/SQL code;]

END;

Option for the row-level triggers

For row-level triggers, there is another optional clause which can be used to further specify the exact condition for which the trigger should fire. This is the "WHEN 'condition' " clause. The 'condition' must be evaluated to be TRUE for the trigger to fire. If it is evaluated to be FALSE or does not evaluate because of NULL values, the trigger will not fire.

For example, if we want to take some action when a student is moved to the Department of Computing Science, we can define a trigger like the following:

CREATE TRIGGER fourth_trigger_on_student

AFTER UPDATE OF DNAME ON STUDENT

FOR EACH ROW WHEN (NEW.DNAME = 'Computing Science')

BEGIN

[the trigger body consisting of PL/SQL code;]

END;

Notice how the WHEN clause is used to specify the exact condition for the trigger to fire. The 'condition' can be a complex Boolean expression connected by AND/OR logical operators.

Also, the notation "NEW.column_name" (such as NEW.DNAME) refers to the column (e.g. DNAME) which has a new value as a result of an INSERT or UPDATE operation. Similarly, the notation "OLD.column_name" refers to the column which still has the old value prior to an UPDATE or DELETE operation. These two notations are very useful for maintaining data integrity. (Note that in the BEGIN … END block, a colon ':' needs to be placed before OLD and NEW.)

Another example: Suppose we want to take some action when a student is to leave the Department of English. We can define an appropriate trigger in the following way:

CREATE TRIGGER fifth_trigger_on_student

BEFORE UPDATE OF DNAME OR DELETE ON STUDENT

FOR EACH ROW WHEN (OLD.DNAME = 'English')

BEGIN

[the trigger body consisting of PL/SQL code;]

END;

Compare "fifth_trigger_on_student" with "fourth_trigger_on_student", and see how they are different.

Removing triggers

Existing triggers can be deleted via the DROP TRIGGER command. For example, the "first_trigger_on_student" trigger is removed from the STUDENT table in the following way:

DROP TRIGGER first_trigger_on_student;

Using triggers to maintain referential integrity

As studied in the earlier part of this chapter, the FOREIGN KEY constraint is often used for ensuring the referential integrity among parent and child tables. However, the FOREIGN KEY constraint can only enforce standard integrity rules. They are:

- The foreign key column in the child table cannot reference non-existing rows in the parent table.
- If the DELETE CASCADE option is not chosen, a row in the parent table that is being referenced via a foreign key column cannot be deleted.
- If the DELETE CASCADE option is chosen, the row can be deleted together with all the rows in the child table which reference the parent row.

If other non-standard rules have to be enforced as well, then appropriate triggers need to be created. Some possible non-standard rules are:

- Cascade updates.
- Set the foreign key column to NULL on updates and deletes.
- Set a default value to the foreign key column on updates and deletes. The meanings of these rules have been explained before.

It must be emphasised that if triggers are used instead of the standard FOREIGN KEY constraint, then for each of the integrity rules (standard and non-standard), one or more triggers may need to be implemented. Also, the FOREIGN KEY constraint must not be declared when creating the corresponding tables. Otherwise, the triggers will not work, because the standard FOREIGN KEY constraint will override the trigger actions.

In this section, we are going to see two examples of using triggers to implement the DELETE CASCADE rule and the UPDATE CASCADE rule. The two tables concerned are STUDENT and RECORD:

STUDENT(SID, SNAME, DNAME, SLEVEL, SEMAIL)

RECORD(SID, CODE, MARK)

We know that SID in RECORD is a foreign key linking to STUDENT.

To create the trigger to cascade deletes:

CREATE TRIGGER cascade_deletes_student_record

BEFORE DELETE ON STUDENT

FOR EACH ROW

BEGIN

DELETE FROM RECORD

WHERE RECORD.SID = :OLD.SID;

END;

It can be seen from the above example that, before the parent row is deleted from the STUDENT table, all the child rows in the RECORD table are deleted. This maintains the referential integrity. (In the PL/ SQL code, :OLD.SID represents the SID of the row in the STUDENT table, which is to be deleted.)

To create the trigger to cascade updates:

CREATE TRIGGER cascade_updates_student_record

AFTER UPDATE OF SID ON STUDENT

FOR EACH ROW

BEGIN

UPDATE RECORD

SET RECORD.SID = :NEW.SID

WHERE RECORD.SID = :OLD.SID;

END;

Again, it can be seen from the example that, after the parent row is updated in the STUDENT table, all the child rows in the RECORD table are updated accordingly. This maintains the referential integrity.

Using triggers to maintain business rules

In the Context section, it was mentioned that in the university database, a lecturer can teach no more than four modules (i.e. this is a business rule). This restriction can be enforced by defining a trigger on the MODULE table to ensure that no lecturer has more than four corresponding rows in the table. The MODULE table's structure is as following:

MODULE(CODE, TITLE, EID, DNAME)

To create the trigger,

CREATE TRIGGER max_teaching_load

BEFORE INSERT OR UPDATE OF EID ON MODULE

FOR EACH ROW

DECLARE NO_OF_MODULES INTEGER(1);

BEGIN

SELECT COUNT(*) INTO NO_OF_MODULES

FROM MODULE

WHERE MODULE.EID = :NEW.EID;

IF NO_OF_MODULES >= 4 THEN

RAISE_APPLICATION_ERROR(-20001, 'Maximum teaching load exceeded for this lecturer!');

END IF;

END;

In the above code, the DECLARE NO_OF_MODULES INTEGER(1) clause defines an integer variable called NO_OF_MODULES.

In this example, the key point is that we use the RAISE_APPLICATION_ERROR procedure (system provided) to generate an error message and stop the execution of any INSERT or UPDATE OF EID operation which may result in a lecturer teaching more than four modules. In RAISE_APPLICATION_ERROR, the number -20001 is a user-defined error number for the condition (the number must be between –20001 and –20999), and the text in single quotes is the error message to be displayed on the screen.

Now suppose that the university has another rule stating that a student's mark cannot be changed by more than 10% of the original mark. In this case, we can define a trigger on the RECORD table in the following way:

CREATE OR REPLACE TRIGGER mark_change_monitoring

BEFORE UPDATE OF MARK ON RECORD

FOR EACH ROW

BEGIN

IF ((:NEW.MARK/:OLD.MARK) >= 1.1) OR ((:OLD.MARK/:NEW.MARK) >= 1.1)

THEN

RAISE_APPLICATION_ERROR(-20002, 'Warning: Large percentage change in marks prohibited.');

END IF;

END;

The above two examples should have shown you what triggers are capable of doing. In fact, using Oracle's PL/SQL language, one can write much more complex triggers to enforce various business rules. The discussion of PL/SQL is beyond the scope of this module. Interested students are again advised to refer to any book specifically dealing with Oracle PL/SQL programming for more information on writing triggers.

Review question 7

1. When and how do we use triggers to maintain referential integrity?
2. How do we use triggers to implement business rules in the database?

Activity 7 - Creating triggers to prevent updates and deletions

In the university database, we can see that the rows in the DEPT table are often referenced by many child rows in a number of other tables (e.g. STUDENT, LECTURER and MODULE). Although there are FOREIGN KEY constraints declared on the child tables to maintain the referential integrity, we can still define a trigger in the parent table (i.e. DEPT) to stop any attempt to change the name of the department and/or to remove any of the DEPT rows. This is corresponding to the business rule stating that once a university department is established, it will be there 'forever' and will not be allowed to change name (we assume that such a rule is necessary).

In this activity, write appropriate PL/SQL statements to create the trigger. After the trigger is created, try to change the name of some departments in DEPT and delete a row from DEPT, and see what happens. Find out how the trigger works.

Note that in order to execute PL/SQL statements in some DBMSs (including Oracle's SQL*PLUS), you may need to end the block of statements with a '/'. Consult your DBMS's documentation for any additional semantics requirements.

Activity 8 - Creating triggers to maintain data validity

In Activity 5, we have declared a CHECK constraint on MARK in the RECORD table. Any mark that is not between 0 and 100 will cause a violation of the

constraint. Applying the CHECK constraint, however, we would not know whether the mark is greater than 100 or smaller than 0 (i.e. a negative number).

In this activity, we will create a trigger to replace the original CHECK constraint, which can tell us how the restriction on MARK is violated. Whenever the value of MARK is beyond the valid range (0 – 100), the trigger will generate an error message informing users whether it is greater than 100 or a negative number.

Write proper PL/SQL statements to create the trigger, and use some SQL UPDATE and INSERT statements to test it. Remember we need to drop the CHECK constraint first, otherwise it will override any other triggers on MARK.

Activity 9 - Creating triggers to validate new column values

In the STUDENT table, the column SLEVEL can only take value 0, 1, 2, 3 or NULL. Any other value is illegal. In order to ensure that only valid values are stored in SLEVEL, we can create a trigger to automatically validate any new value to be updated or inserted. The rules are:

- If a new value is smaller than 0, then set it to 0 before updating or inserting it.
- An appropriate message is always displayed on the screen to inform the user that a proper validation has been carried out.

Write proper PL/SQL statements to create the trigger, and use some SQL UPDATE and INSERT statements to test it.

Note that the Oracle's DBMS_OUTPUT.PUT_LINE procedure can be used to display text on screen. The basic syntax is:

DBMS_OUTPUT.PUT_LINE ('the text to be displayed in single quotes');

Remember that for Oracle, in order to use DBMS_OUTPUT.PUT_LINE to display text on screen, you need to execute the command "SET SERVEROUTPUT ON" once in the SQL*PLUS environment.

Additional features of Oracle

The rest of this chapter deals with some additional features that are present in Oracle. In other advanced DBMSs, similar features are also available. Again, please consult your DBMS's documentation to find out whether or not it supports the following features.

Stored procedures

Some sophisticated business rules and application logic can be implemented and stored as procedures within Oracle. In fact, triggers are special types of

procedures associated with tables and invoked (called upon) by pre-specified events.

Stored procedures, containing SQL or PL/SQL statements, allow one to move code that enforces business rules from the application to the database. As a result, the code can be stored once for use by different applications. Also, the use of stored procedures can make one's application code more consistent and easier to maintain. This principle is similar to the good practice in general programming in which common functionality should be coded separately as procedures or functions.

Some of the most important advantages of using stored procedures are summarised as follows:

- Because the processing of complex business rules can be performed within the database, significant performance improvement can be obtained in a networked client-server environment (refer to client-server chapters for more information).
- Since the procedural code is stored within the database and is fairly static, applications may benefit from the reuse of the same queries within the database. For example, the second time a procedure is executed, the DBMS may be able to take advantage of the parsing that was previously performed, improving the performance of the procedure's execution.
- Consolidating business rules within the database means they no longer need to be written into each application, saving time during application creation and simplifying the maintenance process. In other words, there is no need to reinvent the wheel in individual applications, when the rules are available in the form of procedures.

Function and packages

In Oracle, a procedure is implemented to perform certain operations when called upon by other application programs. Depending on the operations, it may not return any value, or it might return one or more values via corresponding variables when its execution finishes. Unlike procedure, a function always returns a value to the caller as a result of completing its operations. It is also worth mentioning that a function may be invoked by code within a procedure, and a procedure may be called from within a function.

In Oracle, groups of procedures, functions, variables and SQL statements can be organised together into a single unit, called a package. To execute a procedure within a package, one must first specify the package name, followed by the procedure name, as shown in the following example:

DBMS_OUTPUT.PUT_LINE('This is an example to show how to use a procedure within a package.');

In the example, DBMS_OUTPUT is the package containing a number of procedures relating to displaying message on the screen. PUT_LINE is a procedure within the DBMS_OUTPUT package that can take a string of characters (i.e. the text in the single quotes) and output them onto the screen.

Note that in order to use Oracle's DBMS_OUTPUT.PUT_LINE procedure to display text on screen, you need to execute the command SET SERVEROUTPUT ON once in the SQL*PLUS environment.

Creating procedures

We use the following example to illustrate how to create a procedure:

CREATE PROCEDURE check_student_mark (id_number IN INTEGER, module_code IN VARCHAR2, the_mark OUT INTEGER)

AS

BEGIN

SELECT MARK INTO the_mark

FROM RECORD

WHERE SID = id_number AND CODE = module_code;

END;

It can be seen from the example, that the syntax for creating procedures is somewhat similar to that of creating triggers, except that we need to specify the input and/or output variables to be used in the procedure.

The CREATE PROCEDURE clause defines the procedure's name (e.g. "check_student_mark") as well as variables. The keyword IN defines an input variable together with its data type (e.g. id_number IN INTEGER), and OUT defines an output variable together with its data type(e.g. the_mark OUT INTEGER). The query result is returned via the output variable "the_mark".

After the keyword AS, the BEGIN/END block contains PL/SQL code to implement the required operations. It can be as simple as containing a single SQL statement, as in our example. It can also be as complex as necessary.

The created procedure can then be used to retrieve the mark for a particular student on a specific module. For example,

check_student_mark(12345, 'BIS42', the_mark)

will retrieve the mark for the student with identity number of 12345 on module BIS42. The query result is stored in the variable "the_mark".

The CREATE PROCEDURE clause is used for creating a new procedure. To replace an existing one, we can use CREATE OR REPLACE PROCEDURE.

Creating functions

Again we use an example to illustrate how to create a function:

CREATE FUNCTION retrieve_my_mark (id_num IN INTEGER, mod_code IN VARCHAR2)

RETURN INTEGER

IS

my_mark INTEGER;

BEGIN

SELECT MARK INTO my_mark

FROM RECORD

WHERE SID = id_num AND CODE = mod_code; RETURN (my_mark);

END;

It can be seen from the example, that the syntax for creating functions is similar to that of creating procedures. However, there are a couple of important differences.

The CREATE FUNCTION clause defines the function's name (e.g. "retrieve_my_mark") as well as input variables. The keyword IN defines an input variable together with its data type (e.g. id_num IN INTEGER).

The RETURN keyword specifies the data type of the function's return value, which can be any valid PL/SQL data type (e.g. RETURN INTEGER). Every function must have a RETURN clause, because the function must, by definition, return a value to the calling environment.

In the BEGIN/END block, the RETURN(my_mark) command performs the necessary action to return the required value to the calling environment.

The created function can then be used to retrieve the mark for a particular student on a specific module. For example,

my_mark := retrieve_my_mark(12345, 'CSC4001')

will retrieve the mark for the student with identity number of 12345 on module CSC4001. The query result is returned and held in the variable "my_mark".

The CREATE FUNCTION clause is used for creating a new function. To replace an existing one, we can use CREATE OR REPLACE FUNCTION.

Calling a procedure from within a function and vice versa

If the check_student_mark procedure has already been created, then it can be used by function retrieve_my_mark. In this case, the function is redefined as following:

CREATE OR REPLACE FUNCTION retrieve_my_mark (id_num IN INTEGER, mod_code IN VARCHAR2)

RETURN INTEGER

IS

my_mark INTEGER;

BEGIN

check_student_mark(id_num, mod_code, my_mark); RETURN (my_mark);

END;

It can be seen from the above example, that instead of rewriting an SQL retrieval statement, the procedure check_student_mark is used to retrieve the mark and store the result in the variable my_mark. And then the value held in my_mark is returned as the value of the function.

Similarly, if the function is created first, then it may be invoked from within the procedure.

Review question 8

Why are stored procedures useful in databases?

Discussion topics

Having studied this chapter, we should have obtained a fair amount of knowledge about declarative constraints, database triggers and stored procedures. We have seen in Oracle that PL/SQL is a very useful as well as powerful language for creating triggers to enforce business rules.

Now use any database application with which you may be familiar (e.g. for a bank, a car-rental company, etc) to discuss in general what kind of application logic and/or business rules should be implemented in the database using constraints and triggers. The objective of this discussion is to help you understand further the usefulness and benefits of the techniques.

Additional content and activities

In this chapter, we have studied the five declarative constraints and the mechanisms for creating database triggers and procedures in more advanced DBMSs

such as Oracle. We have seen that the constraints, database triggers and procedures can be effectively used to incorporate application logic and enforce business rules in databases.

A number of examples have been provided in this chapter. As additional activities, you may wish to try out those examples, using the university database that has been created during previous activities.

We have also seen in this chapter that the PL/SQL language of Oracle plays a very important role in creating database triggers and procedures. In fact, PL/SQL is a powerful language in Oracle that enables us to construct flexible triggers and procedures to deal with various complex application issues. Although we were not able to cover PL/SQL to a sufficient extent, interested students who want to develop further knowledge on PL/SQL are advised to read relevant books.

Chapter 11. File Organisation and Indexes

Table of contents

    - Performance issues
    - Search, insertion and deletion with B+ trees
    - Dealing with overflow
    - Dealing with underflow
- B* tree: A variation of B-tree and B+ tree
- Summary

Objectives

At the end of this chapter you should be able to:

- Describe how files and records can be placed on disks, and the effective ways in which records can be organised in files.
- Describe a number of different types of indexes commonly found in modern database environments.
- Understand the data structures which can support the various indexes.
- Be fully aware of the proper ways in which indexes are used.
- Use standard SQL syntax to create and remove different types of index on a set of tables.
- Be aware of the typical approaches used in industry and commerce to improve database performance through indexing.

Introduction

In parallel with this chapter, you should read Chapter 16 and Chapter 17 of Ramez Elmasri and Shamkant B. Navathe, " FUNDAMENTALS OF Database Systems", (7th edn.).

In this chapter, we will study how files and records can be placed on disks, and what the effective ways are in which records can be organised in files. The three file organisations we will learn in this chapter are heap file, sorted file and hash file. The only one goal in applying these various file organisation techniques is to provide the database system with good performance.

It is not only important that a database is designed well, but part of the design process also involves ensuring that the structure of the developed system is efficient enough to satisfy users' requirements now and into the future.

Database tuning involves techniques that can be used to improve performance. It is an important and complex subject to which a number of chapters are devoted in this module. We will be looking into design issues of indexes, and the appropriate ways of using indexes. Indexes play a similar role in database

systems as they do in books, in that they are used to speed up access to information. File structures can be affected by different indexing techniques, and they in turn will affect the performance of the databases.

It is worth emphasising again the importance of file organisations and their related access methods. Tuning techniques can help improve performance, but only to the extent that is allowed by a particular file organisation. For example, if you have a Ford car, you may obtain a better performance if you have the car's engine tuned. However, no matter how hard you tune it, you will never be able to get a Ferrari's performance out of it.

Indexes can help database developers build efficient file structures and offer effective access methods. When properly used and tuned, the database performance can be improved further. In fact, indexes are probably the single most important mechanism explicitly available to database developers and administrators for tuning the performance of a database.

The contents of this chapter are related to discussions on Database Administration and Further Performance Tuning Techniques.

Context

In this chapter, we will describe the techniques used to store large amounts of structured data on disks. These techniques are important for database designers, DBAs (database administrators) and implementers of a DBMS. Database designers and DBAs must know the advantages and disadvantages of each storage method in order to develop and operate a DBMS for a specific application. Even an off-the-shelf DBMS will usually have different options available for organising the data, and the process of physical database design involves selecting the most appropriate data organisation technique for the given set of application requirements.

A typical database application will always need to access the database and retrieve some data for processing. Whenever a certain portion of the data is needed, it must be located on disk, loaded to main memory for processing, and then written back to the disk if changes have been made.

The data stored on disk is organised as files of records. Each record is a collection of data values that can be interpreted as facts about entities, their attributes and relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently whenever they are needed.

There are a number of commonly used file organisations which can determine how the records of a file are physically placed on disk. In this chapter, we will be discussing:

- **Heap file:** which places records on disk in no particular order.

- **Sorted sequential file:** which holds records in a particular order based on the value of a specified field (i.e. attribute).
- **Hashed file:** which uses a hash function to decide where a record should be placed on disk.

In this chapter, we will also introduce access structures called indexes, which are used to speed up the retrieval of records if certain requirements on search conditions are met. An index for a file of records works just like an index catalogue in a library. In a normal library environment, for example, there should be catalogues such as author indexes and book title indexes. A user may use one of these indexes to quickly find the location of a required book, if he/she knows the author(s) or title of the book.

Each index (access structure) offers an access path to records. Some types of indexes, called secondary access paths, do not affect the physical placement of records on disk; rather, they provide alternative search paths for locating the records efficiently based on the indexing fields. Other types of indexes can only be constructed on a file with a particular primary organisation.

The focus of our study in this chapter will be on the following:

- Primary indexes
- Clustering indexes
- Secondary indexes
- Multilevel indexes
- B-tree and B+ tree structures.

It must be emphasised that different indexes have their own advantages and disadvantages. There is no universally efficient index. Each technique is best suited for a particular type of database application.

The merits of indexes can be measured in the following aspects:

- **Access types:** The kind of access methods that can be supported efficiently (e.g. value-based search or range search).
- **Access time:** Time needed to locate a particular record or a set of records.
- **Insertion efficiency:** How efficient an insertion operation is.
- **Deletion efficiency:** How efficient a deletion operation is.
- **Storage overhead:** The additional storage requirement of an index structure.

It is worth noting that a file of records can have more than one index, just like for books there can be different indexes such as author index and title index.

An index access structure is usually constructed on a single field of a record in a file, called an indexing field. Such an index typically stores each value of the indexing field, along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are usually sorted (ordered) so that we can perform an efficient binary search on the index.

To give you an intuitive understanding of an index, we look at library indexes again. For example, an author index in a library will have entries for all authors whose books are stored in the library. AUTHOR is the indexing field and all the names are sorted according to alphabetical order. For a particular author, the index entry will contain locations (i.e. pointers) of all the books this author has written. If you know the name of the author, you will be able to use this index to find his/her books quickly. What happens if you do not have an index to use? This is similar to using a heap file and linear search. You will have to browse through the whole library looking for the book.

An index file is much smaller than the data file, and therefore searching the index using a binary search can be carried out quickly. Multilevel indexing goes one step further in the sense that it eliminates the need for a binary search, by building indexes to the index itself. We will be discussing these techniques later on in the chapter.

Organising files and records on disk

In this section, we will briefly define the concepts of records, record types and files. Then we will discuss various techniques for organising file records on disk.

Record and record type

A record is a unit which data is usually stored in. Each record is a collection of related data items, where each item is formed of one or more bytes and corresponds to a particular field of the record. Records usually describe entities and their attributes. A collection of field (item) names and their corresponding data types constitutes a record type. In short, we may say that a record type corresponds to an entity type and a record of a specific type represents an instance of the corresponding entity type.

The following is an example of a record type and its record:

Record Type Name	Field Names	Data Types
STUDENT	ID NUMBER	integer
	NAME	string of characters
	ADDRESS	string of characters
	COURSE	string of characters
	LEVEL	integer

A specific record of the STUDENT type:

STUDENT(9901536, “James Bond”, “1 Bond Street, London”, “Intelligent Services”, 9)

Fixed-length and variable-length records in files

A file basically contains a sequence of records. Usually all records in a file are of the same record type. If every record in the file has the same size in bytes, the records are called fixed-length records. If records in the file have different sizes, they are called variable-length records.

Variable-length records may exist in a file for the following reasons:

- Although they may be of the same type, one or more of the fields may be of varying length. For instance, students’ names are of different lengths.
- The records are of the same type, but one or more of the fields may be a repeating field with multiple values.
- If one or more fields are optional, not all records (of the same type) will have values for them.
- A file may contain records of different record types. In this case, records in the file are likely to have different sizes.

For fixed-length records, the exact size of each record can be determined in advance. As a result, they can easily be allocated to blocks (a block is the unit of transfer of data between main memory and disk). Also, we can identify the starting byte position of each field relative to the starting position of the record, because each of such records has the same fields, and the field lengths are fixed and known beforehand. This provides us with a simple way to find field values of a record in the file.

For records with variable-length fields, we may not know the exact lengths of those fields in advance. In order to determine the bytes that are needed to

accommodate those fields, we can use special separator characters, which do not appear in any field value (such as ~, @, or !), to terminate the variable-length fields. An alternative to this approach is to store the exact length of a variable-length field explicitly in the record concerned.

A repeating field needs one separator character to separate the repeating values of the field, and another separator character to indicate termination of the field. In short, we need to find out the exact size of a variable-length record before allocating it to a block or blocks. It is also apparent that programs that process files of variable-length records will be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.

We have seen that fixed-length records have advantages over variable-length records with respect to storage and retrieving a field value within the record. In some cases, therefore, it is possible and may also be advantageous to use a fixed-length record structure to represent a record that may logically be of variable length.

For example, we can use a fixed-length record structure that is large enough to accommodate the largest variable-length record anticipated in the file. For a repeating field, we could allocate as many spaces in each record as the maximum number of values that the field can take. In the case of optional fields, we may have every field included in every file record. If an optional field is not applicable to a certain record, a special null value is stored in the field. By adopting such an approach, however, it is likely that a large amount of space will be wasted in exchange for easier storage and retrieval.

Allocating records to blocks

The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and main memory. When the record size is smaller than the block size, a block can accommodate many such records. If a record has too large a size to be fitted in one block, two or more blocks will have to be used.

In order to enable further discussions, suppose the size of the block is B bytes, and a file contains fixed-length records of size R bytes. If B # R, then we can allocate bfr = #(B/R)# records into one block, where #(x)# is the so-called floor function which rounds the value x down to the next integer. The value bfr is defined as the blocking factor for the file.

In general, R may not divide B exactly, so there will be some leftover spaces in each block equal to B – (bfr * R) bytes .

If we do not want to waste the unused spaces in the blocks, we may choose to store part of a record in them and the rest of the record in another block. A pointer at the end of the first block points to the block containing the other part of the record, in case it is not the next consecutive block on disk. This

organisation is called 'spanned', because records can span more than one block. If records are not allowed to cross block boundaries, the organisation is called 'unspanned'.

Unspanned organisation is useful for fixed-length records with a length R # B. It makes each record start at a known location in the block, simplifying record processing. For variable-length records, either a spanned or unspanned organisation can be used. It is normally advantageous to use spanning to reduce the wasted space in each block.

For variable-length records using spanned organisation, each block may store a different number of records. In this case, the blocking factor bfr represents the average number of records per block for the file. We can then use bfr to calculate the number of blocks (b) needed to accommodate a file of r records:

b = #(r/bfr)# blocks

where #(x)# is the so-called ceiling function which rounds the value x up to the nearest integer.

It is not difficult to see that if the record size R is bigger than the block size B, then spanned organisation has to be used.

File headers

A file normally contains a file header or file descriptor providing information which is needed by programs that access the file records. The contents of a header contain information that can be used to determine the disk addresses of the file blocks, as well as to record format descriptions, which may include field lengths and order of fields within a record for fixed-length unspanned records, separator characters, and record type codes for variable-length records.

To search for a record on disk, one or more blocks are transferred into main memory buffers. Programs then search for the desired record or records within the buffers, using the header information.

If the address of the block that contains the desired record is not known, the programs have to carry out a linear search through the blocks. Each block is loaded into a buffer and checked until either the record is found or all the blocks have been searched unsuccessfully (which means the required record is not in the file). This can be very time-consuming for a large file. The goal of a good file organisation is to locate the block that contains a desired record with a minimum number of block transfers.

Operations on files

Operations on files can usually be grouped into retrieval operations and update operations. The former do not change anything in the file, but only locate

certain records for further processing. The latter change the file by inserting or deleting or modifying some records.

Typically, a DBMS can issue requests to carry out the following operations (with assistance from the operating-system file/disk managers):

- **Find (or Locate):** Searches for the first record satisfying a search condition (a condition specifying the criteria that the desired records must satisfy). Transfers the block containing that record into a buffer (if it is not already in main memory). The record is located in the buffer and becomes the current record (ready to be processed).
- **Read (or Get):** Copies the current record from the buffer to a program variable. This command may also advance the current record pointer to the next record in the file.
- **FindNext:** Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a buffer, and the record becomes the current record.
- **Delete:** Deletes the current record and updates the file on disk to reflect the change requested.
- **Modify:** Modifies some field values for the current record and updates the file on disk to reflect the modification.
- **Insert:** Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a buffer, writing the (new) record into the buffer, and writing the buffer to the disk file to reflect the insertion.
- **FindAll:** Locates all the records in the file that satisfy a search condition.
- **FindOrdered:** Retrieves all the records in the file in a specified order.
- **Reorganise:** Rearranges records in a file according to certain criteria. An example is the 'sort' operation, which organises records according to the values of specified field(s).
- **Open:** Prepares a file for access by retrieving the file header and preparing buffers for subsequent file operations.
- **Close:** Signals the end of using a file.

Before we move on, two concepts must be clarified:

- **File organisation:** This concept generally refers to the organisation of data into records, blocks and access structures. It includes the way in which records and blocks are placed on disk and interlinked. Access structures are particularly important. They determine how records in a file are interlinked logically as well as physically, and therefore dictate what access methods may be used.

- **Access method:** This consists of a set of programs that allow operations to be performed on a file. Some access methods can only be applied to files organised in certain ways. For example, indexed access methods can only be used in indexed files.

In the following sections, we are going to study three file organisations, namely heap files, sorted files and hash files, and their related access methods.

Review question 1

1. What are the different reasons for having variable-length records?
2. How can we determine the sizes of variable-length records with variable-length fields when allocating them to disk?
3. When is it most useful to use fixed-length representations for a variable-length record?
4. What information is stored in file headers?
5. What is the difference between a file organisation and an access method?

File organisations - organising records in files

Heap file organisation

The heap file organisation is the simplest and most basic type of organisation. In such an organisation, records are stored in the file in the order in which they are inserted, and new records are always placed at the end of the file.

The insertion of a new record is very efficient. It is performed in the following steps:

- The last disk block of the file is copied into a buffer.
- The new record is added.
- The block in the buffer is then rewritten back to the disk.

Remember the address of the last file block can always be kept in the file header.

The search for a record based on a search condition involves a linear search through the file block by block, which is often a very inefficient process. If only one record satisfies the condition then, on average, half of the file blocks will have to be transferred into main memory before the desired record is found. If no records or several records satisfy the search condition, all blocks will have to be transferred.

To modify a record in a file, a program must:

- find it;
- transfer the block containing the record into a buffer;

- make the modifications in the buffer;
- then rewrite the block back to the disk.

As we have seen, the process of finding the record can be time-consuming.

To remove a record from a file, a program must:

- find it;
- transfer the block containing the record into a buffer;
- delete the record from the buffer;
- then rewrite the block back to the disk.

Again, the process of finding the record can be time-consuming.

Physical deletion of a record leaves unused space in the block. As a consequence, a large amount of space may be wasted if frequent deletions have taken place. An alternative method to physically deleting a record is to use an extra bit (called a deletion marker) in all records. A record is deleted (logically) by setting the deletion marker to a particular value. A different value of the marker indicates a valid record (i.e. not deleted). Search programs will only consider valid records in a block. Invalid records will be ignored as if they have been physically removed.

No matter what deletion technique is used, a heap file will require regular reorganisation to reclaim the unused spaces due to record deletions. During such reorganisation, the file blocks are accessed consecutively and some records may need to be relocated to fill the unused spaces. An alternative approach to reorganisation is to use the space of deleted records when inserting new records. However, this alternative will require extra facilities to keep track of empty locations.

Both spanned and unspanned organisations can be used for a heap file of either fixed-length records or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a new record incorporating the required changes, because the modified record may not fit in its old position on disk.

Exercise 1

A file has r = 20,000 STUDENT records of fixed length, each record has the following fields: ID# (7 bytes), NAME (35 bytes), ADDRESS (46 bytes), COURSE (12 bytes), and LEVEL (1 byte). An additional byte is used as a deletion marker. This file is stored on the disk with the following characteristics: block size B = 512 bytes; inter-block gap size G = 128 bytes; number of blocks per track = 20; number of tracks per surface = 400. Do the following exercises:

1. Calculate the record size R.
2. Calculate the blocking factor bfr.

3. Calculate the number of file blocks b needed to store the records, assuming an unspanned organisation.

Exercise 2

Suppose we now have a new disk device which has the following specifications: block size B = 2400 bytes; inter-block gap size G = 600 bytes. The STUDENT file in Exercise 1 is stored on this disk, using unspanned organisation. Do the following exercises:

1. Calculate the blocking factor bfr.
2. Calculate the wasted space in each disk block because of the unspanned organisation.
3. Calculate the number of file blocks b needed to store the records.
4. Calculate the average number of block accesses needed to search for an arbitrary record in the file, using linear search.

Review question 2

Discuss the different techniques for record deletion.

Sorted sequential file organisation

Records in a file can be physically ordered based on the values of one of their fields. Such a file organisation is called a sorted file, and the field used is called the ordering field. If the ordering field is also a key field, then it is called the ordering key for the file.

The following figure depicts a sorted file organisation containing the STUDENT records:

	ID#	NAME	ADDRESS	COURSE	LEVEL
Block 1	9701654				
	9701890				
	.....				
	9702317				
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block 2	9702381				
	9702399				
	.....				
	9703478				
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block 3	9703501				
	9703569				
	.....				
	9801220				
.....					
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block b-1	9902318				
	9902449				
	.....				
	9903778				
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block b	9903791				
	9903799				
	.....				
	9903988				

The sorted file organisation has some advantages over unordered files, such as:

- Reading the records in order of the ordering field values becomes very efficient, because no sorting is required. (Remember, one of the common file operations is FindOrdered.)
- Locating the next record from the current one in order of the ordering field usually requires no additional block accesses, because the next record is often stored in the same block (unless the current record is the last one in the block).
- Retrieval using a search condition based on the value of the ordering field can be efficient when the binary search technique is used.

In general, the use of the ordering field is essential in obtaining the advantages.

Binary search algorithm

A binary search for disk files can be performed on the blocks rather than on the records. Suppose that:

- the file has b blocks numbered 1, 2, …, b;
- the records are ordered by ascending value of their ordering key;
- we are searching for a record whose ordering field value is K;
- disk addresses of the file blocks are available in the file header.

The search algorithm is described below in pseudo-codes:

Binary search based on an ordering key of a disk file

```
M ← 1; n ← b; (* b is the number of blocks, m and n are variables of integer types. *)
While (n >= m) do
        Begin i ← (m+n)/2; (*I is a variable of integer type. *)
        Read ith block from the file into a buffer;
        If K < ordering key field value of the first record in the block
                Then n ← i-1
        Else    if K > ordering key field value of the last record in the bock
                        Then m ← i+1
                Else    if the record with ordering key field value = K is in the buffer
                                Then go to found
                        Else go to notfound
        End;
Go to notfound
```

Explanations:

- The binary search algorithm always begins from the middle block in the file. The middle block is loaded into a buffer.
- Then the specified ordering key value K is compared with that of the first record and the last record in the buffer.
- If K is smaller than the ordering field value of the first record, then it means that the desired record must be in the first half of the file (if it is in the file at all). In this case, a new binary search starts in the upper half of the file and blocks in the lower half can be ignored.
- If K is bigger than the ordering field value of the last record, then it means that the desired record must be in the second half of the file (if it is in the file at all). In this case, a new binary search starts in the lower half of the file and blocks in the upper half can be ignored.
- If K is between the ordering field values of the first and last records, then it should be in the block already in the buffer. If not, it means the record is not in the file at all.

Referring to the example in the figure above, suppose we want to find a student's record whose ID number is 9701890. We further assume that there are five blocks in the file (i.e. the five blocks shown in the figure). Using the binary search, we start in block 3 and find that 9701890 (the specified ordering field value) is smaller than 9703501 (the ordering field value of the first record). Thus, we move to block 2 and read it into the buffer. Again, we find that 9701890 is smaller than 9702381 (the ordering field value of the first record of block 2). As a result, we read in block 1 and find 9701890 is between 9701654 and 9702317. If the record is in the file, it has to be in this block. By conducting a further search in the buffer, we can find the record.

Performance issues

The sorted file organisation can offer very efficient retrieval performance only if the search is based on the ordering field values. For example, the search for the following SQL query is efficient:

select NAME, ADDRESS from STUDENT

where ID# = 9701890;

If ID# is not in the condition, the linear search has to be used and there will be no performance advantages.

Update operations (e.g. insertion and deletion) are expensive for an ordered file because we must always maintain the order of records in the file.

To insert a new record, we must first find its correct position among existing records in the file, according to its ordering field value. Then a space has to be made at that location to store it. This involves reorganisation of the file, and for

a large file it can be very time-consuming. The reason is that on average, half the records of the file must be moved to make the space. For record deletion, the problem is less severe, if deletion markers are used and the file is reorganised periodically.

One option for making insertion more efficient is to keep some unused space in each block for new records. However, once this space is used up, the original problem resurfaces.

The performance of a modification operation depends on two factors: first, the search condition to locate the record, and second, the field to be modified.

- If the search condition involves the ordering field, the efficient binary search can be used. Otherwise, we have to conduct a linear search.
- A non-ordering field value can be changed and the modified record can be rewritten back to its original location (assuming fixed-length records).
- Modifying the ordering field value means that the record may change its position in the file, which requires the deletion of the old record followed by the insertion of the modified one as a new record.

Reading the file records in order of the ordering field is efficient. For example, the operations corresponding to the following SQL query can be performed efficiently:

select ID#, NAME, COURSE from STUDENT

where LEVEL = 2

order by ID#;

The sorted file organisation is rarely used in databases unless a primary index structure is included with the file. The index structure can further improve the random access time based on the ordering field.

Review question 3

1. What major advantages do sorted files have over heap files?
2. What is the essential element in taking full advantage of the sorted organisation?
3. Describe the major problems associated with the sorted files.

Exercise 3

Suppose we have the same disk and STUDENT file as in Exercise 2. This time, however, we assume that the records are ordered based on ID# in the disk file. Calculate the average number of block accesses needed to search for a record with a given ID#, using binary search.

Hash file organisation

The hash file organisation is based on the use of hashing techniques, which can provide very efficient access to records based on certain search conditions. The search condition must be an equality condition on a single field called hash field (e.g. ID# = 9701890, where ID# is the hash field). Often the hash field is also a key field. In this case, it is called the hash key.

Hashing techniques

The principle idea behind the hashing technique is to provide a function h, called a hash function, which is applied to the hash field value of a record and computes the address of the disk block in which the record is stored. A search for the record within the block can be carried out in a buffer, as always. For most records, we need only one block transfer to retrieve that record.

Suppose K is a hash key value, the hash function h will map this value to a block address in the following form:

h(K) = address of the block containing the record with the key value K

If a hash function operates on numeric values, then non-numeric values of the hash field will be transformed into numeric ones before the function is applied.

The following are two examples of many possible hash functions:

- Hash function h(K) = K mod M: This function returns the remainder of an integer hash field value K after division by integer M. The result of the calculation is then used as the address of the block holding the record.
- A different function may involve picking some digits from the hash field value (e.g. the 2nd, 4th and 6th digits from ID#) to form an integer, and then further calculations may be performed using the integer to generate the hash address.

The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses. The reason is that the hash field space (the number of possible values a hash field can take) is usually much larger than the address space (the number of available addresses for records). For example, the hash function h(K) = K mod 7 will hash 15 and 43 to the same address 1 as shown below:

43 mod 7 = 1

15 mod 7 = 1

A collision occurs when the hash field value of a new record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position. The process of finding another position is called collision resolution.

The following are commonly used techniques for collision resolution:

- **Open addressing:** If the required position is found occupied, the program will check the subsequent positions in turn until an available space is located.
- **Chaining:** For this method, some spaces are kept in the disk file as overflow locations. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address needs to be maintained.
- **Multiple hashing:** If a hash function causes a collision, a second hash function is applied. If it still does not resolve the collision, we can then use open addressing or apply a third hash function and so on.

Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. Detailed discussions on them are beyond the scope of this chapter. Interested students are advised to refer to textbooks dealing with data structures.

In general, the goal of a good hash function is to distribute the records uniformly over the address space and minimise collisions, while not leaving many unused locations. Also remember that a hash function should not involve complicated computing. Otherwise, it may take a long time to produce a hash address, which will actually hinder performance.

External hashing

Hashing for disk files is called external hashing. To suit the characteristics of disk storage, the hash address space is made of buckets. Each bucket consists of either one disk block or a cluster of contiguous (neighbouring) blocks, and can accommodate a certain number of records.

A hash function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the relative bucket number into the corresponding disk block address.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing any problem. If the collision problem does occur when a bucket is filled to its capacity, we can use a variation of the chaining method to resolve it. In this situation, we maintain a pointer in each bucket to a linked list of overflow records for the bucket. The pointers in the linked list should be record pointers, which include both a block address and a relative record position within the block.

Dynamic hashing

Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. These are called dynamic hash functions.

Extendable hashing is one form of dynamic hashing, and it works in the following way:

- We choose a hash function that is uniform and random. It generates values over a relatively large range.
- The hash addresses in the address space (i.e. the range) are represented by d-bit binary integers (typically d = 32). As a result, we can have a maximum of 232 (over 4 billion) buckets.
- We do not create 4 billion buckets at once. Instead, we create them on demand, depending on the size of the file. According to the actual number of buckets created, we use the corresponding number of bits to represent their address. For example, if there are four buckets at the moment, we just need 2 bits for the addresses (i.e. 00, 01, 10 and 11).
- At any time, the actual number of bits used (denoted as i and called global depth) is between 0 (for one bucket) and d (for maximum 2d buckets).
- Value of i grows or shrinks with the database, and the i binary bits are used as an offset into a table of bucket addresses (called a directory). In search algorithm above, 3 bits are used as the offset (i.e. 000, 001, 010, ..., 110, 111).
- The offsets serve as indexes pointing to the buckets in which the corresponding records are held. For example, if the first 3 bits of a hash value of a record are 001, then the record is in the bucket pointed by the 001 entry.
- It must be noted that there does not have to be a distinct bucket for each of the 2i directory locations. Several directory locations (i.e. entries) with the same first j bits (j <= i) for their hash values may contain the same bucket address (i.e. point to the same bucket) if all the records that hash to these locations fit in a single bucket.
- j is called the local depth stored with each bucket. It specifies the number of bits on which the bucket contents are based. In search algorithm above, for example, the middle two buckets contain records which have been hashed based on the first 2 bits of their hash values (i.e. starting with 01 and 10), while the global depth is 3.

The value of i can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory table. Doubling is needed if a bucket whose local depth j is equal to the global depth i, overflows. Halving occurs if i > j for all the buckets after some deletions occur.

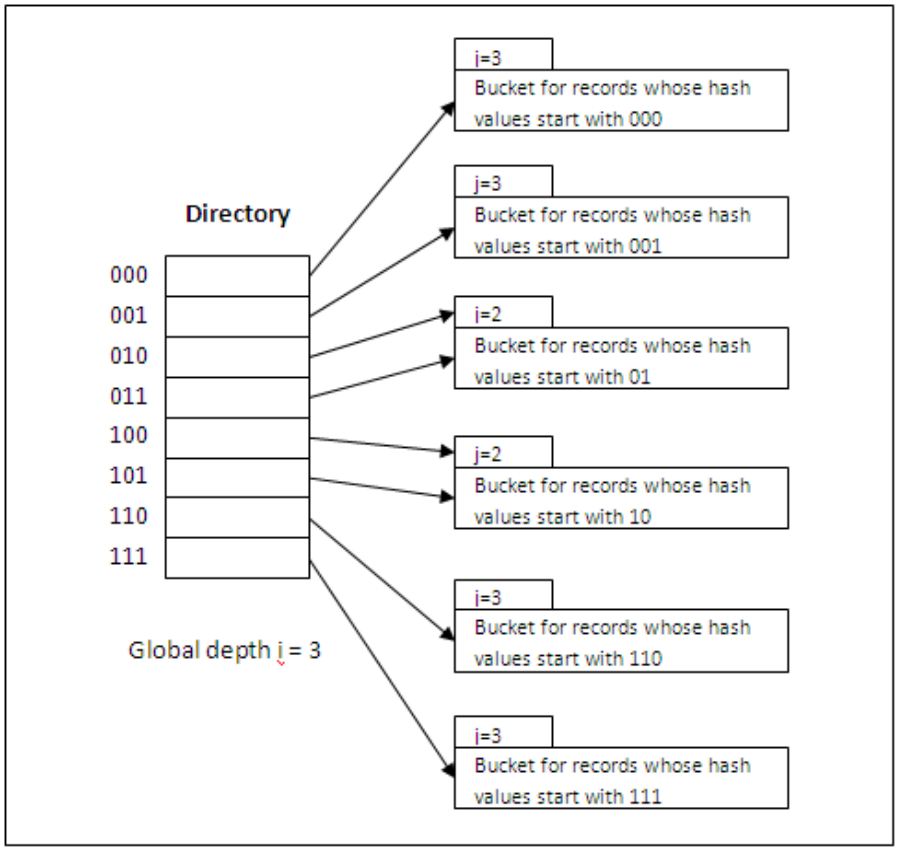
Retrieval - To find the bucket containing the search key value K:

- Compute h(K).
- Take the first i bits of h(K).
- Look at the corresponding table entry for this i-bit string.
- Follow the bucket pointer in the table entry to retrieve the block.

Insertion - To add a new record with the hash key value K:

- Follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, place the record in it.
- If the bucket is full, we must split the bucket and redistribute the records.
- If a bucket is split, we may need to increase the number of bits we use in the hash.

To illustrate bucket splitting (see the figure below), suppose that a new record to be inserted causes overflow in the bucket whose hash values start with 01 (the third bucket). The records in that bucket will have to be redistributed among two buckets: the first contains all records whose hash values start with 010, and the second contains all those whose hash values start with 011. Now the two directory entries for 010 and 011 point to the two new distinct buckets. Before the split, they point to the same bucket. The local depth of the two new buckets is 3, which is one more than the local depth of the old bucket.



If a bucket that overflows and is split used to have a local depth j equal to the global depth i of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. In the above figure, for example, if the bucket for records whose hash values start with 111 overflows, the two new buckets need a directory with global depth i = 4, because the two buckets are now labelled 1110 and 1111, and hence their local depths are both 4. The directory size is doubled and each of the other original entries in the directory is also split into two, both with the same pointer as the original entries.

Deletion may cause buckets to be merged and the bucket address directory may have to be halved.

In general, most record retrievals require two block accesses – one to the directory and the other to the bucket. Extendable hashing provides performance that does not degrade as the file grows. Space overhead is minimal, because no buckets need be reserved for future use. The bucket address directory only contains one pointer for each hash value of current prefix length (i.e. the global

depth). Potential drawbacks are that we need to have an extra layer in the structure (i.e. the directory) and this adds more complexities.

Performance issues

Hashing provides the fastest possible access for retrieving a record based on its hash field value. However, search for a record where the hash field value is not available is as expensive as in the case of a heap file.

Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in the overflow, we simply remove it from the linked list.

To insert a new record, first, we use the hash function to find the address of the bucket the record should be in. Then, we insert the record into an available location in the bucket. If the bucket is full, we will place the record in one of the locations for overflow records.

The performance of a modification operation depends on two factors: first, the search condition to locate the record, and second, the field to be modified.

- If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hash function. Otherwise, we must perform a linear search.
- A non-hash field value can be changed and the modified record can be rewritten back to its original bucket.
- Modifying the hash field value means that the record may move to another bucket, which requires the deletion of the old record followed by the insertion of the modified one as a new record.

One of the most notable drawbacks of commonly used hashing techniques (as presented above) is that the amount of space allocated to a file is fixed. In other words, the number of buckets is fixed, and the hashing methods are referred to as static hashing. The following problems may occur:

- The number of buckets is fixed, but the size of the database may grow.
- If we create too many buckets, a large amount of space may be wasted.
- If there are too few buckets, collisions will occur more often.

As the database grows over time, we have a few options:

- Devise a hash function based on current file size. Performance degradation will occur as the file grows, because the number of buckets will appear to be too small.
- Devise a hash function based on the expected file size. This may mean the creation of too many buckets initially and cause some space wastage.

- Periodically reorganise the hash structure as file grows. This requires devising new hash functions, recomputing all addresses and generating new bucket assignments. This can be costly and may require the database to be shut down during the process.

Review question 4

1. What is a collision and why is it unavoidable in hashing?
2. How does one cope with collisions in hashing?
3. What are the most notable problems that static hashing techniques have?

Suppose we have a VIDEO file holding VIDEO_TAPE records for a rental shop. The records have TAPE# as the hash key with the following values: 2361, 3768, 4684, 4879, 5651, 1829, 1082, 7107, 1628, 2438, 3951, 4758, 6967, 4989, 9201. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and can store up to two records. Load the above records into the file in the given order, using the hash function h(K) = K mod 8. The open addressing method is used to deal with any collision.

1. Show how the records are stored in the buckets.
2. Calculate the average number of block accesses needed for a random retrieval based on TAPE#.

Activity 1 - Record Deletion in Extensible Hashing

The purpose of this activity is to enable you to consolidate what you have learned about extendable hashing. You should work out yourself how to delete a record from a file with the extendable hashing structure. What happens if a bucket becomes empty due to the deletion of its last record? When the size of the directory may need to be halved?

Single-level ordered indexes

There are several types of single-level ordered indexes. A primary index is an index specified on the ordering key field of a sorted file of records. If the records are sorted not on the key field but on a non-key field, an index can still be built that is called a clustering index. The difference lies in the fact that different records have different values in the key field, but for a non-key field, some records may share the same value. It must be noted that a file can have at most one physical ordering field. Thus, it can have at most one primary index or one clustering index, but not both.

A third type of index, called a secondary index, can be specified on any non-ordering field of a file. A file can have several secondary indexes in addition to its primary access path (i.e. primary index or clustering index). As we mentioned earlier, secondary indexes do not affect the physical organisation of records.

Primary indexes

A primary index is built for a file (the data file) sorted on its key field, and itself is another sorted file (the index file) whose records (index records) are of fixed-length with two fields. The first field is of the same data type as the ordering key field of the data file, and the second field is a pointer to a disk block (i.e., the block address).

The ordering key field is called the primary key of the data file. There is one index entry (i.e., index record) in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values. We use the following notation to refer to an index entry i in the index file:

<K(i), P(i)>

K(i) is the primary key value, and P(i) is the corresponding pointer (i.e. block address).

For example, to build a primary index on the sorted file shown below (this is the same STUDENT file we saw in exercise 1), we use the ID# as primary key, because that is the ordering key field of the data file:

	ID#	NAME	ADDRESS	COURSE	LEVEL
Block 1	9701654				
	9701890				
	.....				
	9702317				
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block 2	9702381				
	9702399				
	.....				
	9703478				
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block 3	9703501				
	9703569				
	.....				
	9801220				
.....					
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block b-1	9902318				
	9902449				
	.....				
	9903778				
	ID#	NAME	ADDRESS	COURSE	LEVEL
Block b	9903791				
	9903799				
	.....				
	9903988				

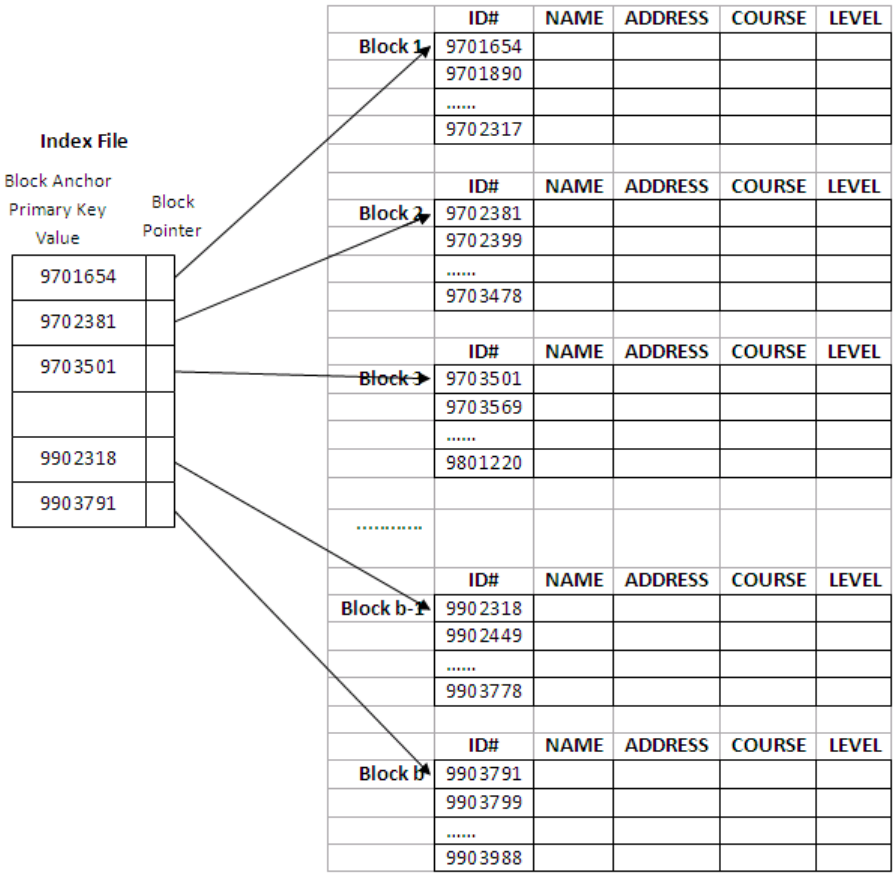
Each entry in the index has an ID# value and a pointer. The first three index entries of such an index file are as follows:

<K(1) = 9701654, P(1) = address of block 1>

<K(2) = 9702381, P(2) = address of block 2>

<K(3) = 9703501, P(3) = address of block 3>

The figure below depicts this primary index. The total number of entries in the index is the same as the number of disk blocks in the data file. In this example, there are b blocks.



The first record in each block of the data file is called the anchor record of that block. A variation to such a primary index scheme is that we could use the last record of a block as the block anchor. However, the two schemes are very similar and there is no significant difference in performance. Thus, it is sufficient to discuss just one of them.

A primary index is an example of a sparse index, in the sense that it contains an entry for each disk block rather than for every record in the data file. A dense index, on the other hand, contains an entry for every data record. A dense index does not require the data file to be a sorted file. Instead, it can be built on any file organisation (typically, a heap file).

By definition, an index file is just a special type of data file of fixed-length records with two fields. We use the term 'index file' to refer to data files storing index entries. The general term 'data file' is used to refer to files containing the actual data records such as STUDENT.

Performance issues

The index file for a primary index needs significantly fewer blocks than does the file for data records for the following two reasons:

- There are fewer index entries than there are records in the data file, because an entry exists for each block rather than for each record.
- Each index entry is typically smaller in size than a data record because it has only two fields. Consequently, more index entries can fit into one block. A binary search on the index file hence requires fewer block accesses than a binary search on the data file.

If a record whose primary key value is K is in the data file, then it has to be in the block whose address is P(i), where K(i) <= K < K(i+1). The ith block in the data file contains all such records because of the physical ordering of the records based on the primary key field. For example, look at the first three entries in the index in the last figure.

<K(1) = 9701654, P(1) = address of block 1>

<K(2) = 9702381, P(2) = address of block 2>

<K(3) = 9703501, P(3) = address of block 3>

The record with ID#=9702399 is in the 2nd block because K(2) <= 9702399 < K(3). In fact, all the records with an ID# value between K(2) and K(3) must be in block 2, if they are in the data file at all.

To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then use the block address contained in the pointer P(i) to retrieve the data block. The following example explains the performance improvement, in terms of the number of block accesses that can be obtained through the use of primary index.

Example 1

Suppose that we have an ordered file with r = 40,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed-length and are unspanned, with a record size R = 100 bytes. The blocking factor for the data file

would be bfr = #(B/R)## = #(1024/100)## = 10 records per block. The number of blocks needed for this file is b = #(r/bfr)## = #(40000/10)## = 4000 blocks. A binary search on the data file would need approximately #(log2b)## = #(log24000)## = 12 block accesses.

Now suppose that the ordering key field of the file is V = 11 bytes long, a block pointer (block address) is P = 8 bytes long, and a primary index has been constructed for the file. The size of each index entry is Ri = (11 + 8) = 19 bytes, so the blocking factor for the index file is bfr i = #(B/Ri)## = #(1024/19)#

= 53 entries per block. The total number of index entries ri is equal to the number of blocks in the data file, which is 4000. Thus, the number of blocks needed for the index file is bi = #(ri/bfri)### = #(4000/53)### = 76 blocks. To perform a binary search on the index file would need #(log2bi)### = #(log276)### = 7 block accesses. To search for the actual data record using the index, one additional block access is needed. In total, we need 7 + 1 = 8 block accesses, which is an improvement over binary search on the data file, which required 12 block accesses.

A major problem with a primary index is insertion and deletion of records. We have seen similar problems with sorted file organisations. However, they are more serious with the index structure because, if we attempt to insert a record in its correct position in the data file, we not only have to move records to make space for the newcomer but also have to change some index entries, since moving records may change some block anchors. Deletion of records causes similar problems in the other direction.

Since a primary index file is much smaller than the data file, storage overhead is not a serious problem.

Exercise 4

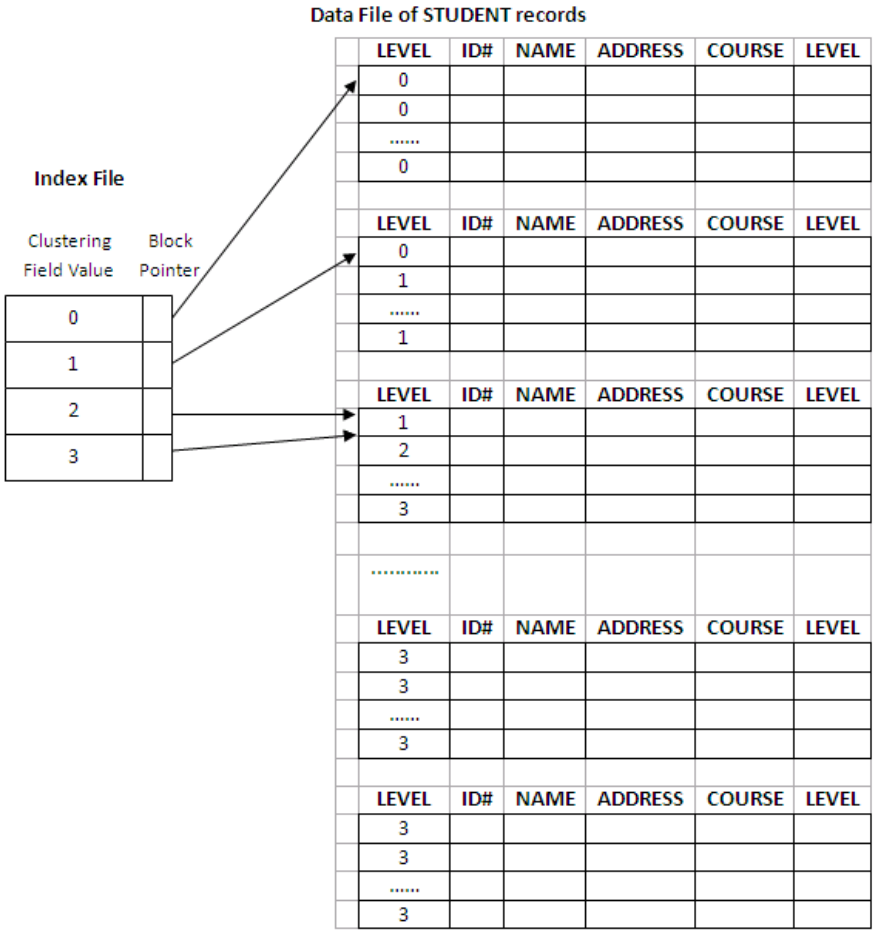
Consider a disk with block size B = 512 bytes. A block pointer is P = 8 bytes long, and a record pointer is Pr = 9 bytes long. A file has r = 50,000 STUDENT records of fixed-size R = 147 bytes. In the file, the key field is ID#, whose length V = 12 bytes. Answer the following questions:

1. If an unspanned organisation is used, what are the blocking factor bfr and the number of file blocks b?
2. Suppose that the file is ordered by the key field ID# and we want to construct a primary index on ID#. Calculate the index blocking factor bfri.
3. What are the number of first-level index entries and the number of first-level index blocks?
4. Determine the number of block accesses needed to search for and retrieve a record from the file using the primary index, if the indexing field value is given.

Clustering indexes

If records of a file are physically ordered on a non-key field which may not have a unique value for each record, that field is called the clustering field. Based on the clustering field values, a clustering index can be built to speed up retrieval of records that have the same value for the clustering field. (Remember the difference between a primary index and a clustering index.)

A clustering index is also a sorted file of fixed-length records with two fields. The first field is of the same type as the clustering field of the data file, and the second field is a block pointer. There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the first block in the data file that holds at least one record with that value for its clustering field. The figure below illustrates an example of the STUDENT file (sorted by their LEVEL rather than ID#) with a clustering index:



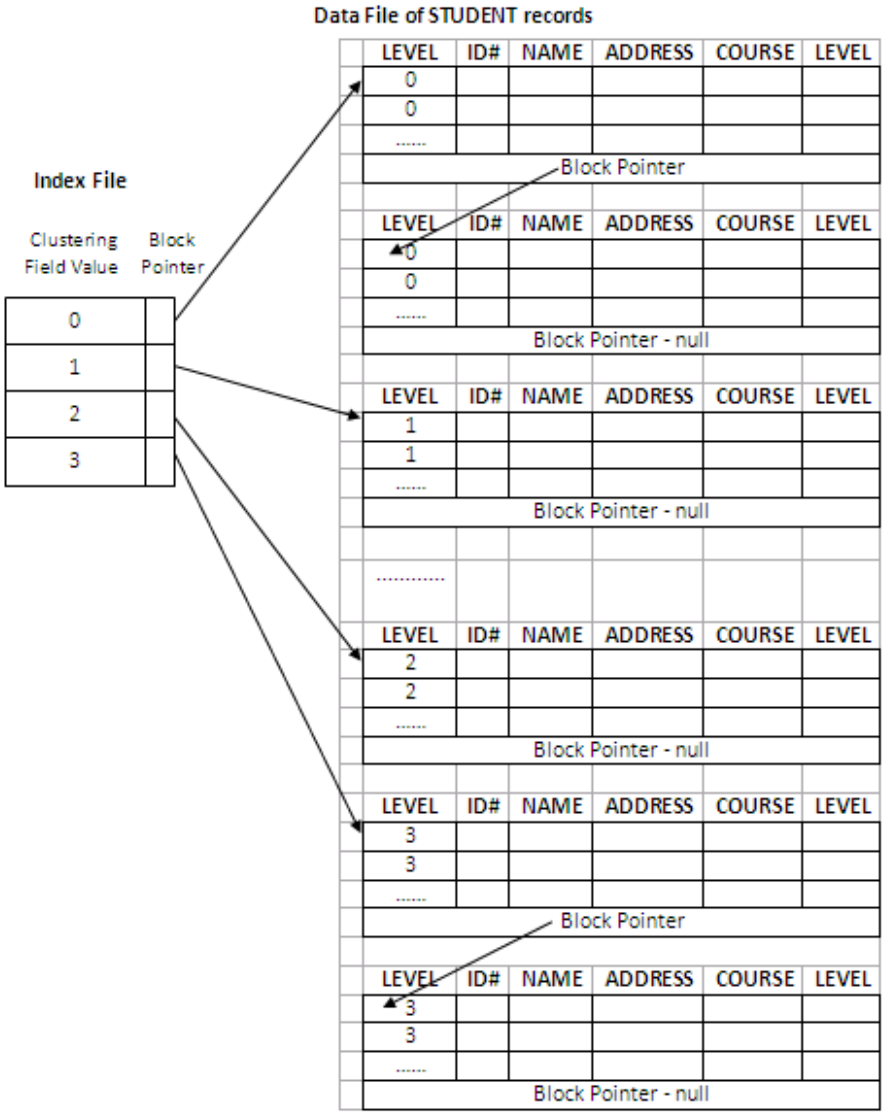
In the above figure, there are four distinct values for LEVEL: 0, 1, 2 and 3. Thus, there are four entries in the clustering index. As can be seen from the figure, many different records have the same LEVEL number and can be stored in different blocks. Both LEVEL 2 and LEVEL 3 entries point to the third block, because it stores the first record for LEVEL 2 as well as LEVEL 3 students. All other blocks following the third block must contain LEVEL 3 records, because all the records are ordered by LEVEL.

Performance issues

Performance improvements can be obtained by using the index to locate a record. However, the record insertion and deletion still causes similar problems to those in primary indexes, because the data records are physically ordered.

To alleviate the problem of insertion, it is common to reserve a whole block for each distinct value of the clustering field; all records with that value are placed in the block. If more than one block is needed to store the records for a particular value, additional blocks are allocated and linked together. To link blocks, the last position in a block is reserved to hold a block pointer to the next block. If there is no following block, then the block pointer will have null value.

Using this linked-blocks structure, no records with different clustering field values can be stored in the same block. It also makes insertion and deletion more efficient than without the linked structure. More blocks will be needed to store records and some spaces may be wasted. That is the price to pay for improving insertion efficiency. The figure below explains the scheme:



A clustering index is another type of sparse index, because it has an entry for each distinct value of the clustering field rather than for every record in the file.

Secondary indexes

A secondary index is a sorted file of records (either fixed-length or variable-length) with two fields. The first field is of the same data type as an indexing

field (i.e. a non-ordering field on which the index is built). The second field is either a block pointer or a record pointer. A file may have more than one secondary index.

In this section, we consider two cases of secondary indexes:

- The index access structure constructed on a key field.
- The index access structure constructed on a non-key field.

Index on key field

Before we proceed, it must be emphasised that a key field is not necessarily an ordering field. In the case of a clustering index, the index is built on a non-key ordering field.

When the key field is not the ordering field, a secondary index can be constructed on it where the key field can also be called a secondary key (in contrast to a primary key where the key field is used to build a primary index). In such a secondary index, there is one index entry for each record in the data file, because the key field (i.e. the indexing field) has a distinct value for every record. Each entry contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself (a pointer to an individual record consists of the block address and the record's position in that block).

A secondary index on a key field is a dense index, because it includes one entry for every record in the data file.

Performance issues

We again use notation <K(i), P(i)> to represent an index entry i. All index entries are ordered by value of K(i), and therefore a binary search can be performed on the index.

Because the data records are not physically ordered by values of the secondary key field, we cannot use block anchors as in primary indexes. That is why an index entry is created for each record in the data file, rather than for each block. P(i) is still a block pointer to the block containing the record with the key field value K(i). Once the appropriate block is transferred to main memory, a further search for the desired record within that block can be carried out.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, much greater improvement in search time for an arbitrary record can be obtained by using the secondary index, because we would have to do a linear search on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main data file, even if the index did not exist.

Example 2 explains the improvement in terms of the number of blocks accessed when a secondary index is used to locate a record.

Example 2

Consider the file in Example 1 with r = 40,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed-length and are unspanned, with a record size R = 100 bytes. As calculated previously, this file has b = 4000 blocks. To do a linear search on the file, we would require b/2 = 4000/2 = 2000 block accesses on average to locate a record.

Now suppose that we construct a secondary index on a non-ordering key field of the file that is V = 11 bytes long. As in Example 1, a block pointer is P = 8 bytes long. Thus, the size of an index entry is Ri = (11 + 8) = 19 bytes, and the blocking factor for the index file is bfri = #(B/Ri)## = # (1024/19) ## = 53 entries per block. In a dense secondary index like this, the total number of index entries ri is equal to the number of records in the data file, which is 40000. The number of blocks needed for the index is hence bi = #(ri/bfri)## = #(40000/53)## = 755 blocks. Compare this to the 76 blocks needed by the sparse primary index in Example 1.

To perform a binary search on the index file would need #(log2bi)## = #(log2755)## = 10 block accesses. To search for the actual data record using the index, one additional block access is needed. In total, we need 10 + 1 = 11 block accesses, which is a huge improvement over the 2000 block accesses needed on average for a linear search on the data file.

Index on a non-key field

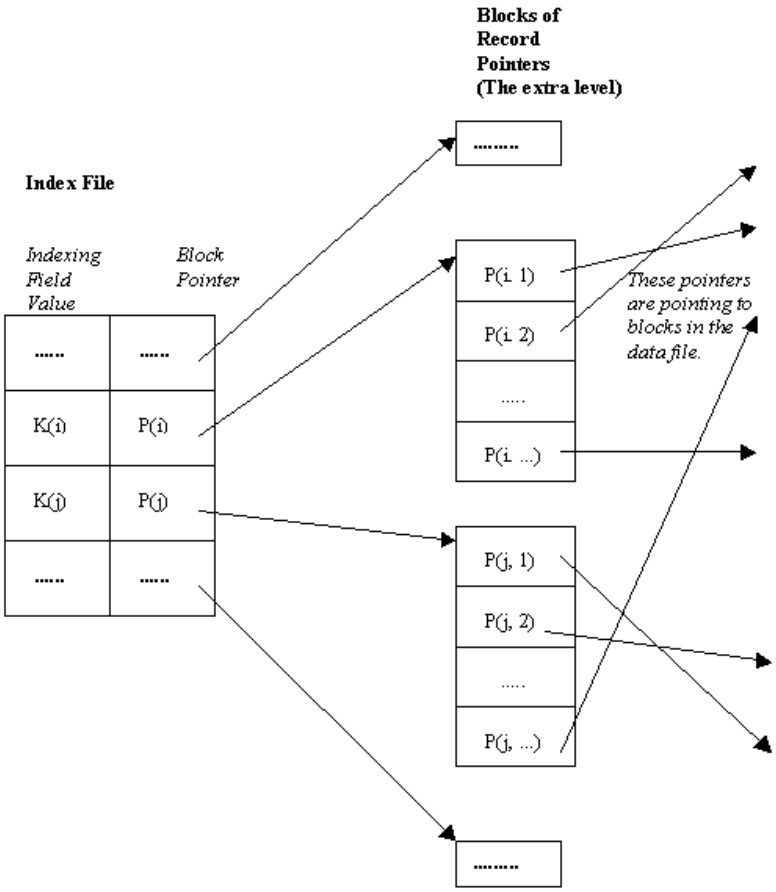
Using the same principles, we can also build a secondary index on a non-key field of a file. In this case, many data records can have the same value for the indexing field. There are several options for implementing such an index.

Option 1: We can create several entries in the index file with the same K(i) value – one for each record sharing the same K(i) value. The other field P(i) may have different block addresses, depending on where those records are stored. Such an index would be a dense index.

Option 2: Alternatively, we can use variable-length records for the index entries, with a repeating field for the pointer. We maintain a list of pointers in the index entry for K(i) – one pointer to each block that contains a record whose indexing field value equals K(i). In other words, an index entry will look like this: <K(i), [P(i, 1), P(i, 2), P(i, 3), ...]>. In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately.

Option 3: This is the most commonly adopted approach. In this option, we keep the index entries themselves at a fixed-length and have a single entry for each indexing field value. Additionally, an extra level of indirection is created to handle the multiple pointers. Such an index is a sparse scheme, and the pointer

P(i) in index entry <K(i), P(i)> points to a block of record pointers (this is the extra level); each record pointer in that block points to one of the data file blocks containing the record with value K(i) for the indexing field. If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single block, a linked list of blocks is used. The figure below explains this option.



From the above figure, it can be seen that instead of using index entries like <K(i), [P(i, 1), P(i, 2), P(i, 3), ……]> and <K(j), [P(j, 1), P(j, 2), P(j, 3), ……]> as in option 2, an extra level of data structure is used to store the record pointers. Effectively, the repeating field in the index entries of option 2 is removed, which makes option 3 more appealing.

It should be noted that a secondary index provides a logical ordering on the data records by the indexing field. If we access the records in order of the entries in the secondary index, the records can be retrieved in order of the indexing field values.

Summary of single-level ordered indexes

The following table summarises the properties of each type of index by comparing the number of index entries and specifying which indexes are dense or sparse and which use block anchors of the data file.

	Properties of Indexes		
Type of Index	**Number of Index Entries**	**Dense or Sparse**	**Using Block Anchor**
Primary	Equal to the number of blocks in the data file	Sparse	Yes
Clustering	Equal to the number of distinct indexing field values	Sparse	Yes if separate blocks are used for records with different indexing field values; No otherwise.
Secondary on a key field	Equal to the number of records in the data file	Dense	No
Secondary on a non-key field	Equal to the number of records for option 1; Equal to the number of distinct indexing field values for options 2 and 3	Dense for option 1; Sparse for options 2 and 3	No

Review question 5

1. What are a primary index, a clustering index and a secondary index?
2. What is a dense index and what is a sparse index?
3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?
4. Why does a secondary index need more storage space and longer search time than a primary index?
5. What is the major problem with primary and clustering indexes?
6. Why does a secondary index provide a logical ordering on the data records by the indexing field?

Exercise 5

Consider the same disk file as in Exercise 4. Answer the following questions (you need to utilise the results from Exercise 4):

- Suppose the key field ID# is NOT the ordering field, and we want to build a secondary index on ID#. Is the index a sparse one or a dense one, and why?
- What is the total number of index entries? How many index blocks are needed (if using block pointers to the data file)?
- Determine the number of block accesses needed to search for and retrieve a record from the file using the secondary index, if the indexing field value is given.

Multilevel indexes

The principle

The index structures that we have studied so far involve a sorted index file. A binary search is applied to the index to locate pointers to a block containing a record (or records) in the file with a specified indexing field value. A binary search requires #log2bi)# block accesses for an index file with bi blocks, because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why the log function to the base 2 is used.

The idea behind a multilevel index is to reduce the part of the index that we have to continue to search by bfri, the blocking factor for the index, which is almost always larger than 2. Thus, the search space can be reduced much faster. The value of bfri is also called the fan-out of the multilevel index, and we will refer to it by the notation fo. Searching a multilevel index requires #log2bi)# block accesses, which is a smaller number than for a binary search if the fan-out is bigger than 2.

The structure

A multilevel index considers the index file, which was discussed in the previous sections as a single-level ordered index and will now be referred to as the first (or base) level of the multilevel structure, as a sorted file with a distinct value for each K(i). Remember we mentioned earlier that an index file is effectively a special type of data file with two fields. Thus, we can build a primary index for an index file itself (i.e. on top of the index at the first level). This new index to the first level is called the second level of the multilevel index.

Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first-level index entries. The blocking factor bfri for the second level – and for all subsequent levels – is the same as that for the first-level index, because all index entries are of the same size; each has one field value and one block address. If the first level has r1 entries, and the blocking factor – which is also the fan-out – for the index is bfri

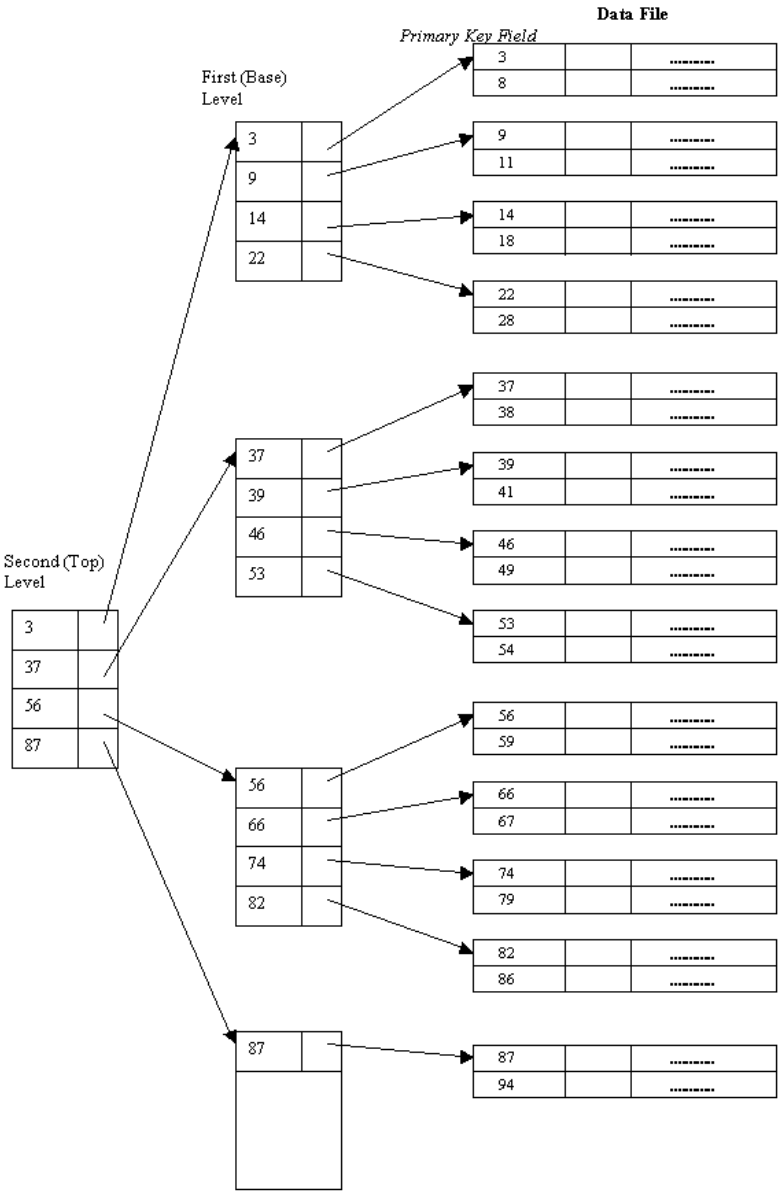
= fo, then the first level needs #(r1/fo)# blocks, which is therefore the number of entries r2 needed at the second level of the index.

The above process can be repeated and a third-level index can be created on top of the second-level one. The third level, which is a primary index for the second level, has an entry for each second-level block. Thus, the number of third-level entries is r3 = #(r2/fo)#. (Important note: We require a second level only if the first level needs more than one block of disk storage, and similarly, we require a third level only if the second level needs more than one block.)

We can continue the index-building process until all the entries of index level d fit in a single block. This block at the dth level is called the top index level (the first level is at the bottom and we work our way up). Each level reduces the number of entries at the previous level by a factor of fo – the multilevel index fan-out – so we can use the formula 1 # (r1/((fo)d)) to calculate d. Hence, a multilevel index with r1 first-level entries will need d levels, where d = #(logfo(r1))#.

Important

The multilevel structure can be used on any type of index, whether it is a primary, a clustering or a secondary index, as long as the first-level index has distinct values for K(i) and fixed-length entries. The figure below depicts a multilevel index built on top of a primary index.



It can be seen above that the data file is a sorted file on the key field. There is a primary index built on the data file. Because it has four blocks, a second-level index is created which fits in a single block. Thus, the second level is also the top level.

Performance issues

Multilevel indexes are used to improve the performance in terms of the number of block accesses needed when searching for a record based on an indexing field value. The following example explains the process.

Example 3

Suppose that the dense secondary index of Example 2 is converted into a multilevel index. In Example 2, we have calculated that the index blocking factor bfri = 53 entries per block, which is also the fan-out for the multilevel index. We also knew that the number of first-level blocks b1 = 755. Hence, the number of second-level blocks will be b2 = #(b1/fo)## = #(755/53)### = 15 blocks, and the number of third-level blocks will be b3 = #(b2/fo)### = #(15/53)### = 1 block. Now at the third level, because all index entries can be stored in a single block, it is also the top level and d = 3 (remember d = #(logfo(r1))

= #(log53(40000))### = 3, where r1 = r = 40000).

To search for a record based on a non-ordering key value using the multilevel index, we must access one block at each level plus one block from the data file. Thus, we need d + 1 = 3 + 1 = 4 block accesses. Compare this to Example 2, where 11 block accesses were needed when a single-level index and binary search were used.

Note

It should be noted that we could also have a multilevel primary index which could be sparse. In this case, we must access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first-level index without having to access the data block, since there is an index entry for every record in the file.

As seen earlier, a multilevel index improves the performance of searching for a record based on a specified indexing field value. However, the problems with insertions and deletions are still there, because all index levels are physically ordered files. To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, database developers often adopt a multilevel structure that leaves some space in each of its blocks for inserting new entries. This is called a dynamic multilevel index and is often implemented by using data structures called B-trees and B+ trees, which are to be studied in the following sections.

Review question 6

1. How does the multilevel indexing structure improve the efficiency of searching an index file?

2. What are the data file organisations required by multilevel indexing?

Exercise 6

Consider a disk with block size B = 512 bytes. A block pointer is P = 8 bytes long, and a record pointer is Pr = 9 bytes long. A file has r = 50,000 STUDENT records of fixed-size R = 147 bytes. The key field ID# has a length V = 12 bytes. (This is the same disk file as in Exercise 4 and 5. Previous results should be utilised.) Answer the following questions:

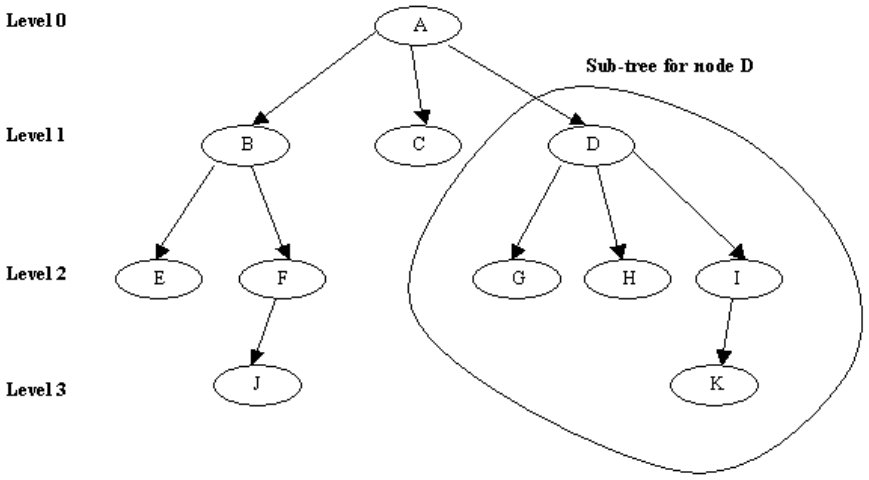
1. Suppose the key field ID# is the ordering field, and a primary index has been constructed (as in Exercise 4). Now if we want to make it into a multilevel index, what is the number of levels needed and what is the total number of blocks required by the multilevel index?
2. Suppose the key field ID# is NOT the ordering field, and a secondary index has been built (as in Exercise 5). Now if we want to make it into a multilevel index, what is the number of levels needed and what is the total number of blocks required by the multilevel index?

Dynamic multilevel indexes using B-trees and B+ trees

The tree data structure

B-trees and B+ trees are special types of the well-known tree data structure. A tree is formed of nodes. Each node in the tree, except for a special node called the root, has one parent node and any number (including zero) of child nodes. The root node has no parent. A node that does not have any child nodes is called a leaf node; a non-leaf node is called an internal node.

The level of a node is always one more than the level of its parent, with the level of the root node being zero. A sub-tree of a node consists of that node and all its descendant nodes (i.e. its child nodes, the child nodes of its child nodes, and so on). The figure below gives a graphical description of a tree structure:



Usually we display a tree with the root node at the top. One way to implement a tree is to have as many pointers in each node as there are child nodes of that node. In some cases, a parent pointer is also stored in each node. In addition to pointers, a node often contains some kind of stored information. When a multilevel index is implemented as a tree structure, this information includes the values of the data file's indexing field that are used to guide the search for a particular record.

Search trees

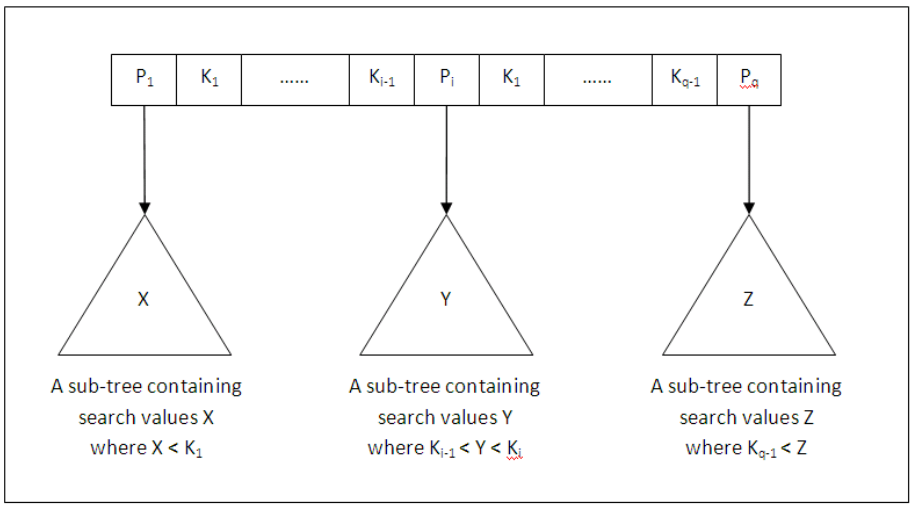
A search tree is a special type of tree that is used to guide the search for a record, given the value of one of its fields. The multilevel indexes studied so far can be considered as a variation of a search tree. Each block of entries in the multilevel index is a node. Such a node can have as many as fo pointers and fo key values, where fo is the index fan-out.

The index field values in each node guide us to the next node (i.e. a block at the next level) until we reach the data file block that contains the required record(s). By following a pointer, we restrict our search at each level to a sub-tree of the search tree and can ignore all other nodes that are not in this sub-tree.

Definition of a search tree

A search tree of order p is a tree such that each node contains at most p – 1 search values and p pointers in the order <P1, K1, P2, K2, ……, Pq-1, Kq-1, Pq>, where p and q are integers and q # p.

Each Pi is a pointer to a child node (or a null pointer in the case of a leaf node), and each Ki is a search value from some ordered set of values. All search values are assumed unique. The figure below depicts in general a node of a search tree:



Two constraints must hold at all times on the search tree:

- Within each node (internal or leaf), K1 < K2 < …… < Kq-1.
- For all values X in the sub-tree pointed at by Pi, we must have Ki-1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki-1 < X for i = q.

Whenever we search for a value X, we follow the appropriate pointer Pi according to the formulae in the second condition above.

A search tree can be used as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the search field (same as the indexing field if a multilevel index guides the search). Each value in the tree is associated with a pointer to the record in the data file with that value. Alternatively, the pointer could point to the disk block containing that record.

The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted, we must update the search tree by including in the tree the search field value of the new record and a pointer to the new record (or the block containing the record).

Algorithms are essential for inserting and deleting search values into and from the search tree while maintaining the two constraints. In general, these algorithms do not guarantee that a search tree is balanced (balanced means that all of the leaf nodes are at the same level). Keeping a search tree balanced is important because it guarantees that no nodes will be at very high levels and

hence require many block accesses during a tree search. Another problem with unbalanced search trees is that record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels.

Please note that you will not be expected to know the details of given algorithms in this chapter, as the assessment will focus on the advantages and disadvantages of indexing for performance tuning. The algorithms are only provided for the interested reader.

Review question 7

1. Describe the tree data structure.
2. What is meant by tree level and what is a sub-tree?

B-trees: Balanced trees

A B-tree is a search tree with some additional constraints on it. The additional constraints ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive.

For a B-tree, however, the algorithms for insertion and deletion become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated when we want to insert a value into a node that is already full or delete a value from a node which is just half full. The reason is that not only must a B-tree be balanced, but also, a node in the B-tree (except the root) cannot have too many (a maximum limit) or too few (half the maximum) search values.

Definition of a B-tree

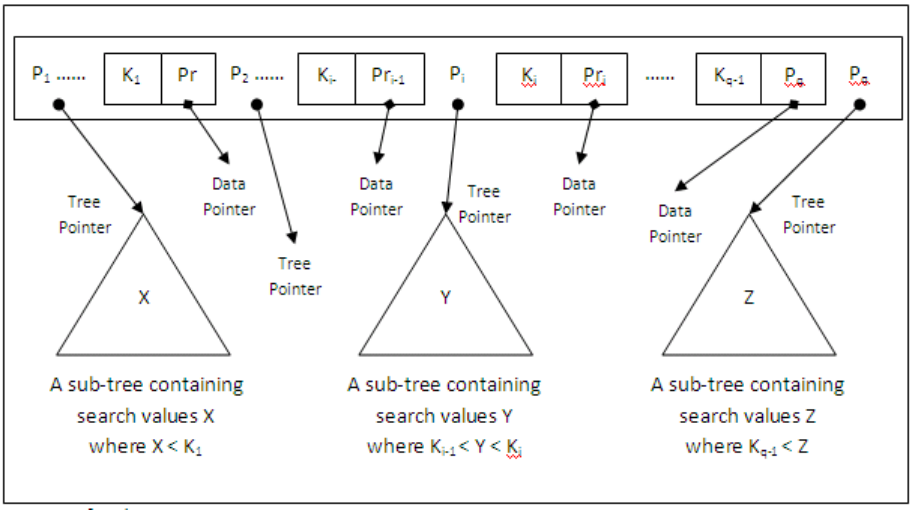
A B-tree of order p is a tree with the following constraints:

- Each internal node is of the form <P1, <K1, Pr1>, P2, <K2, Pr2>, ……, Pq-1, <Kq-1, Prq-1>, Pq> where q <= p. Each Pi is a tree pointer – a pointer to another node in the B-tree. Each Pri is a data pointer – a pointer to the block containing the record whose search field value is equal to Ki.
- Within each node, K1 < K2 < …… < Kq-1.
- For all search key field values X in the sub-tree pointed at by Pi, we have Ki-1 < X < Ki for 1 < i < q; X < Ki for i = 1; Ki-1 < X for i = q.
- Each node has at most p tree pointers and p –1 search key values (p is the order of the tree and is the maximum limit).
- Each node, except the root and leaf nodes, has at least #(p/2)# tree pointers and #(p/2)# - 1 search key values (i.e. must not be less than

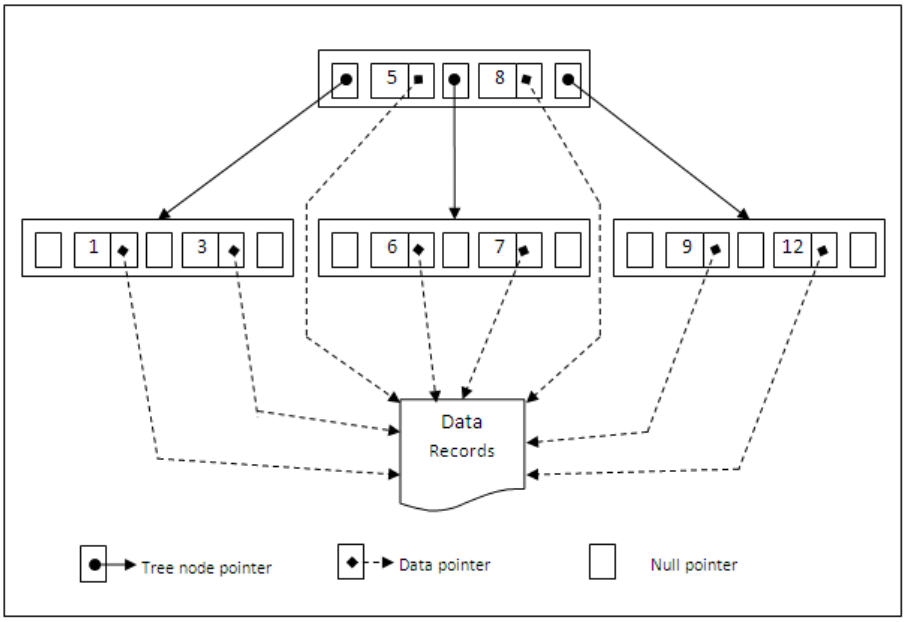
half full). The root node has at least two tree pointers (one search key value) unless it is the only node in the tree.

- A node with q tree pointers, q # p, has q – 1 search key field values (and hence has q – 1 data pointers).
- All leaf nodes are at the same level. They have the same structure as internal nodes except that all of their tree pointers Pi are null.

The figure below illustrates the general structure of a node in a B-tree of order p:



The figure below shows a B-tree of order 3:



Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use B-tree on a non-key field, we must change the definition of the data pointers Pri to point to a block (or a linked list of blocks) that contain pointers to the file records themselves. This extra level of indirection is similar to Option 3 discussed before for secondary indexes.

Performance issues

A B-tree starts with a single root node at level 0. Once the root node is full with p – 1 search key values, an insertion will cause an overflow and the node has to be split to create two additional nodes at level 1 (i.e. a new level is created). Only the middle key value is kept in the root node, and the rest of the key values are redistributed evenly between the two new nodes.

When a non-root node is full and a new entry is inserted into it, that node is split to become two new nodes at the same level, and the middle key value is moved to the parent node along with two tree pointers to the split nodes. If such a move causes an overflow in the parent node, the parent node is also split (in the same way). Such splitting can propagate all the way to the root node, creating a new level every time the root is split. We will study the algorithms in more details when we discuss B+ trees later on.

Deletion may cause an underflow problem, where a node becomes less than half full. When this happens, the underflow node may obtain some extra values from

its adjacent node by redistribution, or may be merged with one of its adjacent nodes if there are not enough values to redistribute. This can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels.

It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69% full when the number of values in the tree stabilises. This is also true of B+ trees. If this happens, node splitting and merging will occur rarely, so insertion and deletion become quite efficient. If the number of values increases, the tree will also grow without any serious problem.

The following two examples show us how to calculate the order p of a B-tree stored on disk (Example 4), and how to calculate the number of blocks and levels for the B-tree (Example 5).

Example 4

Suppose the search key field is V = 9 bytes long, the disk block size is B = 512 bytes, a data pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes. Each B-tree node can have at most p tree pointers, p – 1 search key field values, and p – 1 data pointers (corresponding to the key field values). These must fit into a single disk block if each B-tree node is to correspond to a disk block. Thus, we must have

(p * P) + ((p – 1) * (Pr + V)) # B

That is (p * 6) + ((p – 1) * (7 + 9)) # 512

We have (22 * p) # 528

We can select p to be the largest value that satisfies the above inequality, which gives p = 24.

In practice, the value of p will normally be smaller than 24 (e.g., p = 23). The reason is that, in general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries q in the node and possibly a pointer to the parent node. Thus, we should reduce the block size by the amount of space needed for all such information before we determine the value of p.

Example 5

Suppose that the search field of Example 4 is a non-ordering key field, and we construct a B-tree on this field. Assume that each node of the tree is 69% full. On average, therefore, each node will have p * 0.69 = 23 * 0.69 @ 16 pointers, and hence 15 search key field values. The average fan-out fo = 16.

We can now start from the root and see how many values and pointers may exist, on average, at each subsequent level (see the table below):

Level	Number of Nodes	Number of Entries (Search Key Values)	NumberofTree Pointers
Level 0 (Root)	1	15	16
Level 1	16	240	256
Level 2	256	3840	4096
Level 3	4096	61,440	

At each level, we calculate the number of entries by multiplying the total number of pointers at the previous level by 15 – the average number of entries in each node. Hence, for the given block size, pointer size, and search key field size (as in Example 5), a two level B-tree can hold up to 3840 + 240 + 15 = 4095 entries; a three-level B-tree can hold up to 61440 + 3840 + 240 + 15 = 65,535 entries on average.

B+ trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure, called a B+ tree. In a B-tree, every value of the search field appears once at some level in the tree, along with a corresponding data pointer. In a B+ tree, data pointers are stored only at the leaf nodes of the tree; hence, the structure of leaf nodes differs from that of internal nodes.

The leaf nodes have an entry for every value of the search field, along with a corresponding data pointer to the block containing the record with that value if the search field is a key field. For a non-key search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection (as we have seen before).

The leaf nodes of a B+ tree are usually linked together to provide ordered access on the search field to the records. These leaf nodes are similar to the first level of a multilevel index. Internal nodes correspond to the other levels of the multilevel index. Some search field values from the leaf nodes are duplicated in the internal nodes to guide the search.

Definition of a B+ tree

The structure of the internal nodes of a B+-tree of order p is defined as follows:

- Each internal node is of the form: <P1, K1, P2, K2, ……, Pq-1, Kq-1, Pq> where q <= p and each Pi is a tree pointer.
- Within each internal node, K1 < K2 < …… < Kq-1.
- For all search field values X in the sub-tree pointed at by Pi, we have

  Ki-1 < X <= Ki for 1 < i < q;

X <= Ki for i = 1;

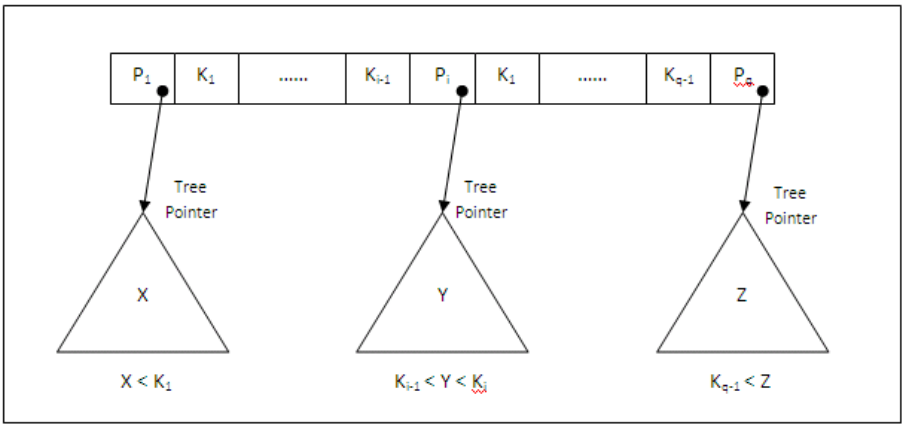
Ki-1 < X for i = q.

- Each internal node has at most p tree pointers and p –1 search key values.
- Each internal node, except the root, has at least #(p/2)# tree pointers and #p/2)# - 1 search key values (i.e. must not be less than half full). The root node has at least two tree pointers (one search key value) if it is an internal node.
- An internal node with q tree pointers, q # p, has q – 1 search key field values.

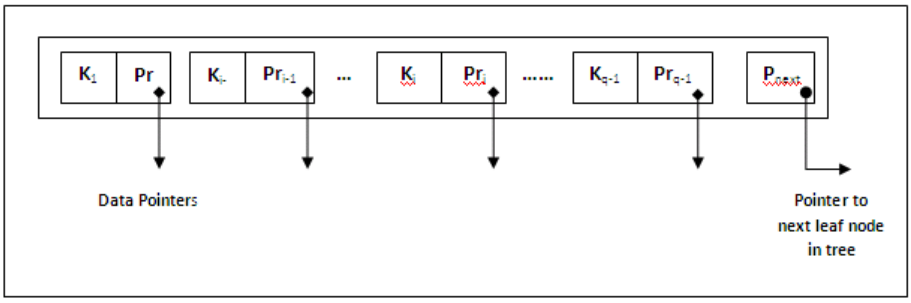
The structure of the leaf nodes of a B+ tree of order p is defined as follows:

- Each leaf node is of the form <<K1, Pr1>, <K2, Pr2>, ……, <Kq-1, Prq-1>, Pnext> where q # p. Each Pri is a data pointer, and Pnext points to the next leaf node of the B+ tree.
- Within each internal node, K1 < K2 < …… < Kq-1.
- Each Pri is a data pointer pointing to the record whose search field value is Ki, or to a file block containing the record (or to a block of record pointers that point to records whose search field value is Ki, if the search field is not a key).
- Each leaf node has at least #(p/2)# values (entries).
- All leaf nodes are at the same level.

The following figure depicts the general structure of an internal node of a B+ tree of order p:



And the following figure depicts the general structure of a leaf node of a B+ tree of order p:



Performance issues

Because entries in the internal nodes just contain search values and tree pointers without any data pointers, more entries can be accommodated into an internal node of a B+ tree than for a similar B-tree. Thus, for the same block (node) size, the order p can be larger for the B+ tree than for a B-tree (see Example 6). This can lead to fewer B+ tree levels, and therefore improve search time.

Because structures of internal nodes and leaf nodes are different, their orders can be different. Generally, the order p is for the internal nodes, and leaf nodes can have a different order denoted as pleaf, which is defined as being the maximum number of data pointers in a leaf node.

Example 6

To calculate the order p of a B+tree, suppose that the search key field is V = 9 bytes long, the block size is B = 512 bytes, a record (data) pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes, all as in Example 4. An internal node of such a B+-tree can have up to p tree pointers and p – 1 search field values: these must fit into a single block. Hence, we have

(p * P) + ((p – 1) * V) # B

That is (p * 6) + ((p – 1) * 9) # 512

We have (15 * p) # 521

We can select p to be the largest value that satisfies the above inequality, which gives p = 34. This is larger than the value of 23 for the B-tree in Example 5, resulting in a larger fan-out and more entries in each of the internal nodes.

The leaf nodes of the B+tree will have the same number of values and pointers, except that the pointers are data pointers and one Pnext pointer (i.e., a block pointer to another leaf node). Hence, the order pleaf for the leaf nodes can be calculated similarly:

(pleaf * (Pr + V)) + P # B

That is (pleaf * (7 + 9)) + 6 # 512

We have (16 * pleaf) # 506

It follows that each leaf node can hold up to pleaf = 31 (key value, data pointer) pairs.

As with B-tree, in practice, the value of p and pleaf will normally be smaller than the above values. The reason is that, in general, we may need additional information in each node to implement the insertion and deletion algorithms. This kind of information can include the type of nodes (i.e., internal or leaf), the number of current entries q in the node, and pointers to the parent and sibling nodes (e.g., Pprevious). Thus, we should reduce the block size by the amount of space needed for all such information before we determine the values of p and pleaf.

The next example explains how we can calculate the number of entries in a B+tree.

Example 7

Suppose that we want to build a B+tree on the field of Example 6. To calculate the approximate number of entries of the B+-tree, we assume that each node is 69% full. On average, each internal node will have 0.69 * p = 0.69 * 34 = 23 pointers and hence 22 values. Each leaf node, on average, will hold 0.69 * pleaf = 0.69 * 31 = 21 data record pointers. Such a B+-tree will have the following average number of entries at each level:

Level	Number of Nodes	Number of Entries (Search Key Values)	NumberofTree Pointers
Level 0 (Root)	1	22	23

Level	Number of Nodes	Number of Entries (Search Key Values)	NumberofTree Pointers
Level 1	23	506	529
Level 2	529	11,638	12,167
Leaf Level	12,167	255,507 (Same number of record pointers)	

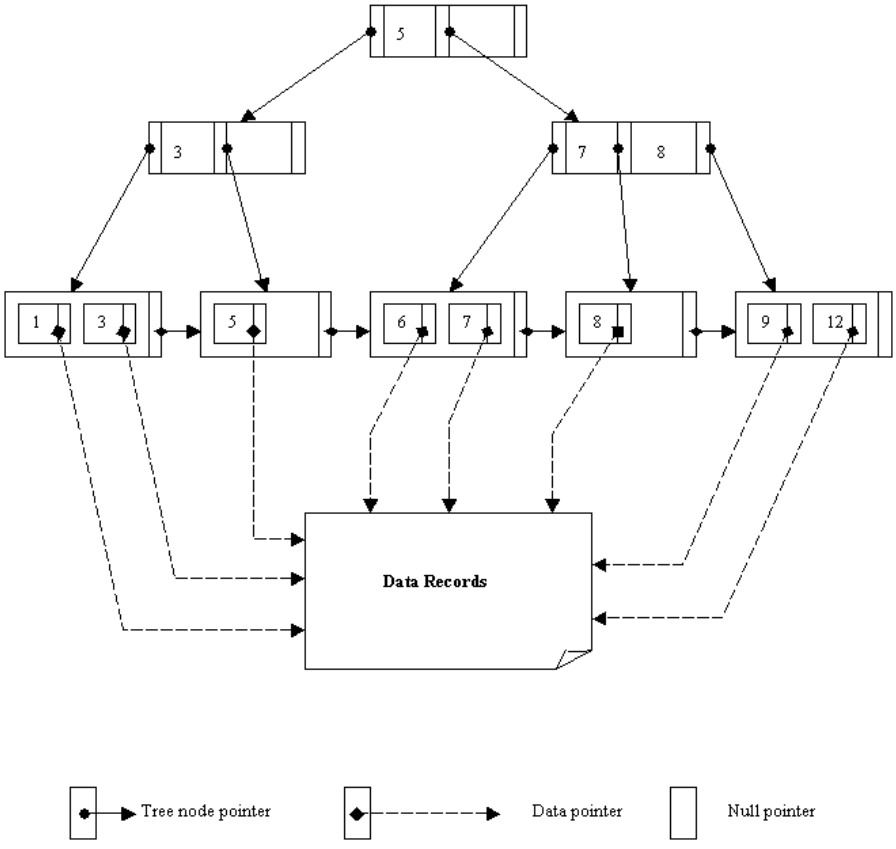
For the block size, pointer size, and search field size provided above, a three-level B+tree holds up to 255,507 record pointers, on average. Compare this to the 65,535 entries for the corresponding B-tree in Example 6. It is apparent that a B+tree is more efficient.

Search, insertion and deletion with B+ trees

To search for a record based on a specified search field value K:

- We always start from the root of the B+ tree. Compare K with the values stored in the root until we find the first Ki where K <= Ki. Then follow the tree pointer Pi to the next level. If no such Ki is found, then follow the rightmost pointer Pq to the next level. (We assume that the root is not the only node in the tree. Otherwise follow the search procedure for a leaf node.)
- If the node at the next level is still an internal node, we repeat the above procedure, until we reach the leaf level.
- Within a leaf node, we search for a stored value Ki that equals K. If found, we follow its data pointer Pri to retrieve the data block. Otherwise, the desired record is not in the file.

The following is an example of a B+ tree with p = 3 and pleaf = 2:



Refer to the tree above. Suppose we want to retrieve the record with value 6

in the search field. Having searched the root, we could not find a value Ki such that 6 # Ki, because there is only one value 5 in the node. Thus, we follow the rightmost pointer to the internal node at the next level. This node has two values 7 and 8, but 7 is the first value that is bigger than 6. Hence, we follow the pointer left to the value 7 to the leaf level. By searching the values in the leaf node, we find the value 6. Then we will use the related data pointer to go to the data file and retrieve the record.

Note that every distinct search value must exist at the leaf level, because all data pointers are at the leaf level. However, only some of the search values exist in internal nodes, to guide the search to reach the appropriate leaf node. Also note that every value appearing in an internal node also appears as the rightmost value in the sub-tree pointed at by the tree pointer to the left of the value. In the above, for example, the value 5 in the root node (an internal node as well) is also stored in the sub-tree pointed at by the left pointer. The value 5 is in the leaf node of the sub-tree and is the rightmost value. The same is true for other values in the internal nodes, such as 3, 7 and 8.

Insertion and deletion of entries in a B+ tree can cause the same overflow and underflow problems as for a B-tree, because of the restrictions (constraints) imposed by the B+ tree definition. These problems are dealt with in a similar way as in a B-tree, i.e. by splitting or merging nodes concerned.

Dealing with overflow

- When a leaf node overflows because of insertion of a new entry, it must be split. The first m = #((pleaf + 1)/2)# entries in the original node are kept there, and the rest are moved to a new leaf node (just created).
- The original pnext pointer is changed and points to the new leaf node, and the same pointer new_pnext in the new node will have the original value of pnext.
- The mth search value is duplicated in the parent node (an internal node), and an extra pointer to the new node is created in the parent.
- If the above move of value causes overflow in the parent node because the parent node is already full, then it must also split.
- The entries in the internal node (now overflowed) up to Pn – the nth tree pointer after inserting the new value and pointer, where n = #((P + 1)/2)#
  - are kept, while the nth search value is moved to the parent, not duplicated.
- A new internal node will now hold the entries from Pn+1 to the end of the entries in the overflowed node.

- This splitting can propagate all the way up to create a new root node and hence a new level for the B+ tree.

Dealing with underflow

- When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node (because it is the rightmost value of a sub-tree), then it must also be removed from there. In this case, the value to its left in the leaf node will replace it in the internal node, because that node is now the rightmost entry in the sub-tree.
- If deletion causes underflow, we will try to find an adjacent leaf node and redistribute the entries among the two nodes so that both can become at least half full.
- If we cannot find such an adjacent node with enough entries, then we will merge them. In this case, the number of leaf nodes is reduced.
- If a merge between two nodes is not possible because of insufficient number of entries, we may merge three leaf nodes into two.
- In merge cases, underflow may propagate to internal nodes because one fewer tree pointer and search value are needed. This can propagate to the root and reduce the tree levels.

The algorithms for maintaining a B+ tree structure are necessarily complex, and therefore some operational details are omitted in this chapter. However, such algorithms are well established and tested, and their efficiency proved. Interested students are advised to read the relevant chapters in the recommended textbooks for details.

B* tree: A variation of B-tree and B+ tree

Recall that the definitions of B-tree and B+ tree require each node to be at least half full. Such a requirement can be changed to require each node to be at least two-thirds full. In this case, the B-tree is called a B*tree. This can improve the retrieval performance a little further without incurring too many overheads on maintaining the tree.

Exercise 7

Consider the disk with block size B = 512 bytes. A block pointer is P = 8 bytes long, and a record pointer is Pr = 9 bytes long. A file has r = 50,000 STUDENT records of fixed-size R = 147 bytes. The key field is ID# whose length is V = 12 bytes. (This is the same disk file as in previous exercises. Some of the early results should be utilised.) Suppose that the file is NOT sorted by the key field ID# and we want to construct a B-tree access structure (index) on ID#. Answer the following questions:

1. What is an appropriate order p of this B-tree?
2. How many levels are there in the B-tree if nodes are approximately 69% full?
3. What is the total number of blocks needed by the B-tree if they are approximately 69% full?
4. How many block accesses are required to search for and retrieve a record from the data file, given an ID#, using the B-tree?

Exercise 8

For the same disk file as in Exercise 7, suppose that the file is NOT sorted by the key field ID# and we want to construct a B+tree access structure (index) on ID#. Answer the following questions:

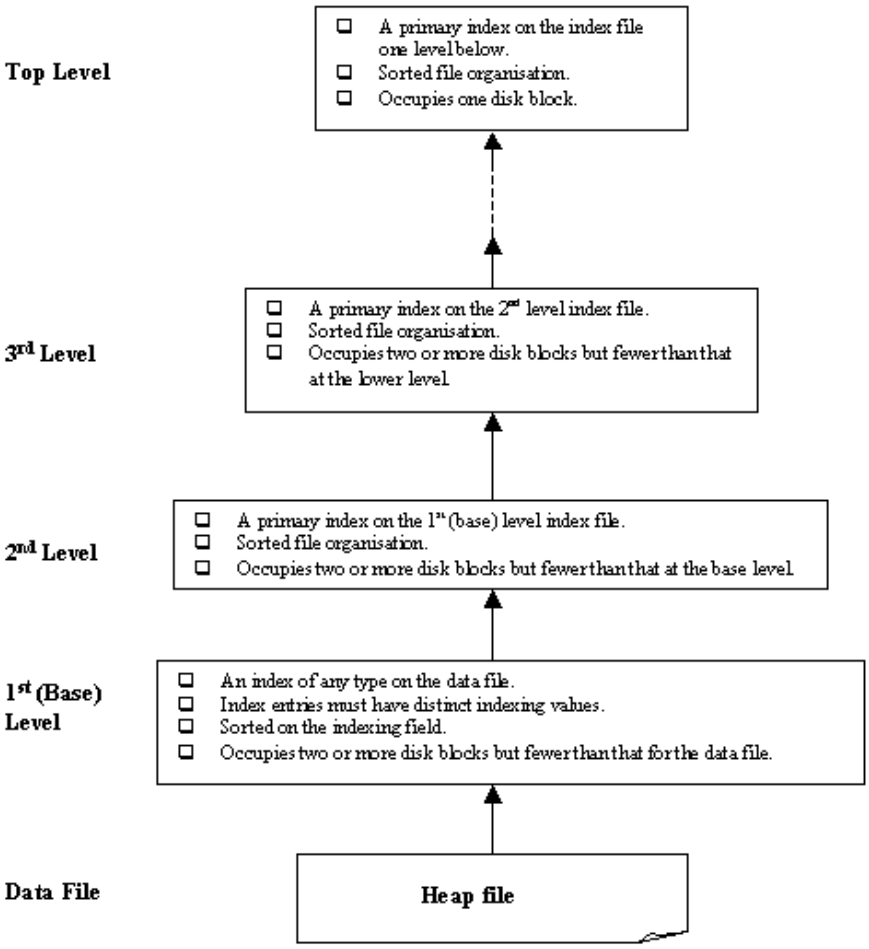
1. What are the appropriate order p (for internal nodes) and pleaf (for leaf nodes) of this B+-tree?
2. How many leaf-level blocks are needed if blocks are approximately 69% full?
3. How many levels are there if internal nodes are also 69% full?
4. How many block accesses are required to search for and retrieve a record from the data file, given an ID#, using the B+tree?

Summary

B-trees/B+ trees/B* trees are data structures which can be used to implement dynamic multilevel indexes very effectively. When we were discussing multilevel indexes, we emphasised that the multilevel structure can be constructed for any type of index, whether it is a primary, a clustering or a secondary index, as long as the first-level index has distinct values for K(i) and fixed-length.

As a logical consequence of the above, the multilevel index structure does not require the underlying data file to have any specific file organisation, i.e. it can be a heap file, a sorted file or some other organisations.

Also recall that a heap file organisation is very efficient with regard to insertion and deletion, but less efficient for retrieval. Such inefficiency for retrieval can be overcome by building a multilevel index structure on top of it. A heap file with a multilevel index is, therefore, a very effective combination that takes full advantage of different techniques while overcoming their respective shortcomings. The figure below illustrates the combination:



Review question 8

1. Describe the general structure of a B-tree node. What is the order p of a B-tree?
2. Describe the structure of both internal and leaf nodes of a B+ tree. What is the order p of a B+ tree?
3. Why is a B+ tree usually preferred as an access structure to a data file rather than a B-tree?
4. In a B+ tree structure, how many block accesses do we need before a record is located? (Assuming the record is indeed in the data file and its indexing field value is specified.)

5. What major problems may be caused by update operations on a B-tree/B+ tree structure and why?

Chapter 12. Database Security

Table of contents

Objectives

At the end of this chapter you should be able to:

- Understand and explain the place of database security in the context of security analysis and management.
- Understand, explain and apply the security concepts relevant to database systems.
- Understand, identify and find solutions to security problems in database systems.
- Understand the basic language of security mechanisms as applied to database systems.
- Analyse access control requirements and perform fairly simple implementations using SQL.
- Appreciate the limitations of security subsystems.

Introduction

In parallel with this chapter, you should read Chapter 19 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

Security is a large subject and one that, because it touches every activity of an information system, appears everywhere. In the main text you will start with a thumbnail introduction to security, while the extension reading contains references for you to pursue when you wish.

In earlier chapters in this module you have met concepts and techniques which can be regarded as security measures. For example, the process of recovery, whether from partial or total failure, should be considered as having a security dimension. Nearly all the work on concurrency (see chapter 13) is directed at another aspect of security. Again, a thumbnail introduction is given.

The main work you do in this chapter, however, is directed to database security rather than security in general, and to the principles of security theory and practice as they relate to database security. These are technical aspects of security rather than the big picture.

The chapter is organised into two parts. The first part covers security principles and models itself in two parts moving from the softer principles (setting the universe of discourse) through to some specific technical issues of database security. The second part is about logical access control in SQL databases.

The major practical area you will cover is the area of access control. After a discussion of the principles, you will quickly be getting into some detail of access

control in SQL databases. Extension reading, both textbooks and websites, is given for you to pursue the detail further.

What is not covered in this chapter, but is covered elsewhere in the module, are the subjects of database administration, transaction recovery and catastrophe recovery. The first of these is directly related to management controls on operation and development. The second is directly related to database integrity and consistency, thus being largely an internal matter. The third is easier to follow as an extension of the first and second. But all three are security based.

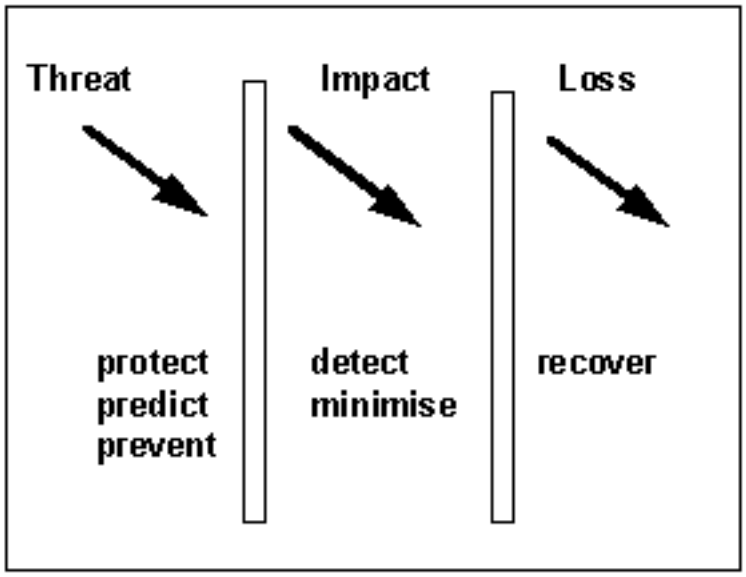
The scope of database security

Overview

All systems have ASSETS and security is about protecting assets. The first thing, then, is to know your assets and their value. In this chapter, concentrate on database objects (tables, views, rows), access to them, and the overall system that manages them. Note that not all data is sensitive, so not all requires great effort at protection. All assets are under threat.

The second thing to know is what THREATs are putting your assets at risk. These include things such as power failure and employee fraud. Note that threats are partly hypothetical, always changing and always imperfectly known. Security activity is directed at protecting the system from perceived threats.

If a threat is potential, you must allow for it to become an actuality. When it becomes actual there is an IMPACT. Impact you can consider and plan for. But in the worst case, there will be a LOSS. Security activity here is directed at minimising the loss and recovering the database to minimise the loss as well as further protecting from the same or similar threats.



An outlined development mechanism is:

1. Document assets (what they are, what their value is).
2. Identify treats (what they are, how likely they are, what the impact is if they occur).
3. Associate threats with each asset.
4. Design mechanisms to protect each asset appropriate to its value and the cost of its protection, to detect a security breach against each asset, to minimise the losses incurred and to recover normal operation.

Threats to the database

You will build your security skills from two directions. One is from the appreciation and awareness of changing threats, and the other from the technical remedies to them. Threats include:

- Unauthorised modification: Changing data values for reasons of sabotage, crime or ignorance which may be enabled by inadequate security mechanisms, or sharing of passwords or password guessing, for example.
- Unauthorised disclosure: When information that should not have been disclosed has been disclosed. A general issue of crucial importance, which can be accidental or deliberate.

- Loss of availability: Sometimes called denial of service. When the database is not available it incurs a loss (otherwise life is better without the system!). So any threat that gives rise to time offline, even to check whether something has occurred, is to be avoided.

The rest of this section is an overview of the categories of specific regulatory threats to database systems.

- **Commercial sensitivity:** Most financial losses through fraud arise from employees. Access controls provide both protection against criminal acts and evidence of attempts (successful or otherwise) to carry out acts detrimental to the organisation, whether fraud, extraction of sensitive data or loss of availability.
- **Personal privacy and data protection:** Internationally, personal data is normally subject to legislative controls. Personal data is data about an identifiable individual. Often the individual has to be alive but the method of identification is not prescribed. So a postal code for a home may in some cases identify an individual, if only one person is living at an address with the postal code. Such data needs careful handling and control.

  For more information see Data Protection later in the chapter. The issues are too extensive to be discussed here but the implications should be noted. Personal data needs to be identified as such. Controls must exist on the use of that data (which may restrict ad-hoc queries). Audit trails of all access and disclosure of the information need to be retained as evidence.
- **Computer misuse:** There is also generally legislation on the misuse of computers. Misuse includes the violation of access controls and attempts to cause damage by changing the database state or introducing worms and viruses to interfere with proper operation. These offences are often extraditable. So an unauthorised access in Hong Kong using computers in France to access databases in Germany which refer to databases in America could lead to extradition to France or Germany or the USA.
- **Audit requirements:** These are operational constraints built around the need to know who did what, who tried to do what, and where and when everything happened. They involve the detection of events (including CONNECT and GRANT transactions), providing evidence for detection, assurance as well as either defence or prosecution. There are issues related to computer-generated evidence not covered here.

In considerations of logical access to the database, it is easy to lose sight of the fact that all system access imposes risks. If there is access to operating system utilities, it becomes possible to access the disk storage directly and copy or damage the whole database or its components. A full consideration has to take all such access into account. Most analysts would be looking to minimise communications (direct, network and telecommunications) and isolate

the system from unnecessary threats. It is also likely that encryption would be used both on the data and the schema. Encryption is the process of converting text and data into a form that can only be read by the recipient of that data or text, who has to know how to convert it back to a clear message.

You will find it easier to consider security and auditing as issues separate from the main database functions, however they are implemented. Visualise the security server and audit servers as separate functional modules.

Principles of database security

To structure thoughts on security, you need a model of security. These come in various forms that depend on roles, degree of detail and purpose. The major categories are areas of interest (threats, impact and loss) as well as the actions involved in dealing with them.

Security risks are to be seen in terms of the loss of assets. These assets include:

- Hardware
- Software
- Data
- Data quality
- Credibility
- Availability
- Business benefit

Here we are primarily concerned with threats to the data and data quality but, of course, a threat to one asset has consequential impact on other assets. What is always important is that you are very clear on just what asset needs protection.

So as a summary:

Problem	Tool	Technique
Reliability (operational security)	Recover from corruption, loss and damage	Back-up, logging, checkpoints.
Access Security	Control Access	Passwords, dialogues.
Integrity (schema security)	Ensure internal consistency	Validation rules, constraints.

You need to accept that security can never be perfect. There always remains an element of risk, so arrangements must be made to deal with the worst eventuality - which means steps to minimise impact and recover effectively from loss or damage to assets. Points to bear in mind:

1. Appropriate security - you do not want to spend more on security than the asset is worth.
2. You do not want security measures to interfere unnecessarily with the proper functioning of the system.

Security models

A security model establishes the external criteria for the examination of security issues in general, and provides the context for database considerations, including implementation and operation. Specific DBMSs have their own security models which are highly important in systems design and operation. Refer to the SeaView model for an example.

You will realise that security models explain the features available in the DBMS which need to be used to develop and operate the actual security systems. They embody concepts, implement policies and provide servers for such functions. Any faults in the security model will translate either into insecure operation or clumsy systems.

Access control

The purpose of access control must always be clear. Access control is expensive in terms of analysis, design and operational costs. It is applied to known situations, to known standards, to achieve known purposes. Do not apply controls without all the above knowledge. Control always has to be appropriate to the situation. The main issues are introduced below.

Authentication and authorisation

We are all familiar as users with the log-in requirement of most systems. Access to IT resources generally requires a log-in process that is trusted to be secure. This topic is about access to database management systems, and is an overview of the process from the DBA perspective. Most of what follows is directly about Relational client-server systems. Other system models differ to a greater or lesser extent, though the underlying principles remain true.

For a simple schematic, see Authorisation and Authentication Schematic.

Among the main principles for database systems are authentication and authorisation.

Authentication

The client has to establish the identity of the server and the server has to establish the identity of the client. This is done often by means of shared secrets (either a password/user-id combination, or shared biographic and/or biometric data). It can also be achieved by a system of higher authority which has previously established authentication. In client-server systems where data (not necessarily the database) is distributed, the authentication may be acceptable from a peer system. Note that authentication may be transmissible from system to system.

The result, as far as the DBMS is concerned, is an authorisation-identifier. Authentication does not give any privileges for particular tasks. It only establishes that the DBMS trusts that the user is who he/she claimed to be and that the user trusts that the DBMS is also the intended system.

Authentication is a prerequisite for authorisation.

Authorisation

Authorisation relates to the permissions granted to an authorised user to carry out particular transactions, and hence to change the state of the database (write-item transactions) and/or receive data from the database (read-item transactions). The result of authorisation, which needs to be on a transactional basis, is a vector: Authorisation (item, auth-id, operation). A vector is a sequence of data values at a known location in the system.

How this is put into effect is down to the DBMS functionality. At a logical level, the system structure needs an authorisation server, which needs to co-operate with an auditing server. There is an issue of server-to-server security and a problem with amplification as the authorisation is transmitted from system to system. Amplification here means that the security issues become larger as a larger number of DBMS servers are involved in the transaction.

Audit requirements are frequently implemented poorly. To be safe, you need to log all accesses and log all authorisation details with transaction identifiers. There is a need to audit regularly and maintain an audit trail, often for a long period.

Access philosophies and management

Discretionary control is where specific privileges are assigned on the basis of specific assets, which authorised users are allowed to use in a particular way. The security DBMS has to construct an access matrix including objects like relations, records, views and operations for each user - each entry separating create, read, insert and update privileges. This matrix becomes very intricate as authorisations will vary from object to object. The matrix can also become very large, hence its implementation frequently requires the kinds of physical

implementation associated with sparse matrices. It may not be possible to store the matrix in the computer's main memory.

At its simplest, the matrix can be viewed as a two-dimensional table:

Database User	Database Object	**Access Rights**
Pay-clerk1	Pay-table-view1	Read, Update
Pay-clerk2	Pay-table	Read, Insert
Pay-auditor	Pay-table	Read

When you read a little more on this subject, you will find several other rights that also need to be recorded, notably the owners' rights and the grant right.

Mandatory control is authorisation by level or role. A typical mandatory scheme is the four-level government classification of open, secret, most secret and top secret. The related concept is to apply security controls not to individuals but to roles - so the pay clerk has privileges because of the job role and not because of personal factors.

The database implication is that each data item is assigned a classification for read, create, update and delete (or a subset of these), with a similar classification attached to each authorised user. An algorithm will allow access to objects on the basis of less than or equal to the assigned level of clearance - so a user with clearance level 3 to read items will also have access to items of level 0, 1 and 2. In principle, a much simpler scheme.

The Bell-LaPadula model (2005) defines a mandatory scheme which is widely quoted:

- A subject (whether user, account or program) is forbidden to read an object (relation, tuple or view) unless the security classification of the subject is greater or equal to that of the object.
- A subject is forbidden to write an object unless the security classification of the subject is less than or equal to that of the object.

Note that a high level of clearance to read implies a low level of clearance to write - otherwise information flows from high to low levels. This is, in highly secure systems, not permitted.

Mandatory security schemes are relatively easy to understand and, therefore, relatively easy to manage and audit. Discretionary security is difficult to control and therefore mistakes and oversights are easy to make and difficult to detect. You can translate this difficulty into costs.

There are perhaps two other key principles in security. One is disclosure, which is often only on a need-to-know basis. This fits in better with discretionary

security than mandatory, as it implies neither any prior existing level nor the need for specific assignment.

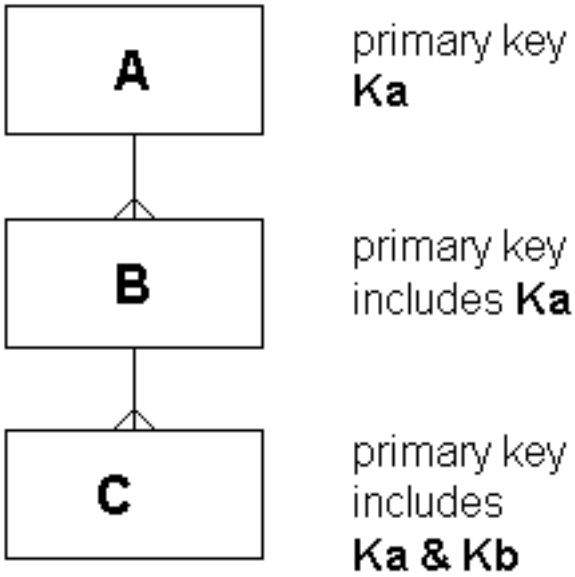
The other principle is to divide responsibilities. The DBA responsible for security management is him/herself a security risk. Management approaches that involve one person or a group of people that have connections in their work represent a similar risk. This emphasises the importance of security auditing and the importance of related procedure design.

Database security issues

This section reviews some of the issues that arise in determining the security specification and implementation of a database system.

Access to key fields

Suppose you have a user role with access rights to table A and to table C but not to table B. The problem is that the foreign key in C includes columns from B. The following questions arise:



Do you have access to the foreign key in C?

If you do, you know at least that a tuple exists in B and you know some information about B that is restricted from you.

Can you update the foreign key columns?

If so, it must cascade, generating an update to B for which no privileges have been given.

These problems do not directly arise where the database is implemented by internal pointers - as a user, you need have no knowledge of the relationships between the data you are accessing. They arise because relationships are data values. Often, knowing the foreign key will not be sensitive in itself. If it is, then the definition of a view may solve the problem.

Access to surrogate information

It is not difficult to conceive of cases where the view of the data provided to a user role extends to the external world.

An example should make the problem clear.

In a retail environment, there are frequent problems with pilferage. To deal with these, private detectives work undercover. They are to all intents and purposes employees of the business and assigned to normal business activities as other members of staff. They get pay checks or slips at the same time as everyone else, they appear in management information (such as the salary analysis) in the same manner. They have a job title and participate in the system as someone they are not. The store manager is unaware of the situation, as is everybody else except the corporate security manager. When the store manager accesses the database, the detective should look like a normal employee. Queries might include:

"What leave is due to …?"

The security staff have different queries:

"Do we have someone in …?"

You can probably envisage all kinds of complications. The detective should receive a pay slip with everyone else, but should not actually be paid (or perhaps he/she should be paid something different from the normal pay for the job).

You may want to handle these situations on separate databases. As a solution it may be appropriate, but the larger the problem the more scope there is for confusion. One suggested solution is the polyinstantiation of tuples - one individual is represented by more than one tuple. The data retrieved will depend on the security classification of the user. Tuples will have the same apparent primary key but different actual primary keys, and all applications will need to be carefully integrated with the security system.

Problems with data extraction

Where data access is visualised directly, the problem can be seen clearly enough: it is to ensure that authenticated users can access only data items which they are authorised to use for the purpose required. When the focus shifts from the data to the implications that can be drawn from that data, more problems arise.

Again, an example should make things clear.

You want to know the pay of the chief executive. You have access rights to the table, except for the MONTHLY-PAY field in this tuple. So you issue an SQL query SUM (MONTHLY-PAY) across the whole table. You then create a view SELECT MONTHLY-PAY … and issue a SUM on this view. Should you get the same answer in both cases?

If not, you can achieve your objective by subtracting the two sums. If you listed the monthly pay for all, what would you expect to see - all the tuples except the one restricted? Would you expect to be notified by asterisks that data was missing which you were not allowed to see?

Another example.

You are trying to trace an individual but have limited information. You feed your limited information into the statistical database (e.g. male, age over 40, white, red car, lives in North London) and retrieve the tuples for all that meet these categories. As you get more information, the number of tuples reduces until only one is left. It is possible to deduce personal information from a statistical database if you have a little information about the structure, even if no conventional personal identifiers are available (i.e. no date of birth, social security number or name).

Some solutions to this security problem are to prevent access to small numbers of tuples and/or to produce inaccurate data (not very inaccurate but sufficiently inaccurate to prevent inferences being drawn).

Access control in SQL

This section is about the implementation of security within SQL. The basics are given in SQL-92 but, as you will realise, much security is DBMS- and hardware-specific. Where necessary, any specifics are given in the SQL of Oracle. For some ideas on Object database management systems (ODBMS) as distinct from Relational, refer to the later chapter on Object databases.

Your first objective is to learn the specifics. The access requirements specification will be implemented using these statements. Your second objective is to extend your understanding of the problem through to the management and audit functions of an operating system.

The basic statements come first, and the management functions are discussed second. In the first part you will learn the SQL needed to manage a user; in the second you will learn a little of the SQL to manage a system.

Discretionary security in SQL

This section introduces the SQL statements needed to implement access control. You should aim at having sufficient knowledge of this area of SQL to translate a simple specification into an SQL script. You should also be conscious of the limitations implicit in this script which hardwires passwords into text.

The basics of SQL are inherently discretionary. Privileges to use a database resource are assigned and removed individually.

The first issue is who is allowed to do what with the security subsystem. You need to have a high level of privilege to be able to apply security measures. Unfortunately, such roles are not within the SQL standard and vary from DBMS to DBMS. A role is defined as a collection of privileges.

As an example, the supplied roles in Oracle include (among others):

- **SYSOPER:** Start and stop the DBMS.
- **DBA:** Authority to create users and to manage the database and existing users.
- **SYSDBA:** All the DBA's authority plus the authority to create, start, stop and recover.

The role of the DBA has been covered in other chapters. The point here is that you realise there are a large number of predefined roles with different privileges and they need to be controlled. It is important to be certain that the SQL defaults do not act in ways you do not anticipate.

Schema level

The first security-related task is to create the schema. In the example below, the authorisation is established with the schema. The authorisation is optional and will default to the current user if it is not specified.

Only the owner of the schema is allowed to manipulate it. Below is an example where a user is given the right to create tables. The creator of the table retains privileges for the tables so created. Similarly, synonyms are only valid for the creator of that synonym.

CREATE SCHEMA student_database AUTHORISATION U1;

The U1 refers to the authorisation identifier of the user concerned, who has to have the right to create database objects of this type – in this case, the schema for a new database.

Provided the authorisation is correct, then the right to access the database using the schema can be granted to others. So to allow the creation of a table:

GRANT CREATETAB TO U1 ;

The topic of schema modifications will not be taken up here.

Authentication

Using the client/server model (see chapter 15), it is necessary first to connect to the database management system, effectively establishing both authentication and the complex layers of communication between the local (client DBMS) and the server.

GRANT CONNECT TO student_database AS U1,U2,U3 IDENTIFIED BY P1,P2,P3;

U1,U2,U3 are user names, P1,P2,P3 are passwords and student_database is the database name.

GRANT CONNECT TO student_database AS U4/P4 ;

Connect rights give no permission for any table within the database. U4/P4 are the identifiers known to this database security services.

Note

Users, roles and privilege levels can be confusing. The following are the key distinctions:

- A user is a real person (with a real password and user account).
- A role, or a user-role, is a named collection of privileges that can be easily assigned to a given or new user. A privilege is a permission to perform some act on a database object.
- A privilege level refers to the extent of those privileges, usually in connection with a database-defined role such as database administrator.

Table level

The authority level establishes some basic rights. The SYSDBA account has full rights and can change everything. Rights to access tables have to be GRANTed separately by the DBA or SYSADM.

The following example assigns a read privilege to a named table (note only a read privilege). The privilege extends to creating a read-only view on the table:

GRANT SELECT ON TABLE1 TO U1;

And that which may be given can be removed. REVOKE is used generally to remove any specific privilege.

REVOKE SELECT ON TABLE1 FROM U1;

The main part of this aspect of security, though, is providing access to the data. In a Relational database we have only one data structure to consider, so if we can control access to one table we can control access to all. And as tables are two dimensional, if we can control access to rows and columns, we can deal with any request for data – including schema data. We still have to know what is allowed and what is not but, given the details, the implementation is not in itself a problem.

Remember that a VIEW is created by an SQL SELECT, and that a view is only a virtual table. Although not part of the base tables, it is processed and appears to be maintained by the DBMS as if it were.

To provide privileges at the level of the row, the column or by values, it is necessary to grant rights to a view. This means a certain amount of effort but gives a considerable range of control. First create the view:

'the first statement creates the view'

CREATE VIEW VIEW1

AS SELECT A1, A2, A3

FROM TABLE1

WHERE A1 < 20000;

'and the privilege is now assigned'

GRANT SELECT ON VIEW1 TO U1

WITH GRANT OPTION;

The optional "with grant option" allows the user to assign privileges to other users. This might seem like a security weakness and is a loss of DBA control. On the other hand, the need for temporary privileges can be very frequent and it may be better that a user assign temporary privileges to cover for an office absence, than divulge a confidential password and user-id with a much higher level of privilege.

The rights to change data are granted separately:

GRANT INSERT ON TABLE1 TO U2, U3;

GRANT DELETE ON TABLE1 TO U2, U3;

GRANT UPDATE ON TABLE1(salary) TO U5;

GRANT INSERT, DELETE ON TABLE1 TO U2, U3;

Notice in the update, that the attributes that can be modified are specified by column name. The final form is a means of combining privileges in one expression.

To provide general access:

GRANT ALL TO PUBLIC;

SQL system tables

The DBMS will maintain tables to record all security information. An SQL database is created and managed by the use of system tables. These comprise a relational database using the same structure and access mechanism as the main database. Examples below show the kind of attributes in some of the tables involving access control. The key declarations have been removed.

The examples are from Oracle database.

Command	Description
ALL_ARGUMENTS	Arguments in object accessible to the user
ALL_CATALOG	All tables, views, synonyms, sequences accessible to the user
ALL_COL_COMMENTS	Comments on columns of accessible tables and views
ALL_CONSTRAINTS	Constraint definitions on accessible tables
ALL_CONS_COLUMNS	Information about accessible columns in constraint definitions
ALL_DB_LINKS	Database links accessible to the user
ALL_ERRORS	Current errors on stored objects that user is allowed to create
ALL_INDEXES	Descriptions of indexes on tables accessible to the user
ALL_IND_COLUMNS	COLUMNs comprising INDEXes on accessible TABLES
ALL_LOBS	Description of LOBs contained in tables accessible to the user
ALL_OBJECTS	Objects accessible to the user
ALL_OBJECT_TABLES	Description of all object tables accessible to the user
ALL_SEQUENCES	Description of SEQUENCEs accessible to the user
ALL_SNAPSHOTS	Snapshots the user can access
ALL_SOURCE	Current source on stored objects that user is allowed to create
ALL_SYNONYMS	All synonyms accessible to the user
ALL_TABLES	Description of relational tables accessible to the user
ALL_TAB_COLUMNS	Columns of user's tables, views and clusters
ALL_TAB_COL_STATISTICS	Columns of user's tables, views and clusters
ALL_TAB_COMMENTS	Comments on tables and views accessible to the user
ALL_TRIGGERS	Triggers accessible to the current user
ALL_TRIGGER_COLS	Column usage in user's triggers or in triggers on user's tables
ALL_TYPES	Description of types accessible to the user
ALL_UPDATABLE_COLUMNS	Description of all updatable columns
ALL_USERS	Information about all users of the database
ALL_VIEWS	Description of views accessible to the user
DATABASE_COMPATIBLE_LEVEL	Database compatible parameter set via init.ora
DBA_DB_LINKS	All database links in the database
DBA_ERRORS	Current errors on all stored objects in the database
DBA_OBJECTS	All objects in the database
DBA_ROLES	All Roles which exist in the database
DBA_ROLE_PRIVS	Roles granted to users and roles
DBA_SOURCE	Source of all stored objects in the database
DBA_TABLESPACES	Description of all tablespaces
DBA_TAB_PRIVS	All grants on objects in the database
DBA_TRIGGERS	All triggers in the database
DBA_TS_QUOTAS	Tablespace quotas for all users
DBA_USERS	Information about all users of the database
DBA_VIEWS	Description of all views in the database
DICTIONARY	Description of data dictionary tables and views
DICT_COLUMNS	Description of columns in data dictionary tables and views
GLOBAL_NAME	global database name
NLS_DATABASE_PARAMETERS	Permanent NLS parameters of the database
NLS_INSTANCE_PARAMETERS	NLS parameters of the instance
NLS_SESSION_PARAMETERS	NLS parameters of the user session
PRODUCT_COMPONENT_VERSION	version and status information for component products
ROLE_TAB_PRIVS	Table privileges granted to roles
SESSION_PRIVS	Privileges which the user currently has set
SESSION_ROLES	Roles which the user currently has enabled.
SYSTEM_PRIVILEGE_MAP	Description table for privilege type codes. Maps privilege type numbers to type names
TABLE_PRIVILEGES	Grants on objects for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
TABLE_PRIVILEGE_MAP	Description table for privilege (auditing option) type codes. Maps privilege (auditing option) type numbers to type names

Mandatory security in SQL

The topic was introduced above. First, classify the subjects (users and their agents) and the database objects concerned. Classify the means by which each has to be assigned a number indicating the security level, as it will be enforced by the applied rules (e.g. the Bell-LaPadula model (2005)).

The classification has to apply by table, by tuple, by attribute and by attribute value as appropriate to the requirement. Separate classifications may be needed to deal with create (INSERT), read (SELECT), UPDATE and DELETE permissions - though this will depend on the rules to be applied. The rules themselves may relate these options as they do in the model quoted.

Additional controls may be needed to deal with statistical security - these might restrict the number of tuples that can be retrieved, or add inaccuracies to the retrieved data, or provide controls on allowed query sequences (to identify and restrict users unnecessarily performing extractive analysis). Design considerations need to decide whether sensitive tuples should be maintained on the database and, if so, how.

So this converts mandatory security to the design of a security database (all the above information has to be stored in tables) with the associated transaction design, to implement the access rules and to control access to the security tables themselves. Mandatory security has to be built from the discretionary SQL tools.

Review question 1

- Distinguish discretionary from mandatory security.
- Describe the log-in process step by step.
- Explain the nature and use of surrogate information.
- Explain the implementation of access control in a distributed database.

Review question 2

What are the threats in the case below? Explain the nature of the threats. Write these down (one paragraph) before you read the notes. If you feel you need more information explain what you need to know?

Case

A senior manager is recorded as being in his office late one night. Subsequently at the time he was in his office the audit trail records several unsuccessful attempts to access database objects using a password of a member of clerical staff to objects to which the manager had no rights of access.

Notes pages

Data protection

Treat the following principles as abstract. Every company that has implemented data protection has followed these guides but, as usual, 'the devil is in the detail'. If you can be sure your database system complies with these you have done well. (And are you happy with storing personal images?)

- The information to be contained in personal data shall be obtained and personal data shall be processed fairly and lawfully.
- Personal data shall be held only for one or more specified and lawful purposes.
- Personal data held for any purpose or purposes shall not be used or disclosed in any manner incompatible with that purpose or those purposes.
- Personal data held for any purpose or purposes shall be adequately relevant and not excessive in relation to that purpose or those purposes.
- Personal data shall be accurate and, where necessary, kept up-to-date.
- Personal data held for any purpose or purposes shall not be kept for longer than is necessary for that purpose or purposes.
- An individual shall be entitled - at reasonable intervals and without undue delay or expense -

1. to be informed by any data user if he/she holds personal data of which the individual is the subject;
2. to access any such data held by a data user; and
3. where appropriate, to have such data corrected or erased.

- Appropriate security measures shall be taken against unauthorised access to, or alteration, disclosure or destruction of, personal data and against loss or destruction of personal data.

Computer misuse

Hacking offences

These are:

- Simple unauthorised access – merely accessing data to which you are not entitled. The law is not concerned with the systems control per se.
- Unauthorised access (with intent to commit an offence) – so you don't actually need to have succeeded, but rather just to have intended, to do something to the database.

- Unauthorised modification – no one can even attempt access without making some changes, but the purpose is to penalise outcomes.

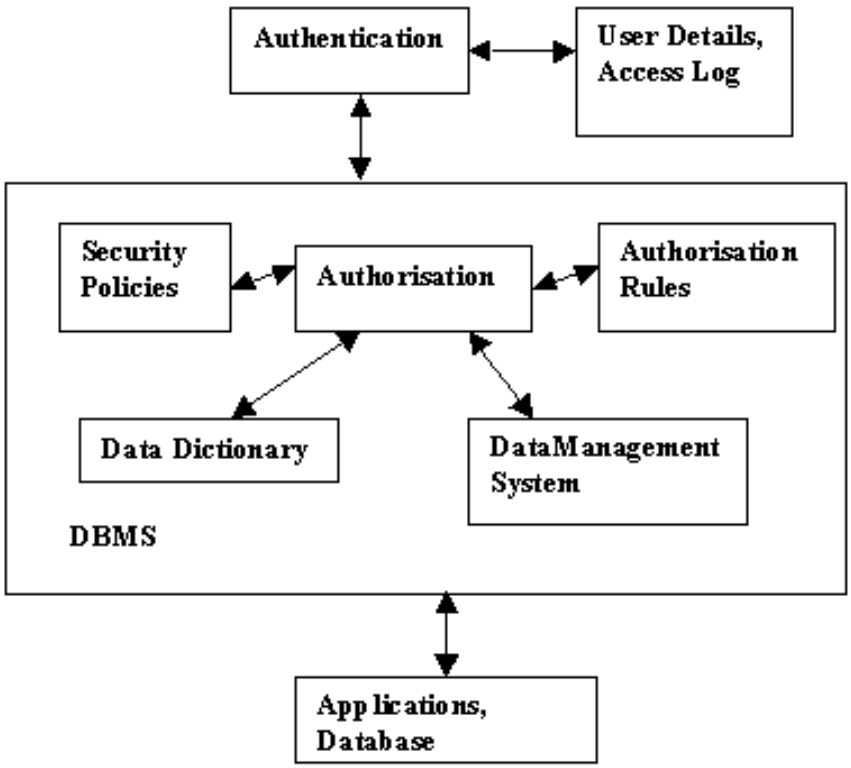
Clearly systems must record all accesses and attempted accesses, whether remote or local. Failure to do so may be seen as negligence, conspiracy or condoning on the part of system managers and designers.

Offences of introducing viruses are often covered in misuse legislation but are not included here. Yet viruses that attack DBMSs are of course possible.

Security plan

- Identify the user community.
- Gather the database information.
- Determine the types of user account (i.e. associate database objects and user roles).
- Undertake a threat analysis.
- Establish DBA authorities and procedures.
- Establish policies for managing (creating, deleting, auditing) user accounts.
- Determine the user tracking policy.
- Establish the user identification method.
- Define security incidents and reporting procedure.
- Assess the sensitivity of specific data objects.
- Establish standards and enforcement procedures (as well as back-up and recovery plans, of course).

Authentication and authorisation schematic



Authentication and authorisation

Generally speaking, when you log into a system, you want to be satisfied that you have logged into the right system and the system equally wants to be satisfied that you are who you claim to be. Think about this with an Internet Banking System as an example. Could a line be intercepted and the person on the other side pretend to be the bank while you disclosed passwords and account numbers? Could someone access the bank and empty your accounts? The part of the process that deals with this area is the authentication server.

There are only two ways to establish authentication: by means of shared secrets – things known only to you and the system - or by appealing to another system that has gone through the same process and is trusted. It is the second point that enables distributed transactions, at least without multiple log-ins. Shared secrets include passwords and personal information as reported to and stored

by the system. In each case, a dialogue appropriate to the situation has to be developed, monitored and managed. In the latter case, notice the need for secure communication between different system components, as secret information or its surrogate has to be transmitted between systems.

The result of authentication is a vector that contains an authentication identifier, usually with other information including date and time. Note that authentication is quite separate from access to database resources. You need to have obtained an authentication identifier before you start accessing the database. See the extension reading for some further insight on authentication.

In an SQL database system, the authentication process is initiated by the CONNECT statement. After successful execution of CONNECT, the resources of the database become potentially available.

Authorisation is permission given by the DBMS to use defined database resources and is based on the authentication identifier and a record of the permissions the owner of that identifier has. The identifier is recorded with the objects the user is allowed to access and the actions that can be performed on those objects. These are recorded either as a separate security database or as part of the normal system tables – accessible only to those with DBA privileges and higher.

Each SQL object has an owner. The owner has privileges on the objects owned. No other user can have any privileges (or even know the object exists) unless the owner supplies the necessary permission. In normal development, the DBA or system administrator will be the owner of the major assets. The scheme is basically discretionary. You need to look at security in SQL more as a toolkit than a solution.

Access control in SQL is implemented by transactions using the GRANT statement. These associate privileges with users and assets. Assets include data items (tables, views) and the privileges are the responsibility of the asset owner.

Access control activities

Overview

The exercise here is a simple implementation from a problem statement. You will produce the schema for a simple database (three tables), populate it with data, and then establish the access controls, test them and review the effect. Suggestions are then made for you to investigate some of the system tables.

The problem

You have received the following account of the access security requirements for an SQL database system.

The database has three tables: CUSTOMER (keyed on name and street number); ORDER (keyed on date-of-receipt and the foreign key from CUSTOMER), and ORDER-ITEM (keyed on item-name and the foreign key from ORDER).

The system is to process sales orders. Entry and access to documents and the progressing of orders is handled by the sales office. There are currently seven staff in this office. Tracey, the supervisor, needs to be able to see and change everything. Bill, Sheila and Govind do most of the routine work, but especially they cannot create new customers.

There are a few other things. Make sure that large orders, say above £1000, cannot be handled by anyone except Tracey and Govind. Also ensure that a temporary member of staff can process orders but cannot see the customer details.

1. Produce an analysis and the necessary SQL statements to handle the problems.
2. Implement the database, populate it with test data and apply your security statements.
3. Test your access control mechanism thoroughly.
4. If, when you see the sample solution, either your analysis or testing are faulty, you need to find out why.
5. Indicate any additional SQL security measures that could be taken.
6. Comment on the strengths and weaknesses of the measures you have taken.

The following activities will help you through solving the problem, step by step.

Activity 1 – Creating the database schema

Using the data definition statements from SQL, produce a script to create three tables (table 1 is related to many rows of table 2 which is related to many rows of table 3).

You may already have a database schema which can be used for this purpose, but in any case you must ensure that the primary and foreign keys are correctly declared.

Activity 2 – Populating the database

This is a security exercise so the data itself need not be to normal test data standards. Make sure that the data is easily recognisable – use meaningful values and remember that foreign key fields should be present.

Activity 3 – Analysing the problem

Treat the names given as user-roles (otherwise the problem becomes too obscure for a first attempt). Develop a matrix of user against database resource. For this exercise, the database resources in question are the base tables and views (virtual tables).

Your matrix will make it clear where access is to be permitted and to whom. These requirements translate directly to an SQL script of GRANT statements. Produce this script on paper and check it manually.

Activity 4 – Executing the security script (if you have a DBMS that permits this)

If you don't have an executing environment, then get a friend to critique your work so far.

Activity 5 – Testing the access control (if you have a DBMS that permits this)

Formulate SELECT statements from the problem specification and issue SELECT statements to check that they have been correctly implemented. If you find problems, correct them at the point they occurred.

Activity 6 – Conclusion

Indicate any additional SQL security measures that could be taken, and comment on the strengths and weaknesses of the measures you have taken.

Activity 7 – Postscript

Well, how well did you do?

Remember Tracey?

After your work, she thought she should have a raise. She asked and was refused and then returned to her desk. To answer these questions, refer to your database design and security script.

1. Tracey then tried to delete the CUSTOMER table. Did she succeed?
2. I hope not, but if so, why? Did you not inadvertently give her SYSADM privileges?
3. She then tried to delete some customers. Did she succeed? Did the deletes cascade?

4. She tried to insert a line in all orders over £1000 for 500 coffee machines. Did she succeed?
5. And how was the problem detected?
6. She tried to change her password? Did she succeed?
7. How much privilege can any one individual ever be given?

Chapter 13. Concurrency Control

Table of contents

    - Discussion topic 1
    - Discussion topic 2
    - Discussion topic 3
- Additional content and exercises
    - Additional content
        * Concurrency control based on timestamp ordering
        * Multiversion concurrency control techniques
        * Multiversion techniques based on timestamp ordering
        * Multiversion two-phase locking
        * Granularity of data items
    - Additional exercises
        * Extension exercise 1
        * Extension exercise 2
        * Extension exercise 3
        * Extension exercise 4

Objectives

At the end of this chapter you should be able to:

- Describe the nature of transactions and the reasons for designing database systems around transactions.
- Explain the causes of transaction failure.
- Analyse the problems of data management in a concurrent environment.
- Critically compare the relative strengths of different concurrency control approaches.

Introduction

In parallel with this chapter, you should read Chapter 20 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

The purpose of this chapter is to introduce the fundamental technique of concurrency control, which provides database systems with the ability to handle many users accessing data simultaneously. In addition, this chapter helps you understand the functionality of database management systems, with special reference to online transaction processing (OLTP). The chapter also describes the problems that arise out of the fact that users wish to query and update stored data at the same time, and the approaches developed to address these problems, together with their respective strengths and weaknesses in a range of practical situations.

There are a number of concepts that are technical and unfamiliar. You will be expected to be able to handle these concepts but not to have any knowledge of the detailed algorithms involved. This chapter fits closely with the one on backup and recovery, so you may want to revisit this chapter later in the course to review the concepts. It will become clear from the information on concurrency control that there are a number of circumstances where recovery procedures may need to be invoked to salvage previous or currently executing transactions. The material covered here will be further extended in the chapter on distributed database systems, where we shall see how effective concurrency control can be implemented across a computer network.

Context

Many criteria can be used to classify DBMSs, one of which is the number of users supported by the system. Single-user systems support only one user at a time and are mostly used with personal computers. Multi-user systems, which include the majority of DBMSs, support many users concurrently.

In this chapter, we will discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. We will start the chapter by introducing some basic concepts of transaction processing. Why concurrency control and recovery are necessary in a database system is then discussed. The concept of an atomic transaction and additional concepts related to transaction processing in database systems are introduced. The concepts of atomicity, consistency, isolation and durability – the so-called ACID properties that are considered desirable in transactions - are presented.

The concept of schedules of executing transactions and characterising the recoverability of schedules is introduced, with a detailed discussion of the concept of serialisability of concurrent transaction executions, which can be used to define correct execution sequences of concurrent transactions.

We will also discuss recovery from transaction failures. A number of concurrency control techniques that are used to ensure noninterference or isolation of concurrently executing transactions are discussed. Most of these techniques ensure serialisability of schedules, using protocols or sets of rules that guarantee serialisability. One important set of protocols employs the technique of locking data items, to prevent multiple transactions from accessing the items concurrently. Another set of concurrency control protocols use transaction timestamps. A timestamp is a unique identifier for each transaction generated by the system. Concurrency control protocols that use locking and timestamp ordering to ensure serialisability are both discussed in this chapter.

An overview of recovery techniques will be presented in a separate chapter.

Concurrent access to data

Concept of transaction

The first concept that we introduce to you in this chapter is a transaction. A transaction is the execution of a program that accesses or changes the contents of a database. It is a logical unit of work (LUW) on the database that is either completed in its entirety (COMMIT) or not done at all. In the latter case, the transaction has to clean up its own mess, known as ROLLBACK. A transaction could be an entire program, a portion of a program or a single command.

The concept of a transaction is inherently about organising functions to manage data. A transaction may be distributed (available on different physical systems or organised into different logical subsystems) and/or use data concurrently with multiple users for different purposes.

Online transaction processing (OLTP) systems support a large number of concurrent transactions without imposing excessive delays.

Transaction states and additional operations

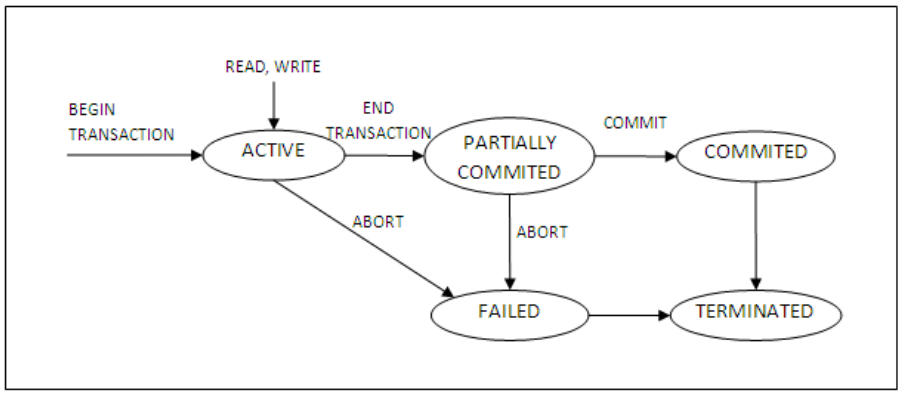
For recovery purposes, a system always keeps track of when a transaction starts, terminates, and commits or aborts. Hence, the recovery manager keeps track of the following transaction states and operations:

- **BEGIN_TRANSTRACTION:** This marks the beginning of transaction execution.
- **READ or WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSTRACTION:** This specifies that read and write operations have ended and marks the end limit of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates concurrency control, or for some other reason (rollback).
- **COMMIT_TRANSTRACTION:** This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **ROLLBACK (or ABORT):** This signals the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

In addition to the preceding operations, some recovery techniques require additional operations that include the following:

- **UNDO:** Similar to rollback, except that it applies to a single operation rather than to a whole transaction.
- **REDO:** This specifies that certain transaction operations must be redone to ensure that all the operations of a committed transaction have been applied successfully to the database.

A state transaction diagram is shown below:



It shows clearly how a transaction moves through its execution states. In the diagram, circles depict a particular state; for example, the state where a transaction has become active. Lines with arrows between circles indicate transitions or changes between states; for example, read and write, which correspond to computer processing of the transaction.

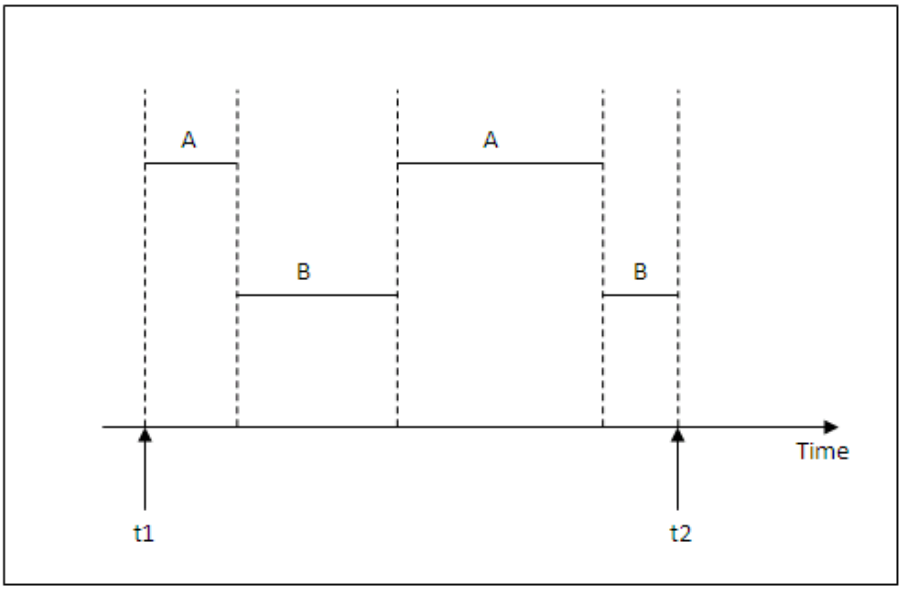
A transaction goes into an active state immediately after it starts execution, where it can issue read and write operations. When the transaction ends, it moves to the partially committed state. At this point, some concurrency control techniques require that certain checks be made to ensure that the transaction did not interfere with other executing transactions. In addition, some recovery protocols are needed to ensure that a system failure will not result in inability to record the changes of the transaction permanently. Once both checks are successful, the transaction is said to have reached its commit point and enters the committed state. Once a transaction enters the committed state, it has concluded its execution successfully.

However, a transaction can go to the failed state if one of the checks fails or if it aborted during its active state. The transaction may then have to be rolled back to undo the effect of its write operations on the database. The terminated state corresponds to the transaction leaving the system. Failed or aborted transactions may be restarted later, either automatically or after being resubmitted, as brand new transactions.

Interleaved concurrency

Many computer systems, including DBMSs, are used simultaneously by more than one user. This means the computer runs multiple transactions (programs) at the same time. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Systems in banks, insurance agencies, stock exchanges and the like are also operated by many users who submit transactions concurrently to the system. If, as is often the case, there is only one CPU, then only one program can be processed at a time. To avoid excessive delays, concurrent systems execute some commands from one program (transaction), then suspended that program and execute some commands from the next program, and so on. A program is resumed at the point where it was suspended when it gets its turn to use the CPU again. This is known as interleaving.

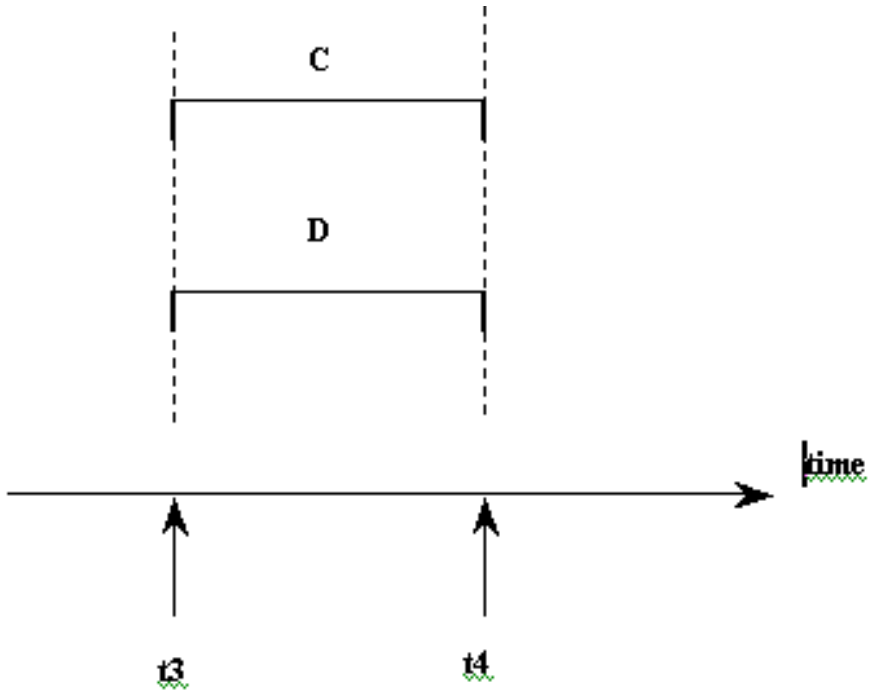
The figure below shows two programs A and B executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when an executing program requires an input or output (I/O) operation, such as reading a block of data from disk. The CPU is switched to execute another program rather than remaining idle during I/O time.



Interleaved vs simultaneous concurrency

If the computer system has multiple hardware processors (CPUs), simultaneous processing of multiple programs is possible, leading to simultaneous rather than

interleaved concurrency, as illustrated by program C and D in the figure below. Most of the theory concerning concurrency control in databases is developed in terms of interleaved concurrency, although it may be adapted to simultaneous concurrency.



Genuine vs appearance of concurrency

Concurrency is the ability of the database management system to process more than one transaction at a time. You should distinguish genuine concurrency from the appearance of concurrency. The database management system may queue transactions and process them in sequence. To the users it will appear to be concurrent but for the database management system it is nothing of the kind. This is discussed under serialisation below.

Read and write operations

We deal with transactions at the level of data items and disk blocks for the purpose of discussing concurrency control and recovery techniques. At this level, the database access operations that a transaction can include are:

- read_item(X): Reads a database item named X into a program variable also named X.
- write_item(X): Writes the value of program variable X into the database item named X.

Executing a read_item(X) command includes the following steps:

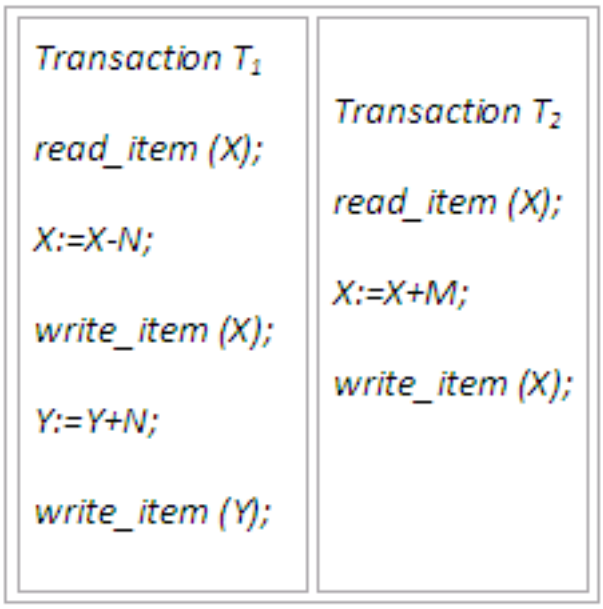
1. Find the address of the disk block that contains item X.
2. Copy the disk block into a buffer in main memory if that disk is not already in some main memory buffer.
3. Copy item X from the buffer to the program variable named X.

Executing a write_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy the disk block into a buffer in main memory if that disk is not already in some main memory buffer.
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Step 4 is the one that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store back a modified disk block that is in a main memory buffer is handled by the recovery manager or the operating system.

A transaction will include read and write operations to access the database. The figure below shows examples of two very simple transactions. Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction.



The above two transactions submitted by any two different users may be executed concurrently and may access and update the same database items (e.g. X). If this concurrent execution is uncontrolled, it may lead to problems such as an inconsistent database. Some of the problems that may occur when concurrent transactions execute in an uncontrolled manner are discussed in the next section.

Activity 1 - Looking up glossary entries

In the Concurrent Access to Data section of this chapter, the following phrases have glossary entries:

- transaction
- interleaving
- COMMIT
- ROLLBACK

1. In your own words, write a short definition for each of these terms.
2. Look up and make notes of the definition of each term in the module glossary.
3. Identify (and correct) any important conceptual differences between your definition and the glossary entry.

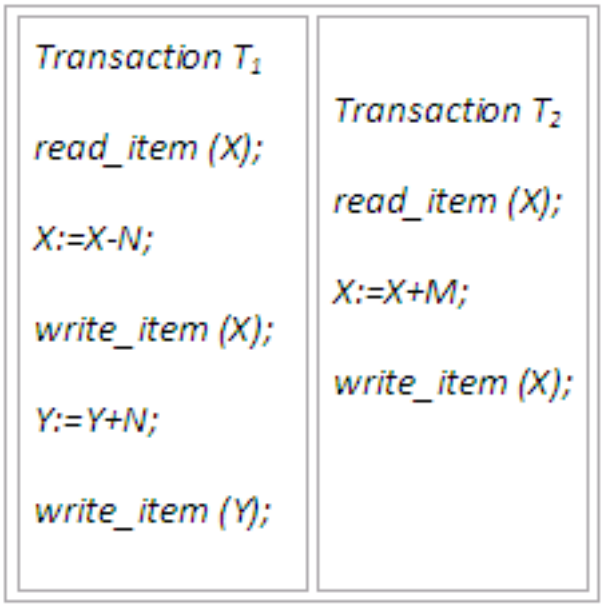
Review question 1

1. Explain what is meant by a transaction. Discuss the meaning of transaction states and operations.
2. In your own words, write the key feature(s) that would distinguish an interleaved concurrency from a simultaneous concurrency.
3. Use an example to illustrate your point(s) given in 2.
4. Discuss the actions taken by the read_item and write_item operations on a database.

Need for concurrency control

Concurrency is the ability of the DBMS to process more than one transaction at a time. This section briefly overviews several problems that can occur when concurrent transactions execute in an uncontrolled manner. Concrete examples are given to illustrate the problems in details. The related activities and learning tasks that follow give you a chance to evaluate the extent of your understanding of the problems. An important learning objective for this section of the chapter is to understand the different types of problems of concurrent executions in OLTP, and appreciate the need for concurrency control.

We illustrate some of the problems by referring to a simple airline reservation database in which each record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a named data item, among other information. Recall the two transactions T1 and T2 introduced previously:



Transaction T1 cancels N reservations from one flight, whose number of reserved seats is stored in the database item named X, and reserves the same number of seats on another flight, whose number of reserved seats is stored in the database item named Y. A simpler transaction T2 just reserves M seats on the first flight referenced in transaction T1. To simplify the example, the additional portions of the transactions are not shown, such as checking whether a flight has enough seats available before reserving additional seats.

When an airline reservation database program is written, it has the flight numbers, their dates and the number of seats available for booking as parameters; hence, the same program can be used to execute many transactions, each with different flights and number of seats to be booked. For concurrency control purposes, a transaction is a particular execution of a program on a specific date, flight and number of seats. The transactions T1 and T2 are specific executions of the programs that refer to the specific flights whose numbers of seats are stored in data item X and Y in the database. Now let's discuss the types of problems we may encounter with these two transactions.

The lost update problem

The lost update problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect. That is, interleaved use of the same data item would cause some problems when an update operation from one transaction overwrites another update from a second transaction.

An example will explain the problem clearly. Suppose the two transactions T1 and T2 introduced previously are submitted at approximately the same time. It is possible when two travel agency staff help customers to book their flights at more or less the same time from a different or the same office. Suppose that their operations are interleaved by the operating system as shown in the figure below:

Time	T_1	T_2	Comments
1	read_item(X);		Read operation is performed in T_1 at time step 1.
2	X:=X-N;		Value of data item X is modified.
3		read_item(X);	Read in value of X (Which value is read in?)
4		X:=X+M	Value of data item X is modified.
5	write_item(X);		Write operation is performed in T_1.
6	read_item(Y);		Read operation is performed in T_1.
7		write_item(X);	Write data item X to database (What value is written in?)
8	Y:=Y+N;		Value of data item is modified.
9	write_item(Y);		Write data item Y to database.

The above interleaved operation will lead to an incorrect value for data item X, because at time step 3, T2 reads in the original value of X which is before T1 changes it in the database, and hence the updated value resulting from T1 is lost. For example, if X = 80, originally there were 80 reservations on the flight, N = 5, T1 cancels 5 seats on the flight corresponding to X and reserves them on the flight corresponding to Y, and M = 4, T2 reserves 4 seats on X.

The final result should be X = 80 – 5 + 4 = 79; but in the concurrent operations of the figure above, it is X = 84 because the update that cancelled 5 seats in T1 was lost.

The detailed value updating in the flight reservation database in the above example is shown below:

Time	T_1	T_2	Value
1	read_item(X);		X = 80
2	X:=X-N;		X =80–5=75 (which is not written into database)
3		read_item(X);	X = 80 (T2 still reads in the original value of X, the updated value of X is lost.)
4		X:=X+M	X=80+4=84
5	write_item(X);		X=75 is written into database
6	read_item(Y);		
7		write_item(X);	X=84 over writes X=75, a wrong record is written in database
8	Y:=Y+N;		
9	write_item(Y);		

Uncommitted dependency (or dirty read / temporary update)

Uncommitted dependency occurs when a transaction is allowed to retrieve or (worse) update a record that has been updated by another transaction, but which has not yet been committed by that other transaction. Because it has not yet been committed, there is always a possibility that it will never be committed but rather rolled back, in which case, the first transaction will have used some data that is now incorrect - a dirty read for the first transaction.

The figure below shows an example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the 'temporary' value of X, which will not be recorded permanently in the database because of the failure of T1. The value of item X that is read by T2 is called dirty data, because it has been created by a transaction that has not been completed and committed yet; hence this problem is also known as the dirty read problem. Since the dirty data read in by T2 is only a temporary value of X, the problem is sometimes called temporary update too.

Time	T_1	T_2	Comment
1	read_item(X);		
2	X:=X-N;		
3	write_item(X);		X is temporarily updated
4		read_item(X);	
5		X:=X+M	
6		write_item(X);	
7	read_item(Y);		
...	...	...	
	ROLLBACK		**T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the temporary incorrect value of X**

The rollback of transaction T1 may be due to a system crash, and transaction T2 may already have terminated by that time, in which case the crash would not cause a rollback to be issued for T2. The following situation is even more unacceptable:

Time	T_1	T_2	Comment
1	read_item(X);		
2	X:=X-N;		
3	write_item(X);		X is temporarily updated
4		read_item(X);	
5		X:=X+M	
6		write_item(X);	
7		**COMMIT**	**T_2 loses update chance**
8	read_item(Y);		
...	...	...	
	ROLLBACK		**T_2 depends on uncommitted value and loses an update at time step 7**

In the above example, not only does transaction T2 becomes dependent on an uncommitted change at time step 6, but it also loses an update at time step 7, because the rollback in T1 causes data item X to be restored to its value before time step 1.

Inconsistent analysis

Inconsistent analysis occurs when a transaction reads several values, but a second transaction updates some of these values during the execution of the first. This problem is significant, for example, if one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records. The aggregate function may calculate some values before they are updated and others after they are updated. This causes an inconsistency.

For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown below occurs, the result of T3 will be off by amount N, because T3 reads the value of X after N seats have been subtracted from it, but reads the value of Y before those N seats have been added to it.

Time	T_1	T_3	Comment
1		sum:=0;	
2		read_item(A);	
3		sum:=sum+A;	
4	read_item(X);		
5	X:=X-N;		
6	write_item(X);		
7		read_item(X);	**T_3 reads X after N is subtracted from it**
8		sum:=sum+X;	
9		read_item(Y);	**T_3 reads Y before N is added to it**
10		sum:=sum+Y;	**A wrong summary is resulted (off by N)**
11	read_item(Y);		
12	Y:=Y+N;		
13	write_item(Y);		

Other problems

Another problem that may occur is the unrepeatable read, where a transaction T1 reads an item twice, and the item is changed by another transaction T2 between reads. Hence, T1 receives different values for its two reads of the same item.

Phantom record could occur when a transaction inserts a record into the database, which then becomes available to other transactions before completion. If the transaction that performs the insert operation fails, it appears that a record in the database disappears later.

Exercise 1

Typical problems in multi-user environment when concurrency access to data is allowed

Some problems may occur in multi-user environment when concurrency access to database is allowed. These problems may cause data stored in the multi-user DBMS to be damaged or destroyed. Four interleaved transaction schedules are given below. Identify what type of problems they have.

1.	Transaction A	Time	Transaction B
	read_item(X)	T_1	
	X:= X+1	T_2	
		T_3	read_item(X)
	write_item(X)	T_4	
		T_5	write_item(X)
2.	Transaction A	Time	Transaction B
		T_1	read-item(X)
		T_2	X:= X-1
		T_3	write_item(X)
	read-item(X)	T_4	
	write_item(X)	T_5	
		T_6	ROLLBACK
3.	Transaction A	Time	Transaction B
		T_1	read-item(X)
		T_2	X:= X-1
		T_3	write_item(X)
	sum:=0;	T_4	
	read_item(X);	T_5	
	sum:=sum+X;	T_6	
	read_item(Y);	T_7	
	sum:=sum+Y;	T_8	
		T_9	read_item(Y);
		T_{10}	Y:=Y+1;
		T_{11}	write_item(Y) ;
4.	Transaction A	Time	Transaction B
	read_item(X)	T_1	
	X:= X+1	T_2	
		T_3	read_item(Y)
	write_item(X)	T_4	
		T_5	write_item(Y)

Exercise 2

Inconsistent analysis problem

Interleaved calculation of aggregates may have some aggregates on early data and some on late data if other transactions are able to update the data. This will cause incorrect summary. Consider the situation below, in which a number of account records have the following values:

ACC1	ACC2	ACC3
40	50	30

To transfer 10 from ACC3 to ACC1 while concurrently calculating the total funds in the three accounts, the following sequence of events may occur. Show the value of each data item in the last column, and discuss the reason for an incorrect summary value.

Transaction A	Time	Transaction B	Value of Sum
sum:=0;	t_1		
read_item(ACC1);	t_2		
sum:=sum+ACC1;	t_3		
	t_4	read_item(ACC3);	
	t_5	ACC3:=ACC3-10;	
	t_6	write_item(ACC3);	
	t_7	read_item(ACC1);	
	t_8	ACC1:=ACC1+10;	
	t_9	write_item(ACC1);	
	t_{10}	COMMIT	
read_item(ACC3)	t_{11}		
sum:=sum+ACC3	t_{12}		
read_item(ACC2);	t_{13}		
sum:=sum+ACC2;	t_{14}		

Review question 2

1. What is meant by interleaved concurrent execution of database transactions in a multi-user system? Discuss why concurrency control is needed, and give informal examples.

Need for recovery

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either:

1. all the operations in the transaction are completed successfully and the effect is recorded permanently in the database; or
2. the transaction has no effect whatsoever on the database or on any other transactions.

The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not. This may happen if a transaction fails after executing some of its operations but before executing all of them.

Transaction problems

The practical aspects of transactions are about keeping control. There are a variety of causes of transaction failure. These may include:

1. Concurrency control enforcement: Concurrency control method may abort the transaction, to be restarted later, because it violates serialisability (the need for transactions to be executed in an equivalent way as would have resulted if they had been executed sequentially), or because several transactions are in a state of deadlock.
2. Local error detected by the transaction: During transaction executions, certain conditions may occur that necessitate cancellation of the transaction (e.g. an account with insufficient funds may cause a withdrawal transaction from that account to be cancelled). This may be done by a programmed ABORT in the transaction itself.
3. A transaction or system error: Some operation in the transaction may cause it to fail, such as integer overflow, division by zero, erroneous parameter values or logical programming errors.
4. User interruption of the transaction during its execution, e.g. by issuing a control-C in a VAX/ VMS or UNIX environment.
5. System software errors that result in abnormal termination or destruction of the database management system.
6. Crashes due to hardware malfunction, resulting in loss of internal (main and cache) memory (otherwise known as system crashes).
7. Disk malfunctions such as read or write malfunction, or a disk read/write head crash. This may happen during a read or write operation of the transaction.

8. Natural physical disasters and catastrophes such as fires, earthquakes or power surges; sabotages, intentional contamination with computer viruses, or destruction of data or facilities by operators or users.

Failures of types 1 to 6 are more common than those of types 7 or 8. Whenever a failure of type 1 through 6 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of 7 or 8 do not happen frequently; if they do occur, it is a major task to recover from these types of failure.

Desirable properties of transactions (ACID)

The acronym ACID indicates the properties of any well-formed transaction. Any transaction that violates these principles will cause failures of concurrency. A brief description of each property is given first, followed by detailed discussions.

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all. A transaction does not partly happen.
2. **Consistency:** The database state is consistent at the end of a transaction.
3. **Isolation:** A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks (see 'Atomicity' below) of transactions unnecessary.
4. **Durability:** When a transaction has made a change to the database state and the change is committed, this change is permanent and should be available to all other transactions.

Atomicity

Atomicity is built on the idea that you cannot split an atom. If a transaction starts, it must finish – or not happen at all. This means if it happens, it happens completely; and if it fails to complete, there is no effect on the database state.

There are a number of implications. One is that transactions should not be nested, or at least cannot be nested easily. A nested transaction is where one transaction is allowed to initiate another transaction (and so on). If one of the nested transactions fails, the impact on other transactions leads to what is known as a cascading rollback.

Transactions need to be identified. The database management system can assign a serial number of timestamp to each transaction. This identifier is required so each activity can be logged. When a transaction fails for any reason, the log is used to roll back and recover the correct state of the database on a transaction basis.

Rolling back and committing transactions

There are two ways a transaction can terminate. If it executes to completion, then the transaction is said to be committed and the database is brought to a new consistent state. Committing a transaction signals a successful end-of-transaction. It tells the transaction manager that a logical unit of work has been successfully completed, the database is (or should be) in a consistent state again, and all of the updates made by that unit of work (transaction) can now be made permanent. This point is known as a synchronisation point. It represents the boundary between two consecutive transactions and corresponds to the end of a logical unit of work.

The other way a transaction may terminate is that the transaction is aborted and the incomplete transaction is rolled back and restored to the consistent state it was in before the transaction started. A rollback signals an unsuccessful end of a transaction. It tells the transaction manager that something has gone wrong, the database might be in an inconsistent state and all of the updates made by the logical unit of work so far must be undone. A committed transaction cannot be aborted and rolled back.

Atomicity is maintained by commitment and rollback. The major SQL operations that are under explicit user control that establish such synchronisation points are COMMIT and ROLLBACK. Otherwise (default situation), an entire program is regarded as one transaction.

Consistency

The database must start and finish in a consistent state. You should note that in contrast, during a transaction, there will be times where the database is inconsistent. Some part of the data will have been changed while other data has yet to be.

A correct execution of the transaction must take the database from one consistent state to another, i.e. if the database was in a consistent state at the beginning of transaction, it must be in a consistent state at the end of that transaction.

The consistency property is generally considered to be the responsibility of the programmers who write the database programs or the DBMS module that enforces integrity constraints. It is also partly the responsibility of the database management system to ensure that none of the specified constraints are violated. The implications of this are the importance of specifying the constraints and domains within the schema, and the validation of transactions as an essential part of the transactions.

Isolation

A transaction should not make its update accessible to other transactions until it has terminated. This property gives the transaction a measure of relative independence and, when enforced strictly, solves the temporary update problem. In general, various levels of isolation are permitted. A transaction is said to have degree 0 isolation if it does not overwrite the dirty reads of higher-degree transactions. A degree 1 isolation transaction has no lost updates, and degree 2 isolation has no lost update and no dirty reads. Finally, degree 3 isolation (also known as true isolation) has, in addition to degree 2 properties, repeatable reads.

Isolation refers to the way in which transactions are prevented from interfering with each other. You might think that one transaction should never be interfered with by any other transactions. This is nearly like insisting on full serialisation of transactions with little or no concurrency. The issues are those of performance.

Durability or permanency

Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

At the end of a transaction, one of two things will happen. Either the transaction has completed successfully or it has not. In the first case, for a transaction containing write_item operations, the state of the database has changed, and in any case, the system log has a record of the activities. Or, the transaction has failed in some way, in which case the database state has not changed, though it may have been necessary to use the system log to roll back and recover from any changes attempted by the transaction.

Review question 3

1. Discuss different types of possible transaction failures, with some examples.
2. Transactions cannot be nested inside one another. Why? Support your answer with an example.

Serialisability

This is a criterion that most concurrency control methods enforce. Informally, if the effect of running transactions in an interleaved fashion is equivalent to running the same transactions in a serial order, they are considered serialisable. We have used the word 'schedule' without a definition before in this chapter. In this section, we will first define the concept of transaction schedule, and then we characterise the types of schedules that facilitate recovery when failures occur.

Schedules of transactions

When transactions are executed concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history). A schedule S of n transactions T1, T2, … Tn is an ordering of operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear in the same order in which they occur in Ti. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Suppose that two users – airline reservation clerks – submit to the DBMS transactions T1 and T2 introduced previously at approximately the same time. If no interleaving is permitted, there are only two possible ways of ordering the operations of the two transactions for execution:

1. Schedule A: Execute all the operations of transaction T1 in sequence followed by all the operations of transaction T2 in sequence.
2. Schedule B: Execute all the operations of transaction T2 in sequence followed by all the operations of transaction T1 in sequence.

These alternatives are shown below:

time	Schedule A		Schedule B	
	T_1	T_2	T_1	T_2
1	read_item(X);			read_item(X);
2	X:=X-N;			X:=X+M;
3	write_item(X);			write_item(X);
4	read_item(Y);		read_item(X);	
5	Y:=Y+N;		X:=X-N;	
6	write_item(Y);		write_item(X);	
7		read_item(X);	read_item(Y);	
8		X:=X+M;	Y:=Y+N;	
9		write_item(X);	write_item(Y);	

If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown below:

time	Schedule C		Schedule D	
	T_1	T_2	T_1	T_2
1	read_item(X);		read_item(X);	
2	X:=X-N;		X:=X-N;	
3		read_item(X);	write_item(X);	
4		X:=X+M;		read_item(X);
5	write_item(X);			X:=X+M;
6	read_item(Y);			write_item(X);
7		write_item(X);	read_item(Y);	
8	Y:=Y+N;		Y:=Y+N;	
9	write_item(Y);		write_item(Y);	

An important aspect of concurrency control, called serialisability theory, attempts to determine which schedules are 'correct' and which are not, and to develop techniques that allow only correct schedules. The next section defines serial and non-serial schedules, presents some of the serialisability theory, and discusses how it may be used in practice.

Serial schedules

Schedules A and B are called serial schedules because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: T1 and then T2 or T2 and then T1 in the diagram below. Schedules C and D are called non-serial because each sequence interleaves operations from the two transactions.

Formally, a schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called non-serial. One reasonable assumption we can make, if we consider the transactions to be independent, is that every serial schedule is considered correct. This is so because we assume that every transaction is correct if executed on its own (by the consistency property introduced previously in this chapter) and that transactions do not depend on one another. Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end without any interference from the operations of other transactions, we get a correct end result on the database. The problem with serial schedules is that they limit concurrency or interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time and making serial schedules generally unacceptable.

To illustrate our discussion, consider the schedules in the diagram below, and

assume that the initial values of database items are X = 90, Y = 90, and that N = 3 and M = 2. After executing transaction T1 and T2, we would expect the database values to be X = 89 and Y = 93, according to the meaning of the transactions. Sure enough, executing either of the serial schedules A or B gives the correct results. This is shown below:

time	Schedule A		Schedule B	
	T_1	T_2	T_1	T_2
1	read_item(X); 90			read_item(X); 90
2	X:=X-N; 87			X:=X+M; 92
3	write_item(X); 87			write_item(X); 92
4	read_item(Y); 90		read_item(X); 92	
5	Y:=Y+N; 93		X:=X-N; 89	
6	write_item(Y); **93**		write_item(X); **89**	
7		read_item(X); 87	read_item(Y); 90	
8		X:=X+M; 89	Y:=Y+N; 93	
9		write_item(X); **89**	write_item(Y); **93**	

Non-serial schedules

Schedules involving interleaved operations are non-serial schedules. Now consider the two non-serial schedules C and D. Schedule C gives the result X = 92 and Y = 93, in which the X value is erroneous, whereas schedule D gives the correct results. This is illustrated below:

time	Schedule C		Schedule D	
	T_1	T_2	T_1	T_2
1	read_item(X); 90		read_item(X); 90	
2	X:=X-N; 87		X:=X-N; 87	
3		read_item(X); **90**	write_item(X); 87	
4		X:=X+M; 92		read_item(X); **87**
5	Write_item(X); 87			X:=X+M; 89
6	read_item(Y); 90			write_item(X); **89**
7		write_item(X); 92	read_item(Y); 90	
8	Y:=Y+N; 93		Y:=Y+N; 93	
9	Write_item(Y); **93**		write_item(Y); **93**	

Schedule C gives an erroneous result because of the lost update problem. Transaction T2 reads the value of X before it is changed by transaction T1, so only the effect of T2 on X is reflected in the database. The effect of T1 on X is lost, overwritten by T2, leading to the incorrect result for item X.

However, some non-serial schedules do give the expected result, such as schedule D in the diagram above. We would like to determine which of the non-serial schedules always give a correct result and which may give erroneous results. The concept used to characterise schedules in this manner is that of serialisability of a schedule.

Serialisable schedule

A schedule S of n transactions is a serialisable schedule if it is equivalent to some serial schedule of the same n transactions. Notice that for n transactions, there are n possible serial schedules, and many more possible non-serial schedules. We can form two disjoint groups of the non-serial schedules: those that are equivalent to one (or more) of the serial schedules, and hence are serialisable; and those that are not equivalent to any serial schedule, and hence are not serialisable.

Saying that a non-serial schedule S is serialisable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct. For example, schedule D is a serialisable schedule, and it is a correct schedule, because schedule D gives the same results as schedules A and B, which are serial schedules. It is essential to guarantee serialisability in order to ensure

database correctness.

Now the question is: when are two schedules considered 'equivalent'? There are several ways to define equivalence of schedules. The simplest, but least satisfactory, definition of schedule equivalence involves comparing the effects of the schedules on the database. Intuitively, two schedules are called result equivalent if they produce the same final state of the database. However, two schedules may accidentally provide the same final state. For example, in the figure below, schedules S1 and S2 will produce the same database state if they execute on a database with an initial value of X = 100; but for other initial values of X, the schedules are not result equivalent. Hence result equivalent is not always the safe way to define schedules equivalence.

Here are two schedules that are equivalent for the initial value of X = 100, but are not equivalent in general:

S_1	S_2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

A serialisable schedule gives us the benefits of concurrent execution without giving up any correctness. In practice, however, it is quite difficult to test for the serialisability of a schedule. The interleaving of operations from concurrent operations is typically determined by the operating system scheduler. Factors such as system load, time of transaction submission, and priorities of transactions contribute to the ordering of operations in a schedule by the operating system. Hence, it is practically impossible to determine how the operations of a schedule will be interleaved beforehand to ensure serialisability. The approach taken in most practical systems is to determine methods that ensure serialisability without having to test the schedules themselves for serialisability after they are executed. One such method uses the theory of serialisability to determine protocols or sets of rules that, if followed by every individual transaction or if enforced by a DBMS concurrency control subsystem, will ensure serialisability of all schedules in which the transactions participate. Hence, in this approach we never have to concern ourselves with the schedule. In the next section of this chapter, we will discuss a number of such different concurrency control protocols that guarantee serialisability.

Exercise 3

Serial, non-serial and serialisable schedules

Given the following two transactions, and assuming that initially x = 3, and y

= 2,

Time	T_1	T_2
1	read_item(y)	read_item(y);
2	y:=y+3;	y:=y*1.2;
3	write_item(y);	write_item(y);
4	read_item(x);	read_item(x);
5	x:=x+y;	x:=x+y;
6	write_item(x);	write_item(x);

1. create all possible serial schedules and examine the values of x and y;
2. create a non-serial interleaved schedule and examine the values of x and y. Is this a serialisable schedule?

Review question 4

1. As a summary of schedules of transactions and serialisability of schedules, fill in the blanks in the following paragraphs:

   A schedule S of n transactions T1, T2, … Tn is an ordering of the operations of the transactions. The operations in S are exactly those operations in _____ including either a _____ or _____ operation as the last operation for each transaction in the schedule. For any pair of operations from the same transaction Ti, their order of appearance in S is as their order of appearance in Ti.

   In serial schedules, all operations of each transaction are executed _____ , without any operations from the other transactions. Every serial schedule is considered _____ .

   A schedule S of n transactions is a fertilisable schedule if it is equivalent to some _____ of the same n transactions.

   Saying that a non-serial schedule is serialisable is equivalent to saying that it is _____ , because it is equivalent to _____ .

2. Compare binary locks with exclusive/shared locks. Why is the latter type of locks preferable?

Locking techniques for concurrency control

One of the main techniques used to control concurrency execution of transactions (that is, to provide serialisable execution of transactions) is based on the concept of locking data items. A lock is a variable associate with a data item in the database and describes the status of that data item with respect to possible operations that can be applied to the item. Generally speaking, there is one lock for each data item in the database. The overall purpose of locking is to obtain maximum concurrency and minimum delay in processing transactions.

In the next a few sections, we will discuss the nature and types of locks, present several two-phase locking protocols that use locking to guarantee serialisability of transaction schedules, and, finally, we will discuss two problems associated with the use of locks – namely deadlock and livelock – and show how these problems are handled.

Types of locks

The idea of locking is simple: when a transaction needs an assurance that some object, typically a database record that it is accessing in some way, will not change in some unpredictable manner while the transaction is not running on the CPU, it acquires a lock on that object. The lock prevents other transactions from accessing the object. Thus the first transaction can be sure that the object in question will remain in a stable state as long as the transaction desires.

There are several types of locks that can be used in concurrency control. Binary locks are the simplest, but are somewhat restrictive in their use.

Binary locks

A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X is locked and cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, item X is unlocked, and it can be accessed when requested. We refer to the value of the lock associated with item X as LOCK(X).

Two operations, lock and unlock, must be included in the transactions when binary locking is used. A transaction requests access to an item X by issuing a lock(X) operation. If LOCK(X) = 1, the transaction is forced to wait; otherwise, the transaction sets LOCK(X) := 1 (locks the item) and allows access. When the transaction is through using the item, it issues an unlock(X) operation, which sets LOCK(X) := 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item. The DBMS has a lock manager subsystem to keep track of and control access to locks.

When the binary locking scheme is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock(X) operation if it already holds the lock on item X.
4. A transaction T will not issue an unlock(X) operation unless it already holds the lock on item X.

These rules can be enforced by a module of the DBMS. Between the lock(X) and unlock(X) operations in a transaction T, T is said to hold the lock on item X. At most, one transaction can hold the lock on a particular item. No two transactions can access the same item concurrently.

Shared and exclusive locks

The binary locking scheme described above is too restrictive in general, because at most one transaction can take hold on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only. However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, we can use a different type of lock called multiple-mode lock. In this scheme, there are three locking operations: read_lock(X), write_lock(X) and unlock(X). That is, a lock associated with an item X, LOCK(X), now has three possible states: 'read-locked', 'write-locked' or 'unlocked'. A read-locked item is also called share-locked, because other transactions are allowed to read access that item, whereas a write-locked item is called exclusive-locked, because a single transaction exclusively holds the lock on the item.

If a DBMS wishes to read an item, then a shared (S) lock is placed on that item. If a transaction has a shared lock on a database item, it can read the item but not update it. If a DBMS wishes to write (update) an item, then an exclusive (X) lock is placed on that item. If a transaction has an exclusive lock on an item, it can both read and update it. To prevent interference from other transactions, only one transaction can hold an exclusive lock on an item at any given time.

If a transaction A holds a shared lock on item X, then a request from another transaction B for an exclusive lock on X will cause B to go into a wait state (and B will wait until A's lock is released). A request from transaction B for a shared lock on X will be granted (that is, B will now also hold a shared lock on X).

If transaction A holds an exclusive lock on record X, then a request from transaction B for a lock of either type on X will cause B to go into a wait state (and B will wait until A's lock is released).

This can be summarised by means of the compatibility matrix below, that shows which type of lock requests can be granted simultaneously:

Type of lock on X that transaction B requests	Type of lock that transaction A hold on data item X		
	X	S	No lock
X	No	No	Yes
S	No	Yes	Yes
No lock	Yes	Yes	Yes

For example, when transaction A holds an exclusive (X) lock on data item X, the request from transaction B for an exclusive lock on X will not be granted. If transaction A holds a shared (S) lock on data item X, the request from transaction B for a shared lock will be granted (two transactions can read access the same item simultaneously) but not for an exclusive lock.

Transaction requests for record locks are normally implicit (at least in most modern systems). In addition, a user can specify explicit locks. When a transaction successfully retrieves a record, it automatically acquires an S lock on that record. When a transaction successfully updates a record, it automatically acquires an X lock on that record. If a transaction already holds an S lock on a record, then the update operation will promote the S lock to X level as long as T is the only transaction with an S lock on X at the time.

Exclusive and shared locks are normally held until the next synchronisation point (review the concept of synchronisation point under 'Atomicity'). However, a transaction can explicitly release locks that it holds prior to termination using the unlock command.

Use of the locking scheme

Using binary locks or multiple-mode locks in transactions as described earlier does not guarantee serialisability of schedules in which the transactions participate. For example, two simple transactions T1 and T2 are shown below:

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read-item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Assume, initially, that X = 20 and Y = 30; the result of serial schedule T1 followed by T2 is X = 50 and Y = 80; and the result of serial schedule T2 followed by T1 is X = 70 and Y = 50. The figure below shows an example where, although the multiple-mode locks are used, a non-serialisable schedule may still result:

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read-item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

The reason this non-serialisable schedule occurs is that the items Y in T1 and X in T2 were unlocked too early. To guarantee serialisability, we must follow an additional protocol concerning the positioning of locking and unlocking operations in every transaction. The best known protocol, two-phase locking, is described below.

Guaranteeing serialisability by two-phase locking (2PL)

Basic 2PL

A transaction is said to follow the two-phase locking protocol (basic 2PL protocol) if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases: an expanding (or growing) phase, during which new locks on items can be acquired but none can be released; and a shrinking phase, during which existing locks can be released but no new locks can be acquired.

Transactions T1 and T2 shown in the last section do not follow the 2PL protocol.

This is because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2. If we enforce 2PL, the transactions can be rewritten as T1' and T2', as shown below:

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read-item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Now the schedule involving interleaved operations shown in the figure above is not permitted. This is because T1' will issue its write_lock(X) before it unlocks Y; consequently, when T2' issues its read_lock(X), it is forced to wait until T1' issues its unlock(X) in the schedule.

It can be proved that, if every transaction in a schedule follows the basic 2PL, the schedule is guaranteed to be serialisable, removing the need to test for serialisability of schedules any more. The locking mechanism, by enforcing 2PL rules, also enforces serialisability.

Another problem that may be introduced by 2PL protocol is deadlock. The formal definition of deadlock will be discussed below. Here, an example is used to give you an intuitive idea about the deadlock situation. The two transactions that follow the 2PL protocol can be interleaved as shown here:

Time	T_1'	T_2'
1	read_lock(Y);	
2	read_item(Y);	
3		read_lock(X);
4		read-item(X);
5	write_lock(X);	
	wait	
6		write_lock(Y);
		wait
	...	...

At time step 5, it is not possible for T1' to acquire an exclusive lock on X as there is already a shared lock on X held by T2'. Therefore, T1' has to wait. Transaction T2' at time step 6 tries to get an exclusive lock on Y, but it is unable to as T1' has a shared lock on Y already. T2' is put in waiting too. Therefore, both transactions wait fruitlessly for the other to release a lock. This situation is known as a deadly embrace or deadlock. The above schedule would terminate in a deadlock.

Conservative 2PL

A variation of the basic 2PL is conservative 2PL also known as static 2PL, which is a way of avoiding deadlock. The conservative 2PL requires a transaction to lock all the data items it needs in advance. If at least one of the required data items cannot be obtained then none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Although conservative 2PL is a deadlock-free protocol, this solution further limits concurrency.

Strict 2PL

In practice, the most popular variation of 2PL is strict 2PL, which guarantees a strict schedule. (Strict schedules are those in which transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted). In strict 2PL, a transaction T does not release any of its locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Notice the difference between conservative and strict 2PL; the

former must lock all items before it starts, whereas the latter does not unlock any of its items until after it terminates (by committing or aborting). Strict 2PL is not deadlock-free unless it is combined with conservative 2PL.

In summary, all type 2PL protocols guarantee serialisability (correctness) of a schedule but limit concurrency. The use of locks can also cause two additional problems: deadlock and livelock. Conservative 2PL is deadlock-free.

Exercise 4

Multiple-mode locking scheme and serialisability of schedules

1. For the example schedule shown again here below, complete the two possible serial schedules, and show the values of items X and Y in the two transactions and in the database at each time step.

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read-item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

2. Discuss why the schedule below is a non-serialisable schedule. What went wrong with the multiple-mode locking scheme used in the example schedule?

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read-item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

Dealing with deadlock and livelock

Deadlock occurs when each of two transactions is waiting for the other to release the lock on an item. A simple example was shown above, where the two transactions T1' and T2' are deadlocked in a partial schedule; T1' is waiting for T2' to release item X, while T2' is waiting for T1' to release item Y. Meanwhile, neither can proceed to unlock the item that the other is waiting for, and other transactions can access neither item X nor item Y. Deadlock is also possible when more than two transactions are involved.

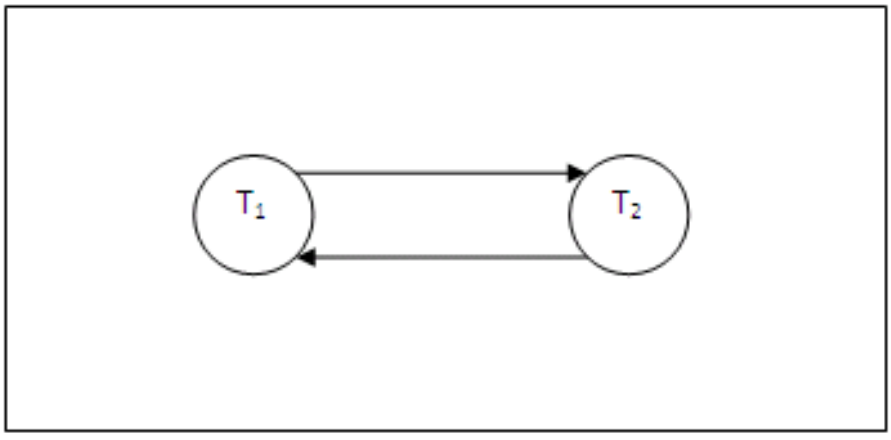
Deadlock detection with wait-for graph

A simple way to detect a state of deadlock is to construct a wait-for graph. One node is created in the graph for each transaction that is currently executing in the schedule. Whenever a transaction Ti is waiting to lock an item X that

is currently locked by a transaction Tj, it creates a directed edge (Ti # Tj). When Tj releases the lock(s) on the items that Ti was waiting for, the directed edge is dropped from the waiting-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. Recall this partial schedule introduced previously:

Time	T_1'	T_2'
1	read_lock(Y);	
2	read_item(Y);	
3		read_lock(X);
4		read-item(X);
5	write_lock(X);	
	wait	
6		write_lock(Y);
		wait
	...	...

The wait-for graph for the above partial schedule is shown below:



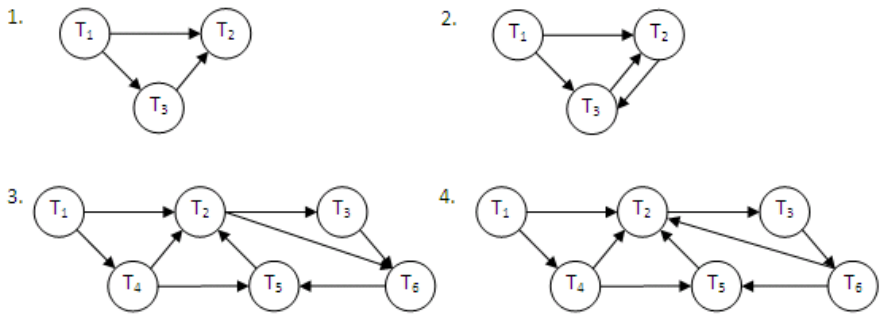
One problem with using the wait-for graph for deadlock detection is the matter of determining when the system should check for deadlock. Criteria such as the number of concurrently executing transactions or the period of time several transactions have been waiting to lock items may be used to determine that the system should check for deadlock.

When we have a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transaction to abort is known as victim selection. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and should try instead to select transactions that have not made many changes or that are involved in more than one deadlock cycle in the wait-for graph. A problem known as cyclic restart may occur, where a transaction is aborted and restarted only to be involved in another deadlock. The victim selection algorithm can use higher priorities for transactions that have been aborted multiple times, so that they are not selected as victims repeatedly.

Exercise 5

Wait-for graph

Given the graph below, identify the deadlock situations.



Ordering data items deadlock prevention protocol

One way to prevent deadlock is to use a deadlock prevention protocol. One such deadlock prevention protocol is used in conservative 2PL. It requires that every transaction lock all the items it needs in advance; if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. This solution obviously limits concurrency. A second protocol, which also limits concurrency, though to a lesser extent, involves ordering all the data items in the database and making sure that a transaction that needs several items will lock them according to that order (e.g. ascending order). For example, data items may be ordered as having rank 1, 2, 3, and so on.

A transaction T requiring data items A (with a rank of i) and B (with a rank of j, and j>i), must first request a lock for the data item with the lowest rank,

namely A. When it succeeds in getting the lock for A, only then can it request a lock for data item B.

All transactions must follow such a protocol, even though within the body of the transaction the data items are not required in the same order as the ranking of the data items for lock requests. For this particular protocol to work, all locks to be applied must be binary locks (i.e. the only locks that can be applied are write locks).

Wait-die or wound-wait deadlock prevention protocol

A number of deadlock prevention schemes have been proposed that make a decision on whether a transaction involved in a possible deadlock situation should be blocked and made to wait, should be aborted, or should preempt and abort another transaction. These protocols use the concept of transaction timestamp TS(T), which is a unique identifier assigned to each transaction. The timestamps are ordered based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2). Notice that the older transaction has a smaller timestamp value. This can be easily understood and remembered in the following way. For an older transaction T which is 'born' earlier, its birthday (i.e. TS(T) = 10am) is smaller than a younger transaction T', which is 'born' at 11am (i.e. TS(T') = 11am, and TS(T) < TS(T')).

Two schemes that use transaction timestamp to prevent deadlock are wait-die and wound-wait. Suppose that transaction Ti tries to lock an item X, but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rules followed by these schemes are as follows:

- **wait-die:** if TS(Ti) < TS(Tj) (Ti is older than Tj) then Ti is allowed to wait, otherwise abort Ti (Ti dies) and restart it later with the same timestamp.
- **wound-wait:** if TS(Ti) < TS(Tj) (Ti is older than Tj) then abort Tj (Ti wound Tj) and restart it later with the same timestamp, otherwise Ti is allowed to wait.

In wait-die, an older transaction is allowed to wait on a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-die approach does the opposite: a younger transaction is allowed to wait on an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it. Both schemes end up aborting the younger of the two transactions that may be involved in a deadlock, and it can be shown that these two techniques are deadlock-free. The two schemes can be summarised in the following two tables. The one below is for the wait-die protocol:

	Holding T younger	Holding T older
Requesting T older	Requesting T waits	
Requesting T younger		Requesting T aborted ad roll back

And this one below is for the wound-die protocol:

	Holding T younger	Holding T older
Requesting T older	Holding T aborted and roll back	
Requesting T younger		Requesting

The problem with these two schemes is that they cause some transactions to be aborted and restarted even though those transactions may never actually cause a deadlock. Another problem can occur with wait-die, where the transaction Ti may be aborted and restarted several times in a row because an older transaction Tj continues to hold the data item that Ti needs.

Exercise 6

Deadlock prevention protocols

A DBMS attempts to run the following schedule. Show:

1. How conservative 2PL would prevent deadlock.
2. How ordering all data items would prevent deadlock.
3. How the wait-for scheme would prevent deadlock.
4. How the wound-wait scheme would prevent deadlock.

operation	T_1	T_2	operation
1	read_lock(y);		
2	read_item(y);		
3	y := y+3		
		read_lock(y);	1
		read_item(y);	2
		y: = 1.2*y;	3
4	write_lock(y);		
5	write_item(y);		
6	read_lock(x);		
7	read_item(x);		
8	x := x+y;		
9	write_lock(x);		
10	write_item(x);		
11	unlock(x);		
12	unlock(y);		
		write_lock(y);	4
		write_item(y);	5
		read_lock(x);	6
		read_item(x);	7
		x:= x+y;	8
		write_lock(x);	9
		write_item(x);	10
		unlock(x);	11
		unlock(y);	12

Livelock

Another problem that may occur when we use locking is livelock. A transaction is in a state of livelock if it cannot proceed for an indefinite period while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. The standard solution for livelock is to have a fair waiting scheme. One such scheme uses a first-come-first-serve queue; transactions are enabled to lock an item in the order in which they originally requested to lock the item. Another

scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.

Review question 5

1. Complete the following table to describe which type of lock requests can be granted to the particular transaction.

Type of lock on X that transaction B requests	Type of lock that transaction A hold on data item X		
	X	S	No lock
X			
S			
No lock			

2. What is two-phase locking protocol? How does it guarantee serialisability?
3. Are the following statements true or false?

- If a transaction has a shared (read) lock on a database item, it can read the item but not update it.
- If a transaction has an exclusive (write) lock on a database item, it can update it but not read it.
- If a transaction has an exclusive (write) lock on a database item, it can both read and update it.
- If transaction A holds a shared (read) lock on a record R, another transaction B can issue a write lock on R.
- Basic 2PL, conservative 2PL and strict 2PL can all guarantee serialisability of a schedule as well as prevent deadlock.
- Strict 2PL guarantees strict schedules and is deadlock-free.
- Conservative 2PL is deadlock-free but limits the amount of concurrency.
- The wait-die and wound-wait schemes of deadlock prevention all end up aborting the younger of the two transactions that may be involved in a deadlock. They are both deadlock-free.

4. Complete the following tables of wait-die and wound-wait protocols for deadlock prevention.

	Holding T younger	Holding T older
Requesting T older		
Requesting T younger		

Wait-die protocol

	Holding T younger	Holding T older
Requesting T older		
Requesting T younger		

Wound-wait protocol

5. What is timestamp? Discuss how it is used in deadlock prevention protocols.

Discussion topics

Discussion topic 1

There are many new concepts in this chapter. If you want to discuss them with your colleagues or make comments about the concepts, use the online facilities.

Discussion topic 2

Compare the following pairs of concepts/techniques:

1. Interleaved vs simultaneous concurrency
2. Serial vs serialisable schedule
3. Shared vs exclusive lock
4. Basic vs conservative 2PL
5. Wait-for vs wound-wait deadlock prevention protocol
6. Deadlock vs livelock

Discussion topic 3

Analyse the relationships among the following terminology:

Problems of concurrency access to database (lost update, uncommitted dependency); serialisable schedule; basic 2PL; deadlock; conservative 2PL; wait-die and wound-wait

Additional content and exercises

Additional content

Concurrency control based on timestamp ordering

The idea of this scheme is to order the transactions based on their timestamps. (Recall the concept of timestamp.) A schedule in which the transactions participate is then serialisable, and the equivalent serial schedule has the transactions in order of their timestamp value. This is called timestamp ordering (TO). Notice the difference between this scheme and the two-phase locking. In two-phase locking, a schedule is serialisable by being equivalent to some serial schedule allowed by the locking protocols; in timestamp ordering, however, the schedule is equivalent to the particular serial order that corresponds to the order of the transaction timestamps. The algorithm must ensure that, for each item accessed by more than one transaction in the schedule, the order in which the item is accessed does not violate the serialisability of the schedule. To do this, the basic TO algorithm associates with each database item X two timestamp (TS) values:

- read_TS(X): The read timestamp of item X; this is the largest timestamp among all the timestamps of transactions that have successfully read item X.
- write_TS(X): The write timestamp of item X; this is the largest of all the timestamps of transactions that have successfully written item X.

Whenever some transaction T tries to issue a read_item(X) or write_item(X) operation, the basic TO algorithm compares the timestamps of T with the read and write timestamp of X to ensure that timestamp order of execution of the transactions is not violated. If the timestamp order is violated by the operation, then transaction T will violate the equivalent serial schedule, so T is aborted. Then T is resubmitted to the system as a new transaction with a new timestamp. If T is aborted and rolled back, any transaction T' that may have used a value written by T must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with the basic TO, since the schedule produced is not recoverable. The concurrency control algorithm must check whether the timestamp ordering of transactions is violated in the following two cases:

1. Transaction T issues a write_item(X) operation:

- If read_TS(X)>TS(T) or if write_TS(X)>TS(T), then abort and roll back T and reject the operation. This should be done because some transaction with a timestamp greater than TS(T) – and hence after T in the timestamp ordering – has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
- If the condition above does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2. Transaction T issues read_item(X) operation:

- If write_TS(X)>TS(T), then abort and roll back T and reject the operation. This should be done because some transaction with a timestamp greater than TS(T) – and hence after T in the timestamp ordering – has already written the value of item X before T had a chance to read X.
- If write_TS(X) <=TS(T), then execute the read_item(X) operation of T and set read_TS(T) to the larger of TS(T) and the current read_TS(X).

Multiversion concurrency control techniques

Multiversion concurrency control techniques keep the old values of a data item when the item is updated. Several versions (values) of an item are maintained. When a transaction requires access to an item, an appropriate version is chosen to maintain the serialisability of the concurrently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted, by reading an older version of the item to maintain serialisability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway – for example, for recovery purpose. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a temporal database, which keeps track of all changes and the items at which they occurred. In such cases, there is no additional penalty for multiversion techniques, since older versions are already maintained.

Multiversion techniques based on timestamp ordering

In this technique, several versions X1, X2, … Xk of each data item X are kept by the system. For each version, the value of version Xi and the following two timestamps are kept:

1. read_TS(Xi): The read timestamp of Xi; this is the largest of all the timestamps of transactions that have successfully read version Xi.
2. write_TS(Xi): The write timestamp of Xi; this is the timestamp of the transaction that wrote the value of version Xi.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version of item X, Xk+1, is created, with both the write_TS(Xk+1) and the read_TS(Xk+1) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version Xi, the value of read_TS(Xi) is set to the largest of read_TS(Xi) and TS(T).

To ensure serialisability, we use the following two rules to control the reading and writing of data items:

1. If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(Xi) of all versions of X which is also less than or equal to TS(T), and TS(T) < read_TS(Xi), then abort and roll back transaction T; otherwise, create a new version Xj of X with read_TS(Xj) = write_TS(Xj) = TS(T).
2. If transaction T issues a read_item(X) operation, and version i of X has the highest write_TS(Xi) of all versions of X which is also less than or equal to TS(T), then return the value of Xi to transaction T, and set the value of read_TS(Xj) to the largest of TS(T) and the current read_TS(Xj).

Multiversion two-phase locking

In this scheme, there are three locking modes for an item: read, write and certify. Hence, the state of an item X can be one of 'read locked', 'write locked', 'certify locked' and 'unlocked'. The idea behind the multiversion two-phase locking is to allow other transactions T' to read an item X while a single transaction T holds a write lock X. (Compare with standard locking scheme.) This is accomplished by allowing two versions for each item X; one version must always have been written by some committed transaction. The second version X' is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the committed version X while T holds the write lock. Now transaction T can change the value of X' as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a certify lock on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write lock items are released by any reading transactions. At this point, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown below:

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

In this multiversion two-phase locking scheme, reads can proceed concurrently with a write operation – an arrangement not permitted under the standard two-phase locking schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on all items it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are

only allowed to read the version X that was written by committed transaction. However, deadlock may occur.

Granularity of data items

All concurrency control techniques assumed that the database was formed of a number of items. A database item could be chosen to be one of the following:

- A database record.
- A field value of a database record.
- A disk block.
- A whole file.
- The whole database.

Several trade-offs must be considered in choosing the data item size. We shall discuss data item size in the context of locking, although similar arguments can be made for other concurrency control techniques.

First, the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item is a disk block, a transaction T that needs to lock a record A must lock the whole disk block X that contains A. This is because a lock is associated with the whole data item X. Now, if another transaction S wants to lock a different record B that happens to reside in the same block X in a conflicting disk mode, it is forced to wait until the first transaction releases the lock on block X. If the data item size was a single record, transaction S could proceed as it would be locking a different data item (record B).

On the other hand, the smaller the data item size is, the more items will exist in the database. Because every item is associated with a lock, the system will have a larger number of locks to be handled by the lock manger. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for the read_TS and write_TS for each data item, and the overhead of handling a large number of items is similar to that in the case of locking.

The size of data items is often called the data item granularity. Fine granularity refers to small item size, whereas coarse granularity refers to large item size. Given the above trade-offs, the obvious question to ask is: What is the best item size? The answer is that it depends on the types of transactions involved. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. On the other hand, if a transaction typically accesses many records of the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

Most concurrency control techniques have a uniform data item size. However, some techniques have been proposed that permit variable item sizes. In these techniques, the data item size may be changed to the granularity that best suits the transactions that are currently executing on the system.

Additional exercises

There are four suggested extension exercises for this chapter.

Extension exercise 1

Interleaved concurrency

T1, T2 and T3 are defined to perform the following operations: T1: Add one to A. T2: Double A. T3: Display A on the screen and then set A to one.

1. Supposed the above three transactions are allowed to execute concurrently. If A has an initial value zero, how many correct results are there? Enumerate them.
2. Suppose the internal structure of T1, T2, and T3 is as indicated below:

T1	T_2	T_3
F_1: fetch A into t_1 $t_1 := t_1+1$	F_2: fetch A into t_2 $t_2 := t_2*2$	F_3: fetch A into t_3 display t_3
U_1: update A from t_1	U_2: update A from t_2	U_3: update A from t_3

If the transactions execute without any locking, how many possible interleaved executions are there?

Extension exercise 2

Deadlock

The following list represents the sequence of events in an interleaved execution of a set of transaction T1 to T12. A, B, … H are data items in the database. Assume that FETCH R acquires an S lock on R, and UPDATE R promotes that lock to X level. Assume also all locks are held until the next synchronisation point. Are there any deadlocks at time tn?

Time	Transaction	Operation
t_0	T1	FETCH A
t_1	T2	FETCH B
	T1	FETCH C
...	T4	FETCH D
	T5	FETCH A
	T2	FETCH E
	T2	UPDATE E
	T3	FETCH F
	T2	FETCH F
	T5	UPDATE A
	T1	COMMIT
	T6	FETCH A
	T5	ROLLBACK
	T6	FETCH C
	T6	UPDATE C
	T7	FETCH G
	T8	FETCH H
	T9	FETCH G
	T9	UPDATE G
	T8	FETCH E
	T7	COMMI
	T9	FETCH H
	T3	FETCH G
	T10	FETCH A
	T9	UPDATE H
	T6	COMMIT
	T11	FETCH C
	T12	FETCH D
	T2	UPDATE F
	T11	UPDATE C
	T12	FETCH A
	T10	UPDATE A
	T12	UPDATE D
	T4	FETCH G
t_n	...	...

Extension exercise 3

Multiversion two-phase locking

What is a certify lock? Discuss multiversion two-phase locking for concurrency control.

Extension exercise 4

Granularity of data items

How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items?

Chapter 14. Backup and Recovery

Table of contents

Objectives

At the end of this chapter you should be able to:

- Describe a range of causes of database failure, and explain mechanisms available to deal with these.
- Understand a range of options available for the design of database backup procedures.

- Analyse the problems of data management in a concurrent environment.
- Be able to discuss the software mechanisms you would expect to be provided for recovery in a large, multi-user database environment.

Relationship to other chapters

The design of suitable backup and recovery procedures will usually be carried out by the database administrator (DBA) or DBA group, probably in conjunction with representatives of management and application developers. The way in which the databases in use provide for the concurrent processing of transactions, covered in the chapter Concurrency Control, will have an impact on the design of the backup and recovery procedures required.

Context

In the previous chapter, Concurrency Control, we discussed the different causes of failure such as transaction errors and system crashes. In this chapter we will introduce some of the techniques that can be used to recover from transaction failures.

We will first introduce some concepts that are used by recovery processes such as the system log, checkpoints and commit points. We then outline the recovery procedures. The process of rolling back (undoing) the effect of a transaction will be discussed in detail.

We will present recovery techniques based on deferred update, also known as the NO-UNDO/REDO technique, and immediate update, which is known as UNDO/REDO. We also discuss the technique known as shadowing or shadow paging, which can be categorised as a NO-UNDO/NO-REDO algorithm. Recovery in multidatabase transactions is briefly discussed in the chapter. Techniques for recovery from catastrophic failure are also discussed briefly.

Introduction

In parallel with this chapter, you should read Chapter 22 of Ramez Elmasri and Shamkant B. Navathe, " FUNDAMENTALS OF Database Systems", (7th edn.).

Database systems, like any other computer system, are subject to failures. Despite this, any organisation that depends upon a database must have that database available when it is required. Therefore, any DBMS intended for a serious business or organisational user must have adequate facilities for fast recovery after failure. In particular, whenever a transaction is submitted to a DBMS for execution, the system must ensure that either all the operations in the

transaction are completed successfully and their effect is recorded permanently in the database, or the transaction has no effect whatsoever on the database or on any other transactions.

The understanding of the methods available for recovering from such failures are therefore essential to any serious study of database. This chapter describes the methods available for recovering from a range of problems that can occur throughout the life of a database system. These mechanisms include automatic protection mechanisms built into the database software itself, and non-automatic actions available to people responsible for the running of the database system, both for the backing up of data and recovery for a variety of failure situations.

Recovery techniques are intertwined with the concurrency control mechanisms: certain recovery techniques are best used with specific concurrency control methods. Assume for the most part that we are dealing with large multi-user databases. Small systems typically provide little or no support for recovery; in these systems, recovery is regarded as a user problem.

A typical recovery problem

Data updates made by a DBMS are not automatically written to disk at each synchronisation point. Therefore there may be some delay between the commit and the actual disk writing (i.e. regarding the changes as permanent and worthy of being made to disk). If there is a system failure during this delay, the system must still be able to ensure that these updates reach the disk copy of the database. Conversely, data changes that may ultimately prove to be incorrect, made for example by a transaction that is later rolled back, can sometimes be written to disk. Ensuring that only the results of complete transactions are committed to disk is an important task, which if inadequately controlled by the DBMS may lead to problems, such as the generation of an inconsistent database. This particular problem can be clearly seen in the following example.

Suppose we want to enter a transaction into a customer order. The following actions must be taken:

START

1. *Change the customer record with the new order data*
2. *Change the salesperson record with the new order data*
3. *Insert a new order record into the database*

STOP

And the initial values are shown in the figure below:

CUSTOMER:

C-No	Order-No	Description	Cost
123	1,000	400 Tennis balls	£2,400

SALESPERSON:

Name	Total-Sales
Jones	£3,200

ORDERS:

Order-No
1000
2000
3000
4000
5000

If only operations 1 and 2 are successfully performed, this results in the values shown in the figure below:

CUSTOMER:

C-No	Order-No	Description	Cost
123	1,000	400 Tennis balls	£2,400
123	8,000	250 Cricket balls	£6,500

SALESPERSON:

Name	Total-Sales
Jones	£9,700

ORDERS:

Order-No
1000
2000
3000
4000
5000

This database state is clearly unacceptable as it does not accurately reflect reality; for example, a customer may receive an invoice for items never sent, or a salesman may make commission on items never received. It is better to treat the whole procedure (i.e. from START to STOP) as a complete transaction and not commit any changes to the database until STOP has been successfully reached.

Transaction logging

System log

The recovery manager overcomes many of the potential problems of transaction failure by a variety of techniques. Many of these are heavily dependent upon the existence of a special file known as a system log, or simply a log (also sometimes called a journal or audit trail). It contains information about the start and end of each transaction and any updates which occur in the transaction. The log keeps track of all transaction operations that affect the values of database items. This information may be needed to recover from transaction failure. The log is kept on disk (apart from the most recent log block that is in the process of being generated, this is stored in the main memory buffers). Thus, the majority of the log is not affected by failures, except for a disk failure or catastrophic failure. In addition, the log is periodically backed up to archival storage (e.g. tape) to guard against such catastrophic failures. The types of entries that are written to the log are described below. In these entries, T refers to a unique transaction identifier that is generated automatically by the system and used to uniquely label each transaction.

- **start_transaction(T):** This log entry records that transaction T starts the execution.
- **read_item(T, X):** This log entry records that transaction T reads the value of database item X.
- **write_item(T, X, old_value, new_value):** This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before image of X, and the new value is known as an after image of X.
- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T):** This records that transaction T has been aborted.
- **checkpoint:** This is an additional entry to the log. The purpose of this entry will be described in a later section.

Some protocols do not require that read operations be written to the system log, in which case, the overhead of recording operations in the log is reduced, since fewer operations – only write – are recorded in the log. In addition, some protocols require simpler write entries that do not include new_value.

Because the log contains a record of every write operation that changes the value of some database item, it is possible to undo the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values. We can also redo the

effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T to their new_values. Redoing the operations of a transaction may be required if all its updates are recorded in the log but a failure occurs before we can be sure that all the new_values have been written permanently in the actual database.

Committing transactions and force-writing

A transaction T reaches its commit point when all its operations that access the database have been executed successfully; that is, the transaction has reached the point at which it will not abort (terminate without completing). Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, we search back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process. Transactions that have written their commit(T) entry in the log must also have recorded all their write operations in the log - otherwise they would not be committed - so their effect on the database can be redone from the log entries.

Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process, because the contents of main memory may be lost. Hence, before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing of the log file, before committing a transaction. A commit does not necessarily involve writing the data items to disk; this depends on the recovery mechanism in use.

A commit is not necessarily required to initiate writing of the log file to disk. The log may sometimes be written back automatically when the log buffer is full. This happens irregularly, as usually one block of the log file is kept in main memory until it is filled with log entries and then written back to disk, rather than writing it to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same information.

Checkpoints

In the event of failure, most recovery managers initiate procedures that involve redoing or undoing operations contained within the log. Clearly, not all operations need to be redone or undone, as many transactions recorded on the log will have been successfully completed and the changes written permanently to disk. The problem for the recovery manager is to determine which operations

need to be considered and which can safely be ignored. This problem is usually overcome by writing another kind of entry in the log: the checkpoint entry.

The checkpoint is written into the log periodically and always involves the writing out to the database on disk the effect of all write operations of committed transactions. Hence, all transactions that have their commit(T) entries in the log before a checkpoint entry will not require their write operations to be redone in case of a system crash. The recovery manager of a DBMS must decide at what intervals to take a checkpoint; the intervals are usually decided on the basis of the time elapsed, or the number of committed transactions since the last checkpoint. Performing a checkpoint consists of the following operations:

- Suspending executions of transactions temporarily;
- Writing (force-writing) all modified database buffers of committed transactions out to disk;
- Writing a checkpoint record to the log; and
- Writing (force-writing) all log records in main memory out to disk.

A checkpoint record usually contains additional information, including a list of transactions active at the time of the checkpoint. Many recovery methods (including the deferred and immediate update methods) need this information when a transaction is rolled back, as all transactions active at the time of the checkpoint and any subsequent ones may need to be redone.

In addition to the log, further security of data is provided by generating backup copies of the database, held in a separate location to guard against destruction in the event of fire, flood, disk crash, etc.

Undoing

If a transaction crash does occur, then the recovery manager may undo transactions (that is, reverse the operations of a transaction on the database). This involves examining a transaction for the log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to old-value. Undoing a number of write_item operations from one or more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

Redoing

Redoing transactions is achieved by examining a transaction's log entry and for every write_item(T, x, old_value, new_value) entry, the value of item x in the database is set to new_value. Redoing a number of transactions from the log must proceed in the same order in which the operations were written in the log.

The redo operation is required to be idempotent; that is, executing it over and over is equivalent to executing it just once. In fact, the whole recovery process should be idempotent. This is so because, if the system were to fail during the recovery process, the next recovery attempt might redo certain write_item operations that had already been redone during the previous recovery process. The result of recovery from a crash during recovery should be the same as the result of recovering when there is no crash during recovery. Of course, repeating operations, as long as they are done in the correct way, should never leave the database in an inconsistent state, although as we have seen the repetitions may be unnecessary.

It is only necessary to redo the last update of x from the log during recovery, because the other updates would be overwritten by this last redo. The redo algorithm can be made more efficient by starting from the end of the log and working backwards towards the last checkpoint. Whenever an item is redone, it is added to a list of redone items. Before redo is applied to an item, the list is checked; if the item appears on the list, it is not redone, since its last value has already been recovered.

Activity 1 - Looking up glossary entries

In the Transaction Logging section of this chapter, the following terms have glossary entries:

- system log
- commit point
- checkpoint
- force-writing

1. In your own words, write a short definition for each of these terms.
2. Look up and make notes of the definition of each term in the module glossary.
3. Identify (and correct) any important conceptual differences between your definition and the glossary entry.

Review question 1

1. Unfortunately, transactions fail frequently, and they do so due to a variety of causes. Review the chapter on Concurrency Control, and discuss the different causes of the transaction failures.
2. What is meant by a system log? Discuss how a system log is needed in the recovery process.
3. Discuss the actions involved in writing a checkpoint entry.

4. Discuss how undo and redo operations are used in the recovery process.

Recovery outline

Recovery from transaction failures usually means that the database is restored to some state from the past, so that a correct state – close to the time of failure – can be reconstructed from that past state. To do this, the system must keep information about changes to data items during transaction execution outside the database. This information is typically kept in the system log. It is important to note that a transaction may fail at any point, e.g. when data is being written to a buffer or when a log is being written to disk. All recovery mechanisms must be able to cope with the unpredictable nature of transaction failure. Significantly, the recovery phase itself may fail; therefore, the recovery mechanism must also be capable of recovering from failure during recovery.

A typical strategy for recovery may be summarised based on the type of failures.

Recovery from catastrophic failures

The main technique used to handle catastrophic failures including disk crash is that of database backup. The whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. When the system log is backed up, users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up can have their effect on the database reconstructed. A new system log is started after each database backup operation. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been entered in the backed-up copy of the system log are reconstructed.

Recovery from non-catastrophic failures

When the database is not physically damaged but has become inconsistent due to non-catastrophic failure, the strategy is to reverse the changes that caused the inconsistency by undoing some operations. It may also be necessary to redo some operations that could have been lost during the recovery process, or for

some other reason, in order to restore a consistent state of the database. In this case, a complete archival copy of the database is not required; rather, it is sufficient that the entries kept in the system log are consulted during the recovery.

There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates. The deferred update techniques do not actually update the database until after a transaction reaches its commit point; then the updates are recorded in the database. Before commit, all transaction updates are recorded in the local transaction workspace. During commit, the updates are first recorded persistently in the log and then written to the database. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been written in the database. Hence, deferred update is also known as the NO-UNDO/REDO algorithm.

In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force-writing before they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both undo and redo are required during recovery, so it is known as the UNDO/REDO algorithm.

Transaction rollback

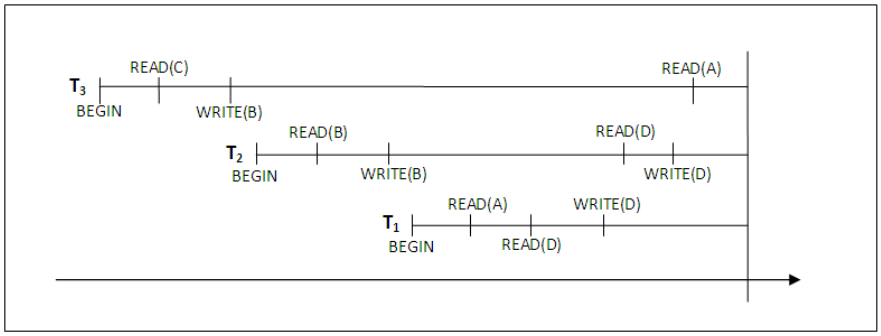
If a transaction fails for whatever reason after updating the database, it may be necessary to roll back or UNDO the transaction. Any data item values that have been changed by the transaction must be returned to their previous values. The log entries are used to recover the old values of data items that must be rolled back.

If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some item Y written by S must also be rolled back; and so on. This phenomenon is called cascading rollback. Cascading rollback, understandably, can be quite time-consuming. That is why most recovery mechanisms are designed such that cascading rollback is never required.

The table below shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown below:

T_1	T_2	T_3
Read_item(A);	read_item(B);	Read_item(C);
Read_item(D);	write_item(B);	Write_item(B);
Write_item(D);	read_item(D);	Read_item(A);
	write_item(D);	Write_item(A);

The diagram below graphically shows the operations of different transactions along the time axis:



The figure below shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of A, B, C and D, which are used by the transactions, are shown to the right of the system log entries. At the point of system crash, transaction T3 has not reached its conclusion and must be rolled back. The write operations of T3, marked by a single *, are the operations that are undone during transaction rollback.

Log	A	B	C	D
	30	15	40	20
start_transaction(T_3)				
read_item(T_3, C)				
*write_item(T_3, B, 15, 12)		12		
start_transaction(T_2)				
read_item(T_2, B)				
**write_item(T_2, B, 12, 18)		18		
start_transaction(T_1)				
read_item(T_1, A)				
read_item(T_1, D)				
write_item(T_1, D, 20, 25)				25
read_item(T_2, D)				
**write_item(T_2, D, 25, 26)				26
read_item(T_3, A)				
System crash				

* T_3 is rolled back because it did not reach its commit point when system crash happens.

** T_2 is rolled back because it reads the value of item B written by T_3.

The rest of the write entries in the log are redone.

We must now check for cascading rollback. In the diagram above, which shows

transactions along the time axis, we see that transaction T2 reads the value B which was written by T3; this can also be determined by examining the log. Because T3 is rolled back, T2 must also be rolled back. The write operations of T2, marked by ** in the log, are the ones that are undone. Note that only write operations need to be undone during transaction rollback; read operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary. If rollback of transactions is never required by the recovery method, we can keep more limited information in the system log. There is also no need to record any read_item operations in the log, because these are needed only for determining cascading rollback.

Review question 2

1. What is catastrophic failure? Discuss how databases can recover from catastrophic failures.

2. What is meant by transaction rollback? Why is it necessary to check for cascading rollback?

3. Compare deferred update with immediate update techniques by filling in the following blanks.

   The deferred update techniques do not actually update the database until a transaction reaches its commit point; then the updates are recorded in the database. Before commit, all transaction updates are recorded in the local transaction workspace. During commit, the updates are first recorded persistently in the _____ and then written to the _____. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so is not needed. It may be necessary to _____ the effect of the operations of a committed transaction in the log, because their effect may not yet have been written in the database. Hence, deferred update is also known as the _____ algorithm. In the immediate update techniques, the database may be updated by some operations of a transaction _____ the transaction reaches its commit point. However, these operations are typically recorded in the log on disk before they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be _____; that is, the transaction must be _____. In the general case of immediate update, both _____ and _____ are required during recovery, so it is known as the _____ algorithm.

Recovery techniques based on deferred update

Deferred update

The idea behind deferred update is to defer or postpone any actual updates to the database itself until the transaction completes its execution successfully and reaches its commit point. During transaction execution, the updates are recorded only in the log and in the transaction workspace. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database itself. If a transaction fails before reaching its commit point, there is no need to undo any operations, because the transaction has not affected the database in any way.

The steps involved in the deferred update protocol are as follows:

1. When a transaction starts, write an entry start_transaction(T) to the log.
2. When any operation is performed that will change values in the database, write a log entry write_item(T, x, old_value, new_value).
3. When a transaction is about to commit, write a log record of the form commit(T); write all log records to disk.
4. Commit the transaction, using the log to write the updates to the database; the writing of data to disk need not occur immediately.
5. If the transaction aborts, ignore the log records and do not write the changes to disk.

The database is never updated until after the transaction commits, and there is never a need to UNDO any operations. Hence this technique is known as the NO-UNDO/REDO algorithm. The REDO is needed in case the system fails after the transaction commits but before all its changes are recorded in the database. In this case, the transaction operations are redone from the log entries. The protocol and how different entries are affected can be best summarised as shown:

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, x)	No	N/A	N/A
write_item(T, x)	No	No	No
commit(T)	Yes	Yes	*Yes
checkpoint	Yes	Undefined	Yes(of committed Ts)

*Yes: writing back to disk may occur not immediately.

Deferred update in a single-user environment

We first discuss recovery based on deferred update in single-user systems, where no concurrency control is needed, so that we can understand the recovery process independently of any concurrency control method. In such an environment, the recovery algorithm can be rather simple. It works as follows.

Use two lists to maintain the transactions: the committed transactions list, which contains all the committed transactions since the last checkpoint, and the active transactions list (at most one transaction falls in this category, because the system is a single-user one). Apply the REDO operation to all the write_item operations of the committed transactions from the log in the order in which they were written to the log. Restart the active transactions.

The REDO procedure is defined as follows:

Redoing a write_item operation consists of examining its log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to new_value. The REDO operation is required to be idempotent, as discussed before.

Notice that the transaction in the active list will have no effect on the database because of the deferred update protocol, and is ignored completely by the recovery process. It is implicitly rolled back, because none of its operations were reflected in the database. However, the transaction must now be restarted, either automatically by the recovery process or manually by the user.

The method's main benefit is that any transaction operation need never be undone, as a transaction does not record its changes in the database until it reaches its commit point.

The protocol is summarised in the diagram below:

Action	Entry in log	
	start_transaction(T)	commit(T)
Re-submit	Yes	No
Redo	Yes	Yes

The diagram below shows an example of recovery in a single-user environment, where the first failure occurs during execution of transaction T2. The recovery process will redo the write_item(T1, D, 20) entry in the log by resetting the value of item D to 20 (its new value). The write(T2, …) entries in the log are ignored by the recovery process because T2 is not committed. If a second failure occurs during recovery from the first failure, the same recovery process is repeated from start to finish, with identical results.

start_transaction(T1)	
write_item(T1, D, 20)	
commit(T1)	
start_transaction(T2)	
write_item(T2, B, 10)	
write_item(T2, D 25)	← System crash

T_1	T_2
read_item(A)	read_item(B)
read_item(D)	write_item(B)
write_item(D)	read_item(D)
	write_item(D)

Deferred update in a multi-user environment

For a multi-user system with concurrency control, the recovery process may be more complex, depending on the protocols used for concurrency control. In many cases, the concurrency control and recovery processes are interrelated. In general, the greater the degree of concurrency we wish to achieve, the more difficult the task of recovery becomes.

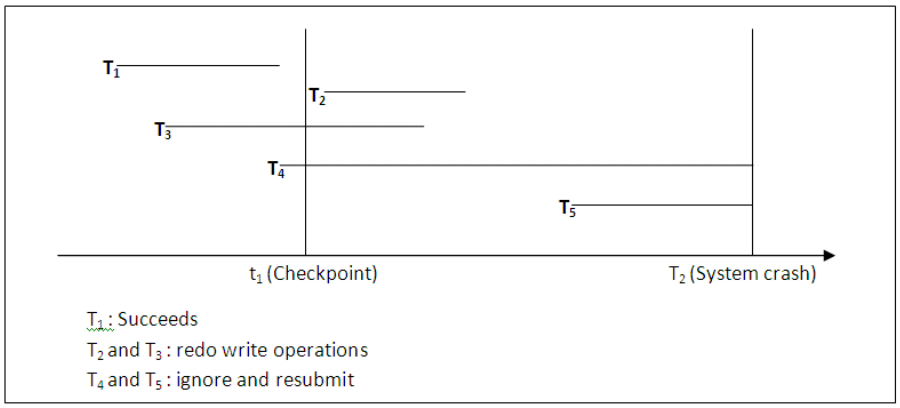
Consider a system in which concurrency control uses two-phase locking (basic 2PL) and prevents deadlock by pre-assigning all locks to items needed by a transaction before the transaction starts execution. To combine the deferred update methods for recovery with this concurrency control technique, we can keep all the locks on items in effect until the transaction reaches its commit point. After that, the locks can be released. This ensures strict and serialisable schedules. Assuming that checkpoint entries are included in the log, a possible

recovery algorithm for this case is given below.

Use two lists of transactions maintained by the system: the committed transactions list which contains all committed transactions since the last checkpoint, and the active transactions list. REDO all the write operations of the committed transactions from the log, in the order in which they were written into the log. The transactions in the active list that are active and did not commit are effectively cancelled and must be resubmitted.

The REDO procedure is the same as defined earlier in the deferred update in the single-user environment.

The diagram below shows an example schedule of executing transactions. When the checkpoint was taken at time t1, transaction T1 had committed, whereas transaction T3 and T4 had not. Before the system crash at time t2, T3 and T2 were committed but not T4 and T5. According to the deferred update method, there is no need to redo the write operations of transaction T1 or any transactions committed before the last checkpoint time t1. Write operations of T2 and T3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transaction T4 and T5 are ignored: they are effectively cancelled and rolled back because none of their write operations were recorded in the database under the deferred update protocol.



Transaction actions that do not affect the database

In general, a transaction will have actions that do not affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. Hence, such

reports should be generated only after the transaction reaches its commit point. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs. The batch jobs are executed only after the transaction reaches its commit point. If the transaction does not reach its commit point because of a failure, the batch jobs are cancelled.

Exercise 1: Deferred update protocol

Given the operations of the four concurrent transactions in (1) below and the system log at the point of system crash in (2), discuss how each transaction recovers from the failure using deferred update technique.

1. The read and write operations of four transactions:

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)

2. System log at the point of crash:

start_transaction(T_1)	
write_item(T_1, D, 20)	
commit(T_1)	
checkpoint	
start_transaction(T_4)	
write_item(T_4, B, 15)	
commit(T_4)	
start_transaction(T_2)	
write_item(T_2, B, 12)	
start_transaction(T_3)	
write_item(T_3, A, 30)	
write_item(T_2, D, 25)	← System crash

Exercise 2: Recovery management using deferred update with incremental log

Below, a schedule is given for five transactions A, B, C, D and E.

Assume the initial values for the variables are a=1, b=2, c=3, d=4 and e=5.

Using an incremental log with deferred updates, for each operation in each of the transactions, show:

1. The log entries.
2. Whether the log is written to disk.
3. Whether the output buffer is updated.
4. Whether the DBMS on disk is updated.
5. The values of the variables on the disk.

Discuss how each transaction recovers from the failure.

A	B	C	D	E	Log entries	Log to disc	Buffer changed	DBMS on disk changed	a	b	c	d	e
									1	2	3	4	5
start A	start A												
read value a	read value a												
a := a * 2	a := a * 2												
write value a	write value a												
commit A	commit A												
		start B											
		read value b											
		b := b * 2											
		write value b											
			start C										
			read value c										
			c := c * 2										
			write value c										
						Check point							
		b := 0											
		write value b											
		commit B											
				start D									
				read value d									
				d := d * 2									
				write value d									
				commit D									
					start E								
					read value e								
					e := e * 2								
					write value e								
						FAIL URE							

Review question 3

1. Use your own words to describe the deferred update method for recovery management in a multi-user environment. Complete the following table to show how the deferred update protocol affects the log on disk, database buffer and database on disk.

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
Start_transaction(T)			
Read_item(T, x)			
Write_item(T, x)			
Commit(T)			
Checkpoint			

2. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by the transaction?

Recovery techniques based on immediate update

Immediate update

In the immediate update techniques, the database may be updated by the operations of a transaction immediately, before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force-writing before they are applied to the database, so that recovery is possible.

When immediate update is allowed, provisions must be made for undoing the effect of update operations on the database, because a transaction can fail after it has applied some updates to the database itself. Hence recovery schemes based on immediate update must include the capability to roll back a transaction by undoing the effect of its write operations.

1. When a transaction starts, write an entry start_transaction(T) to the log;
2. When any operation is performed that will change values in the database, write a log entry write_item(T, x, old_value, new_value);
3. Write the log to disk;
4. Once the log record is written, write the update to the database buffers;
5. When convenient, write the database buffers to the disk;
6. When a transaction is about to commit, write a log record of the form commit(T);
7. Write the log to disk.

The protocol and how different entries are affected can be best summarised below:

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, x)	No	N/A	N/A
write_item(T, x)	Yes	Yes	*Yes
commit(T)	Yes	Undefined	Undefined
Checkpoint	Yes	Undefined	Yes(of committed Ts)

*Yes: writing back to disk may not occur immediately

In general, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction

are recorded in the database on disk before the transaction commits, there is never a need to redo any operations of committed transactions. Such an algorithm is called UNDO/NO-REDO. On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the UNDO/REDO method, the most general recovery algorithm. This is also the most complex technique. Recovery activities are summarised below:

Action	Entry in log	
	start_transaction(T)	commit(T)
Undo and resubmit T	Yes	No
Redo	Yes	Yes

Immediate update in a single-user environment

We first consider a single-user system so that we can examine the recovery process separately from concurrency control. If a failure occurs in a single-user system, the executing transaction at the time of failure may have recorded some changes in the database. The effect of all such operations must be undone as part of the recovery process. Hence, the recovery algorithm needs an UNDO procedure, described subsequently, to undo the effect of certain write operations that have been applied to the database following examination of their system log entry. The recovery algorithm also uses the redo procedure defined earlier. Recovery takes place in the following way.

Use two lists of transaction maintained by the system: the committed transactions since the last checkpoint, and the active transactions (at most one transaction will fall in this category, because the system is single user). Undo all the write operations of the active transaction from the log, using the UNDO procedure described hereafter. Redo all the write operations of the committed transactions from the log, in the order in which they were written in the log, using the REDO procedure.

The UNDO procedure is defined as follows:

Undoing a write operation consists of examining its log entry write_item(T, x, old_value, new_value) and setting the value of x in the database to old_value. Undoing a number of such write operations from one or more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

Immediate update in a multi-user environment

When concurrency execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure below outlines a recovery technique for concurrent transactions with immediate update. Assume that the log includes checkpoints and that the concurrency control protocol produces strict schedules – as, for example, the strict 2PL protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that last wrote the item has committed. However, deadlocks can occur in strict 2PL, thus requiring UNDO of transactions.

Use two lists of transaction maintained by the system: the committed transactions since the last checkpoint, and the active transactions. Undo all the write operations of the active (uncommitted) transaction from the log, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log. Redo all the write operations of the committed transactions from the log, in the order in which they were written in the log, using the REDO procedure.

Exercise 3: Immediate update protocol

Given the same transactions and system log in exercise 1, discuss how each transaction recovers from the failure using immediate update technique.

Exercise 4: Recovery management using immediate update with incremental log

The same schedule and initial values of the variables in exercise 2 are given; use the immediate update protocol for recovery to show how each transaction recovers from the failure.

Review question 4

1. Use your own words to describe the immediate update method for recovery management in a multi-user environment. Complete the following table to show how the immediate update protocol affects the log on disk, database buffer and database on disk.

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
Start_transaction(T)			
Read_item(T, x)			
Write_item(T, x)			
Commit(T)			
Checkpoint			

2. In general, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transac-

tion are recorded in the database on disk before the transaction commits, there is never a need to redo any operations of committed transactions. Such an algorithm is called UNDO/NO-REDO. On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the UNDO/REDO method.

Recovery in multidatabase transactions

So far, we have implicitly assumed that a transaction accesses a single database. In some cases a single transaction, called a multidatabase transaction, may require access to multiple database. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be Relational, whereas others are hierarchical or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction will have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat similar to the case of a distributed database management system, where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of multidatabase transaction, it is necessary to have a two-level recovery mechanism. A global recovery manager, or coordinator, is needed in addition to the local recovery managers. The coordinator usually follows a protocol called the two-phase commit protocol, whose two phases can be stated as follows.

PHASE 1: When all participating databases signal the coordinator that the part of the multidatabase transaction involving them has concluded, the coordinator sends a message "prepare for commit" to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records to disk and then send a "ready to commit" or "OK" signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a "cannot commit" or "not OK" signal to the coordinator. If the coordinator does not receive a reply from a database within a certain time interval, it assumes a "not OK" response.

PHASE 2: If all participating databases reply "OK", the transaction is successful, and the coordinator sends a "commit" signal for the transaction to the participating databases. Because all the local effects of the transaction have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a commit(T) entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases have a "not OK" response to the coordinator, the transaction has failed, and the coordinator sends a message to "roll back" or UNDO the local effect of the transaction to each participating database. This is done

by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants – or the coordinator – fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before phase 1 usually requires the transaction to be rolled back, whereas a failure during phase 2 means that a successful transaction can recover and commit.

Review question 5

Describe the two-phase commit protocol for multidatabase transactions.

Additional content and exercise

Shadow paging

In the shadow page scheme, the database is not directly modified but a copy, stored on permanent storage (e.g. disk), is made of the portion of the database to be modified and all modifications are made to this copy. Meanwhile, the old version of the database remains intact. Once the transaction commits, the modified copy replaces the original in an atomic manner, i.e. the replacement is carried out in its entirety or not at all. If a system crashes at this point, the old version is still available for recovery.

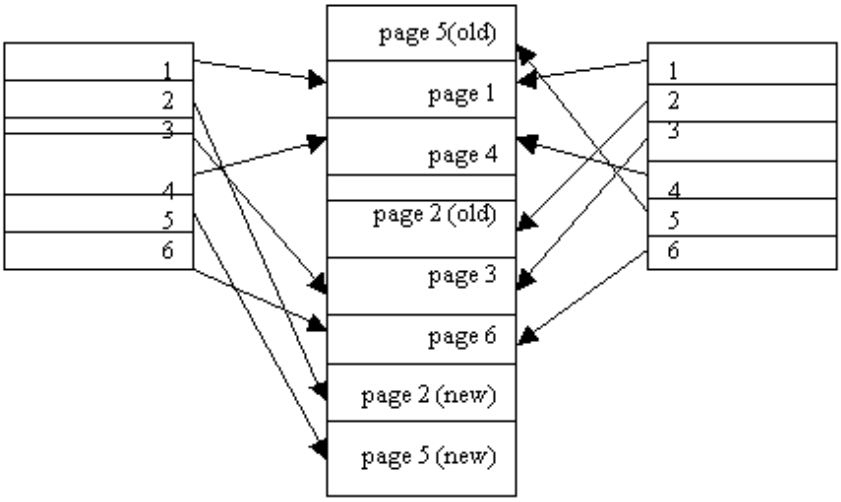
Page management

Before we discuss this scheme, a review of the paging scheme as used in the operating system for virtual memory management is appropriate. The memory that is accessed by a process (a program in execution is a process) is called virtual memory. Virtual memory is divided into pages that are all of a certain size (commonly 4096 bytes or 4K). The virtual or logical pages are mapped onto physical memory blocks (i.e. disk physical records) of the same size as the pages. The mapping is achieved by consulting a page table (or directory). The page table lists each logical page identifier and the address of the physical blocks that actually stores the logical page. The advantage of this scheme is that the consecutive logical pages need not be mapped onto consecutive physical blocks.

Shadow paging scheme considers the database to be made up of a number of fixed-size disk pages (or disk blocks) – say, n – for recovery purposes. A page table (or directory) with n entries is constructed, where the ith page table entry points to the ith database page on disk. The page table is kept in main memory if it is not too large, and all references – reads or writes – to database pages on disk go through the page table.

Shadow paging scheme in a single-user environment

In the shadow page scheme, two page tables are used. The original page table (shadow page table) and the current page table. Initially, both page tables point to the same blocks of physical storage. The current page table is the only route through which a transaction can gain access to the data stored on disk. That is, a transaction always uses the current page table to retrieve the appropriate database blocks.



During transaction execution, the shadow page table is never modified. When a write operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere – on some previously unused disk block. The current page table entry is modified to point to the new disk block, whereas the shadow page table is not modified and continues to point to the old unmodified disk block. The diagram above illustrates the concepts of a shadow page table and a current page table. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow page table and the new version by the current page table.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current page table. The state of the database before transaction execution is available through the shadow page table, and that state is recovered by reinstating the shadow page table so that it becomes the current page table once more. The database thus is returned to its state prior to the transaction that was executing when the crash occurred,

and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow page table and freeing old page tables that it references. Since recovery involves neither undoing nor redoing data items, this technique is called the NO-UNDO/NO-REDO recovery technique.

The advantage of shadow paging is that it makes undoing the effect of the executing transaction very simple. There is no need to undo or redo any transaction operations. In a multi-user environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the page table (directory) is large, the overhead of writing shadow page tables to disk as transactions commit is significant. A further complication is how to handle garbage collection when a transaction commits. The old pages referenced by the shadow page that has been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits, and the current page table replaces the shadow page table to become the valid page table.

Extension exercise 1: Shadow paging

What is a current page table and a shadow page table? Discuss the advantages and disadvantages of the shadow paging recovery scheme.

Chapter 15. Distributed Database Systems

Table of contents

Objectives

At the end of this chapter you should be able to:

- Understand what is meant by client-server database systems, and describe variations of the client-server approach.
- Describe the essential characteristics of distributed database systems.
- Distinguish between client-server databases and distributed databases.
- Describe mechanisms to support distributed transaction processing.

- Compare strategies for performing updates in distributed database systems.
- Describe the reference architecture of distributed DBMSs.

Introduction

In parallel with this chapter, you should read Chapter 22 and Chapter 23 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

Distributed databases have become an integral part of business computing in the past years. The ability to maintain the integrity of data and provide accurate and timely processing of database queries and updates across multiple sites has been an important factor in enabling businesses to utilise data in a range of different locations, sometimes on a global scale. Standardisation of query languages, and of the Relational and Object models, has assisted the integration of different database systems to form networks of integrated data services. The difficulties of ensuring the integrity of data, that updates are timely, and that users receive a uniform rate of response no matter where on the network they are situated, remain, in many circumstances, major challenges to database vendors and users. In this chapter we shall introduce the topic of distributed database systems. We shall examine a range of approaches to distributing data across networks, and examine a range of strategies for ensuring the integrity and timeliness of the data concerned. We shall look at mechanisms for enabling transactions to be performed across different machines, and the various update strategies that can be applied when data is distributed across different sites.

Context

Many of the issues considered in other chapters of this module require a degree of further consideration when translated into a distributed context. When it becomes a requirement to distribute data across a network, the processes of transaction processing, concurrency control, recovery, security and integrity control and update propagation become significantly more involved. In this chapter we shall introduce a number of extensions to mechanisms which we have previously considered for non-distributed systems.

Client-server databases

For years, serious business databases were monolithic systems running only on one large machine, accessed by dumb terminals. In the late 1980s, databases evolved so that an application, called a 'client', on one machine could run against a database, called a 'server', on another machine. At first, this client-server

database architecture was only used on mini and mainframe computers. Between 1987 and 1988, vendors like Oracle Corp and Gupta first moved the client function and then the database server down to microcomputers and local area networks (LANs).

Today, the client-server concept has evolved to cover a range of approaches to distributing the processing of an application in a variety of ways between different machines.

An example of the client-server approach is the SQLserver system, from Microsoft. The SQLserver system is run on a server machine, which is usually a fairly powerful PC. A client program is run, usually on a separate machine, and makes requests for data to SQLserver via a local area network (LAN). The application program would typically be written in a language such as Visual Basic or Java. This approach allows multiple client machines on the network to request the same records from the database on the server. SQLserver will ensure that only one user at a time modifies any specific record.

The 2-tier model

Client-server architecture involves multiple computers connected in a network. Some of the computers (clients) process application programs and some computers (servers) perform database processing.

This approach is known as the 2-tier model of client-server computing, as it is made up of the two types of component, clients and servers. It is also possible that a machine that acts as a server to some clients, may itself act as a client to another server. This arrangement also falls under the 2-tier model, as it still only comprises the two types of machine within the network.

The client

This is the front-end of the client-server system. It handles all aspects of the user interface — it is the front-end because the client presents the system to the user. It can also be used to provide PC-based application development tools used to enter, display, query and manipulate data on the central server, and to build applications. The client operating system is usually Windows, MACOS, Linux or Unix.

The server

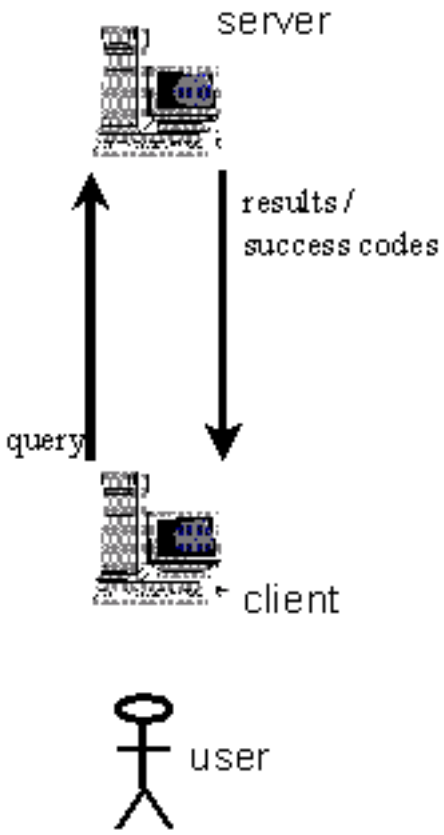
Servers perform functions such as database storage, integrity checking and data dictionary maintenance, and provide concurrent access control. Moreover, they also perform recovery and optimise query processing. The server controls access to the data by enforcing locking rules to ensure data integrity during transactions. The server can be a PC, mini or mainframe computer, and usually

employs a multi-tasking operating system such as Ubuntu Server and Windows Server OS.

Query processing in 2-tier client-server systems

Typically, in a client-server environment, the user will interact with the client machine through a series of menus, forms and other interface components. Supposing the user completes a form to issue a query against a customer database. This query may be transformed into an SQL SELECT statement by code running on the client machine. The client will then transmit the SQL query over the network to the server. The server receives the command, verifies the syntax, checks the existence and availability of the referenced objects, verifies that the user has SELECT privileges, and finally executes the query. The resulting data is formatted and sent to the application, along with return codes (used to identify whether the query was successful, or if not, which error occurred). On receipt of the data, the client might carry out further formatting - for example, creating a graph of the data - before displaying it to the user.

The following diagram illustrates how a user interacts with a client system, which transmits queries to the database server. The server processes the queries and returns the results of the query, or a code indicating failure of the query for some reason.



Advantages of the client-server approach

- Centralized storage: Users do not have to retain copies of corporate data on their own PCs, which would become quickly out of date. They can be assured that they are always working with the current data, which is stored on the server machine.
- Improved performance: Processing can be carried out on the machine most appropriate to the task. Data-intensive processes can be carried out on the server, whereas data-entry validation and presentation logic can be executed on the client machine. This reduces unnecessary network traffic and improves overall performance. It also gives the possibility of optimising the hardware on each machine to the particular tasks required of that machine.
- Scalability: If the number of users of an application grows, extra client

machines can be added (up to a limit determined by the capacity of the network or server) without significant changes to the server.

Disadvantages of the client-server approach

- Complexity: Operating database systems over a LAN or wide area network (WAN) brings extra complexities of developing and maintaining the network. The interfaces between the programs running on the client and server machines must be well understood. This usually becomes increasingly complex when the applications and/or DBMS software come from different vendors.
- Security: A major consideration, as preventative measures must be in place to protect against data theft or corruption on the client and server machines, and during transmission over the network.

Review question 1

- Explain what is meant by the 2-tier model of client-server computing.
- Why is it sometimes said that client-server computing improves the scalability of applications?
- What additional security issues are involved in the use of a client-server system, compared with a traditional mainframe database accessed via dumb terminals?

Variants of the 2-tier model

Business (application) logic

The relative workload on the client and server machines is greatly affected by the way in which the application code is distributed. The more application logic that is placed on client machines, the more of the work these machines will have to do. Furthermore, more data will need to be transmitted from servers to client machines because the servers do not contain the application logic that might have been used to eliminate some of the data prior to transmission. We can avoid this by transferring some of the application logic to the server. This helps to reduce the load on both the clients and the network. This is known as the split logic model of client-server computing. We can take this a stage further, and leave the client with only the logic needed to handle the presentation of data to users, placing all of the functional logic on servers; this is known as the remote presentation model.

Business logic implemented as stored procedures

The main mechanism for placing business logic on a server is known as 'stored procedures'. Stored procedures are collections of code that usually include SQL

for accessing the database. Stored procedures are invoked from client programs, and use parameters to pass data between the procedure and invoking program. If we choose to place all the business logic in stored procedures on the server, we reduce network traffic, as intermediate SQL results do not have to be returned to client machines. Stored procedures are compiled, in contrast with uncompiled SQL sent from the client. A further performance gain is that stored procedures are usually cached, therefore subsequent calls to them do not require additional disk access.

2-tier client-server architectures in which a significant amount of application logic resides on client machines suffer from the following further disadvantages:

- Upgrades and bug fixes must be made on every client machine. This problem is compounded by the fact that the client machines may vary in type and configuration.
- The procedural languages used commonly to implement stored procedures are not portable between machines. Therefore, if we have servers of different types, for example Oracle and Microsoft, the stored procedures will have to be coded differently on the different server machines. In addition, the programming environments for these languages do not provide comprehensive language support as is found in normal programming languages such as C++ or Java, and the testing/debugging facilities are limited.

Though the use of stored procedures improves performance (by reducing the load on the clients and network), taking this too far will limit the number of users that can be accommodated by the server — i.e. there will reach a point at which the server becomes a bottleneck because it is overloaded with the processing of application transactions.

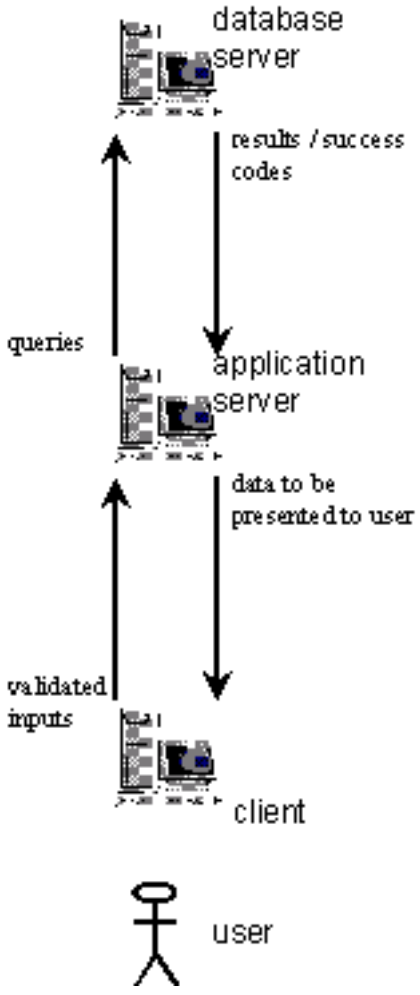
The 3-tier architecture

The 3-tier architecture introduces an applications server, which is a computer that fits in the middle between the clients and server. With this configuration, the client machines are freed up to focus on the validation of data entered by users, and the formatting and presentation of results. The server is also freed to concentrate on data-intensive operations, i.e. the retrieval and update of data, query optimisation, processing of declarative integrity constraints and database triggers, etc. The new middle tier is dedicated to the efficient execution of the application/business logic.

The application server can perform transaction management and, if required, ensure distributed database integrity. It centralises the application logic for easier administration and change.

The 3-tier model of client-server computing is best suited to larger installations. This is true because smaller installations simply do not have the volume of transactions to warrant an intermediate machine dedicated to application logic.

The following diagram illustrates how a user interacts with a client system, which deals with the validation of user input and the presentation of query results. The client system communicates with a middle tier system, the applications server, which formulates queries from the validated input data and sends queries to the database server. The database server processes the queries and sends to the applications server the results of the query, or a code indicating failure of the query for some reason. The application server then processes these results (or code) and sends data to the client system to be formatted and presented to the user.



Activity 1 – Investigating stored procedures

Read the documentation of the DBMS of your choice and investigate stored procedures as implemented in the environment. A visit to the software's website

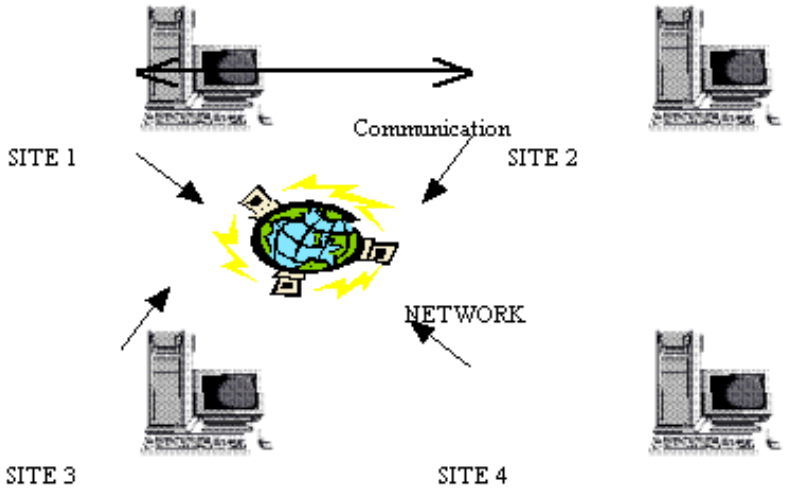
maybe also be useful. Identify the types of situations in which stored procedures are used, and by looking at a number of examples of stored procedures, develop an overall understanding for the structure of procedures and how they are called from a PL/SQL program.

Distributed database systems

Background to distributed systems

A distributed database system consists of several machines, the database itself being stored on these machines, which communicate with one another usually via high-speed networks or telephone lines. It is not uncommon for these different machines to vary in both technical specification and function, depending upon their importance and position in the system as a whole. Note that the generally understood description of a distributed database system given here is rather different from the client-server systems we examined earlier in the chapter. In client-server systems, the data is not itself distributed; it is stored on a server machine, and accessed remotely from client machines. In a distributed database system, on the other hand, the data is itself distributed among a number of different machines. Decisions therefore need to be made about the way in which the data is to be distributed and also about how updates are to be propagated over multiple sites.

The distribution of data by an organisation throughout its various sites and departments, allows data to be stored where it was generated, or indeed is most used, while still being easily accessible from elsewhere. The general structure of a distributed system is shown below:



A distributed system

In effect, each separate site in the example is really a database system site in its own right, each location having its own databases, as well as its own DBMS and transaction management software. It is commonly assumed when discussion arises concerning distributed database systems, that the many databases involved are widely spread from each other in geographical terms. Importantly, it should be made clear that geographical distances will have little or no effect upon the overall operation of the distributed system; the same technical problems arise whether the databases are simply logically separated or if separated by a great distance.

Motivation for distributed database systems

So why have distributed databases become so desirable? There are a number of reasons that promote the use of a distributed database system; these can include such factors as:

- The sharing of data
- Local autonomy
- Data availability
- Improving performance
- Improving system reliability

Furthermore, it is likely that the organisation that has chosen to implement the system will itself be distributed. By this, we mean that there are almost always several departments and divisions within the company structure. An illustrative example is useful here in clarifying the benefits that can be gained by the use of distributed database systems.

Scenario banking system

Imagine a banking system that operates over a number of separate sites; for the sake of this example, let us consider two offices, one in Manchester and another in Birmingham. Account data for Manchester accounts is stored in Manchester, while Birmingham's account data is stored in Birmingham. We can now see that two major benefits are afforded by the use of this system: efficiency of processing is increased due to the data being stored where it is most frequently accessed, while an increase in the accessibility of account data is also gained.

The use of distributed database systems is not without its drawbacks, however. The main disadvantage is the added complexity that is involved in ensuring that proper co-ordination between the various sites is possible. This increase in complexity can take a variety of forms:

- **Greater potential for bugs:** With a number of databases operating concurrently, ensuring that algorithms for the operation of the system are correct becomes an area of great difficulty. The potential is there for the existence of extremely subtle bugs.
- **Increased processing overhead:** The additional computation required in order to achieve inter-site co-ordination is a considerable overhead not present in centralised systems.

Date (1999) gives the 'fundamental principle' behind a truly distributed database:

"to the user; a distributed system should look exactly like a NONdistributed system.""

In order to accomplish this fundamental principle, twelve subsidiary rules have been established. These twelve objectives are listed below:

1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location independence
5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

These twelve objectives are not all independent of one another. Furthermore, they are not necessarily exhaustive and, moreover, they are not all equally significant.

Probably the major issue to be handled in distributed database systems is the way in which updates are propagated throughout the system. Two key concepts play a major role in this process:

- **Data fragmentation:** The splitting up of parts of the overall database across different sites.
- **Data replication:** The process of maintaining updates to data across different sites.

Fragmentation independence

A system can support data fragmentation if a given stored relation can be divided up into pieces, or 'fragments', for physical storage purposes. Fragmentation is desirable for performance reasons: data can be stored at the location where it is most frequently used, so that most operations are purely local and network traffic is reduced.

A fragment can be any arbitrary sub-relation that is derivable from the original relation via restriction (horizontal fragmentation – subset of columns) and projection (vertical fragmentation – subset of tuples) operations. Fragmentation independence (also known as fragmentation transparency), therefore, allows users to behave, at least from a logical standpoint, as if the data were not fragmented at all. This implies that users will be presented with a view of the data in which the fragments are logically combined together by suitable joins and unions. It is the responsibility of the system optimiser to determine which fragment needs to be physically accessed in order to satisfy any given user request.

Replication independence

A system supports data replication if a given stored relation - or, more generally, a given fragment - can be represented by many distinct copies or replicas, stored at many distinct sites.

Replication is desirable for at least two reasons. First, it can mean better performance (applications can operate on local copies instead of having to communicate with remote sites); second, it can also mean better availability (a given replicated object - fragment or whole relation - remains available for processing so long as at least one copy remains available, at least for retrieval purposes).

What problems are associated with data replication and fragmentation?

Both data replication and fragmentation have their related problems in implementation. However, distributed non-replicated data only has problems when the relations are fragmented.

The problem of supporting operations, such as updating, on fragmented relations has certain points in common with the problem of supporting operations on join and union views. It follows too that updating a given tuple might cause that tuple to migrate from one fragment to another, if the updated tuple no longer satisfies the relation predicate for the fragment it previously belonged to (Date, 1999).

Replication also has its associated problems. The major disadvantage of replication is that, when a given replicated object is updated, all copies of that object must be updated — the update propagation problem. Therefore, in addition to transaction, system and media failures that can occur in a centralised DBMS,

a distributed database system (DDBMS) must also deal with communication failures. Communication failures can result in a site that holds a copy of the object being unavailable at the time of the update.

Furthermore, the existence of both system and communication failures poses complications because it is not always possible to differentiate between the two (Ozsu and Valduriez, 1996).

Update strategies for replicated and non-replicated data

There are many update strategies for replicated and fragmented data. This section will explore these strategies and will illustrate them with examples from two of the major vendors.

Eager (synchronous) replication

Gray et al (1996) states that eager replication keeps all replicas exactly synchronised at all nodes (sites) by updating all the replicas as part of one atomic transaction. Eager replication gives serialisable execution, therefore there are no concurrency anomalies. But eager replication reduces update performance and increases transaction times, because extra updates and messages are added to the transaction. Eager replication typically uses a locking scheme to detect and regulate concurrent execution.

With eager replication, reads at connected nodes give current data. Reads at disconnected nodes may give stale (out-of-date) data. Simple eager replication systems prohibit updates if any node is disconnected. For high availability, eager replication systems allow updates among members of the quorum or cluster. When a node joins the quorum, the quorum sends the node all replica updates since the node was disconnected.

Eager replication and distributed reliability protocols

Distributed reliability protocols (DRPs) are implementation examples of eager replication. DRPs are synchronous in nature (ORACLE, 1993), and often use remote procedure calls (RPCs).

DRPs enforce atomicity (the all-or-nothing property) of transactions by implementing atomic commitment protocols such as the two-phase commit (2PC) (Gray, 1979). Although a 2PC is required in any environment in which a single transaction can interact with several autonomous resource managers, it is particularly important in a distributed system (Ozsu and Valduriez, 1996). 2PC extends the effects of local atomic commit actions to distributed transactions, by insisting that all sites involved in the execution of a distributed transaction

agree to commit the transaction before its effects are made permanent. Therefore, 2PC is an example of one copy equivalence, which asserts that the values of all physical copies of a logical data item should be identical when the transaction that updates it terminates.

The inverse of termination is recovery. Distributed recovery protocols deal with the problem of recovering the database at a failed site to a consistent state when that site recovers from failure (Ozsu and Valduriez, 1996). The 2PC protocol also incorporates recovery into its remit.

Exercise 1

What is meant by the terms 'atomic commitment protocol' and 'one copy equivalence'?

The two-phase commit (2PC) protocol

The 2PC protocol works in the following way (adapted from Date, 1999): COMMIT or ROLLBACK is handled by a system component called the Co-ordinator, whose task it is to guarantee that all resource managers commit or rollback the updates they are responsible for in unison - and furthermore, to provide that guarantee even if the system fails in the middle of the process.

Assume that the transaction has completed its database processing successfully, so that the system-wide operation it issues is COMMIT, not ROLLBACK. On receiving the COMMIT request, the Co-ordinator goes through the following two-phase process:

1. First, the Co-ordinator instructs all resource managers to get ready either to commit or rollback the current transaction. In practice, this means that each participant in the process must force-write all log entries for local resources used by the transaction out to its own physical log. Assuming the force-write is successful, the resource manager now replies 'OK' to the Co-ordinator, otherwise it replies 'Not OK'.

2. When the Co-ordinator has received replies from all participants, it force-writes an entry to its own physical log, recording its decision regarding the transaction. If all replies were 'OK', that decision is COMMIT; if any replies were 'Not OK', the decision is ROLLBACK. Either way, the Co-ordinator then informs each participant of its decision, or each participant must then COMMIT or ROLLBACK the transaction locally, as instructed.

If the system or network fails at some point during the overall process, the restart procedure will look for the decision record in the Co-ordinator's log. If it finds it, then the 2PC process can pick up where it left off. If it does not find it, then it assumes that the decision was ROLLBACK, and again the process can complete appropriately. However, in a distributed system, a failure on the part of the Co-ordinator might keep some participants waiting for the Co-ordinator's

decision. Therefore, as long as the participant is waiting, any updates made by the transaction via that participant are kept locked.

Review question 2

- Explain the role of an application server in a 3-tier client-server network.
- Distinguish between the terms 'fragmentation' and 'replication' in a distributed database environment.
- Describe the main advantage and disadvantage of eager replication.
- During the processing of the two-phase commit protocol, what does the Co-ordinator do if it is informed by a local resource manager process that it was unable to force-write all log entries for the local resources used by the transaction out to its own physical log?

Read-once / write-all protocol

A further replica-control protocol that enforces one-copy serialisability is known as read-once / write-all (ROWA) protocol. ROWA protocol is simple, but it requires that all copies of a logical data item be updated before the transaction can terminate.

Failure of one site may block a transaction, reducing database availability.

A number of alternative algorithms have been proposed that reduce the requirement that all copies of a logical data item be updated before the transaction can terminate. They relax ROWA by mapping each write to only a subset of the physical copies. One well-known approach is quorum-based voting, where copies are assigned votes, and read and write operations have to collect votes and achieve a quorum to read/write data.

Three-phase commit is a non-blocking protocol which prevents the 2PC blocking problem from occurring by removing the uncertainty for participants after their votes have been placed. This is done through the inclusion of a pre-commit phase that relays information to participants, advising them that a commit will occur in the near future.

Lazy or asynchronous replication

Eager replication update strategies, as identified above, are synchronous, in the sense that they require the atomic updating of some number of copies. Lazy group replication and lazy master replication both operate asynchronously.

If the users of distributed database systems are willing to pay the price of some inconsistency in exchange for the freedom to do asynchronous updates, they will insist that:

1. the degree of inconsistency be bounded precisely, and that

2. the system guarantees convergence to standard notions of 'correctness'.

Without such properties, the system in effect becomes partitioned as the replicas diverge more and more from one another (Davidson et al, 1985).

Lazy group replication

Lazy group replication allows any node to update any local data. When the transaction commits, a transaction is sent to every other node to apply the root transaction's updates to the replicas at the destination node. It is possible for two nodes to update the same object and race each other to install their updates at other nodes. The replication mechanism must detect this and reconcile the two transactions so that their updates are not lost (Gray et al, 1996).

Timestamps are commonly used to detect and reconcile lazy-group transactional updates. Each object carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old object timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica's timestamp and the update's old timestamp are equal. If so, the update is safe. The local replica's timestamp advances to the new transaction's timestamp and the object value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be 'dangerous'. In such cases, the node rejects the incoming transaction and submits it for reconciliation. The reconciliation process is then responsible for applying all waiting update transactions in their correct time sequence.

Transactions that would wait in an eager replication system face reconciliation in a lazy group replication system. Waits are much more frequent than deadlocks because it takes two waits to make a deadlock.

Lazy master replication

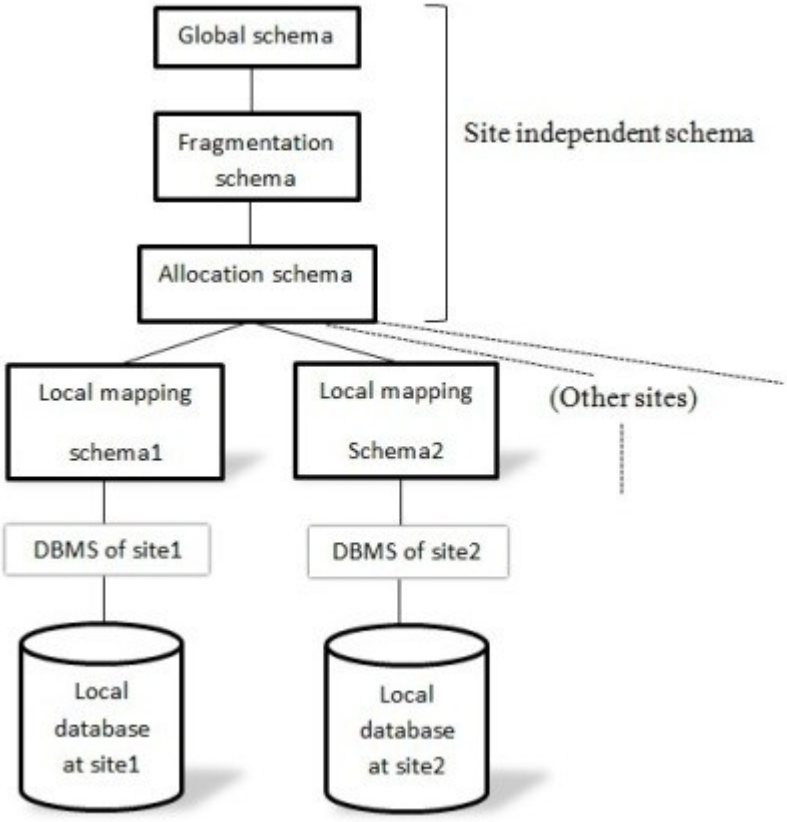
Another alternative to eager replication is lazy master replication. Gray et al (1996) states that this replication method assigns an owner to each object. The owner stores the object's correct value. Updates are first done by the owner and then propagated to other replicas. When a transaction wants to update an object, it sends a remote procedure call (RPC) to the node owning the object. To achieve serialisability, a read action should send read-lock RPCs to the masters of any objects it reads. Therefore, the node originating the transaction broadcasts the replica updates to all the slave replicas after the master transaction commits. The originating node sends one slave transaction to each slave node. Slave updates are timestamped to assure that all the replicas converge to the same final state. If the record timestamp is newer than a replica update timestamp, the update is 'stale' and can be ignored. Alternatively, each master node sends replica updates to slaves in sequential commit order.

Review question 3

When an asynchronous update strategy is being used, if two copies of a data item are stored at different sites, what mechanism can be used to combine the effect of two separate updates being applied to these different copies?

Reference architecture of a distributed DBMS

In chapter 1 we looked at the ANSI_SPARC three-level architecture of a DBMS. The architecture reference shows how different schemas of the DBMS can be organised. This architecture cannot be applied directly to distributed environments because of the diversity and complexity of distributed DBMSs. The diagram below shows how the schemas of a distributed database system can be organised. The diagram is adopted from Hirendra Sisodiya (2011).



Reference architecture for distributed database

1. **Global schema**

   The global schema contains two parts, a global external schema and a global conceptual schema. The global schema gives access to the entire system. It provides applications with access to the entire distributed database system, and logical description of the whole database as if it was not distributed.

2. **Fragmentation schema**

   The fragmentation schema gives the description of how the data is partitioned.

3. **Allocation schema**

   Gives a description of where the partitions are located.

4. **Local mapping**

   The local mapping contains the local conceptual and local internal schema. The local conceptual schema provides the description of the local data. The local internal schema gives the description of how the data is physically stored on the disk.

Review question 4

List the characteristics of applications that can benefit most from:

- synchronous replication
- asynchronous replication

Discussion topics

1. We have covered the client-server and true distributed database approaches in this chapter. Client-server systems distribute the processing, whereas distributed systems distribute both the processing and the data. Discuss the proposition that most commercial applications are adequately supported by a clientserver approach, and do not require the additional features of a truly distributed database system.

2. Discuss the proposition that, in those situations where a distributed database solution is required, most applications are adequately provided for by a lazy or asynchronous replication strategy, and do not require the sophistication of an eager or synchronous replication system. Discuss the implications for end users of synchronous and asynchronous updating.

Chapter 16. Object-Oriented Database Systems

Table of contents

Objectives

At the end of this chapter you should be able to:

- Describe the essential characteristics of Object databases.
- Critically assess the strengths and weaknesses of Object databases with respect to Relational systems.
- Describe why Object databases appear to be such a good fit for a number of major growth areas in computing, such as Web-based and multimedia

information systems.

- Describe the strategy being adopted by major database supplier Oracle to address the apparent threat of Object database systems, and critically compare this approach with a pure Object technology approach.

Introduction

In parallel with this chapter, you should read Chapter 25 and Chapter 26 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

The Object data model provides a richer set of semantics than the Relational model. Most of the major database vendors are extending the Relational model to include some of the mechanisms available in Object databases. These extended Relational databases are often called Object-Relational. In this sense the Object data model can be seen as an enriching of the Relational model, giving a wider range of modelling capabilities. The topics of design, concurrency control, performance tuning and distribution are just as relevant for Object databases as for Relational systems.

Relational database systems have been the mainstay of commercial systems since the 80s. Around about the same time, however, developments in programming languages were giving rise to a new approach to system development. These developments lead to the widespread use of Object technology, and in particular, Object-oriented programming languages such as C++ and Java. Many people expected a similar growth in the commercial use of Object database systems, but these have been relatively slow to be adopted in industry and commerce. In this chapter we will explore the reasons why Object databases have not so far had a major impact in the commercial arena, and examine whether the continuing growth of the World Wide Web and multimedia information systems could lead to a major expansion in the use of Object database technology.

Motivation

The Relational database model has many advantages that make it ideally suited to numerous business applications. Its ability to efficiently handle simple data types, its powerful and highly optimisable standard query language, and its good protection of data from programming errors make it an effective model. However, a number of limitations exist with the model, which have become increasingly clear as more developers and users of database systems seek to extend the application of DBMS technology beyond traditional transaction processing applications, such as order processing, financial applications, stock control, etc.

Among the applications that have proved difficult to support within Relational environments are those involving the storage and manipulation of design data.

Design data is often complex and variable in length, may be highly interrelated, and its actual structure, as well as its values, may evolve rapidly over time, though previous versions may be required to be maintained. This is quite different to the typically fixed-length, slowly evolving data structures which characterise transaction processing applications.

The query languages used to manipulate Relational databases are computationally incomplete; that is, they cannot be used to perform any arbitrary calculation that might be needed. The SQL language standard, and its derivative languages, are essentially limited to Relational Algebra-based operations, providing very little in the way of computational power to handle numerically complex applications.

Further to the problems that have been associated with Relational databases since their inception, a significant problem that has come to light relatively recently is the need to be able to store and manipulate ever more complicated data types, such as video, sound, complex documents, etc. This is putting an increasing strain on the model and restricting the kinds of business solutions that can be provided. One reason for this increase in data complexity is the explosion in popularity of the Internet and Web, where it is necessary to store large quantities of unstructured text, multimedia, images and spatial data.

Other examples of applications that have proved difficult to implement in Relational systems include:

- Geographical information systems
- Applications for processing large and inter-related documents
- Databases to support CASE tools
- Image processing applications

What is Object database technology?

Capturing semantics

Although the Relational model enforces referential integrity, as we saw in the chapter on integrity constraints and database triggers, it has no mechanism for distinguishing and enforcing the different kinds of relationship which may exist between entities. Examples of these relationships include:

- **Existence-dependency:** Where one entity can only exist in relation to another. An example of such a relationship is that of an order-line entity instance, which only makes sense within the context of its corresponding parent order entity.
- **Associations:** When entities of different types are associated with one another; for example, when a car entity is associated with a particular

person through an 'owns' relationship.

- **Categorisations:** When a number of different entity types are classified into a particular overall grouping; for example, lecturers, administrators, deans, professors and administrators are all categorised as university employees.

Such distinctions between relationship types can be made in a conceptual entity-relationship model, but not explicitly when mapped to the Relational model. If such distinctions are made, it is possible to define the semantics of operations to create, update and delete instances of relationships differently for each case.

Semantic data models are data models that attempt to capture more of the semantics of the application domain, and are frequently defined as extensions to the Relational model. Such models enable the representation of different types of entity, and the description of different types of relationship between entity types, such as those described above.

Semantic models therefore aim to support a higher level of 'understanding' of the data within the system; however, these models do not increase support for the manipulation of data. The extended data structuring mechanisms are accompanied by the same general set of operators (create entity, delete entity and update entity). We would be able to constrain the data structures more naturally if we recognised that the data structures that have been defined are accessed and updated through a fixed set of data-type specific operators. On creating a new entity it is often necessary to carry out a number of checks on other entities before allowing the new entity to be created. It may be necessary to invoke other operations as a consequence of the new entity's creation. These checks and operations are entity-type specific.

The next stage in semantic data modelling, is the integration of operator definition with the data structuring facilities, such that operator definitions are entity-type specific. The Object-oriented paradigm is one possible way to attempt this integration, by providing a mechanism for progressing from a purely structural model of data towards a more behavioural model, combining facilities for both the representation and manipulation of data within the same model.

Review questions 1

- Describe some of the shortcomings in the Relational approach to database systems, which have lead people to look for alternative database technologies for some applications.
- Identify a further example of each of the three types of relationship mentioned in the text: existence dependency, association and categorisation.

Object-oriented concepts

Combining structure and behaviour

A basic difference between traditional databases and Object databases, is the way in which the passive and active elements of the underlying system are implemented. Traditional databases are seen as passive, storing data which is retrieved by an application, manipulated and then updated on the database. This is in contrast to the active, Object-oriented approach where the manipulation occurs within the database itself. It is also possible to use Object-oriented (OO) databases passively; however, this means that they are not necessarily being used to their full potential.

The inclusion of the behaviour, or processing, related to an object, along with the definition of the structure of the object, stored within the database itself, is what distinguishes the Object-oriented approach from semantic data models, which purely try to improve the level of meaning supported by the data model of the database system. The way in which active behaviour is supported within Object databases, is via the message/method feature.

Messages

If object A in the database wants object B to do something, it sends B a message. The success or failure of the requested operation may be conveyed back from object B to object A, via a further message. In general, each object has a set of messages that it can receive and a set of replies it can send. An object does not need to know anything about the other objects it interacts with, other than what messages can be sent to them, and what replies it can receive from them. The internal workings are thus encapsulated into the definition for each object.

Methods

Methods are procedures, internal to each object, which alter an object's private state. State here means the values of the data items of the object in question.

Examples of methods

Some examples of commonly found methods are as follows:

- **Constructors:** These are used whenever a new instance of an object is created. They initialise the data items contained within the object instance. It is possible for objects to have a number of different constructors, if it is required that they should sometimes be created with different starting values for their data items.

- **Destructors:** These methods are used when an instance of an object is deleted. They ensure that any resources that are held by the object instance, such as storage space, are released.
- **Transformers:** These methods are used to change an object's internal state. There may be a number of transformer methods used to bring about changes to the data items of an object instance.

The Object-oriented approach, therefore, provides the ability to deal with objects and operations on those objects, that are more closely related to the real world. This has the effect of raising the level of abstraction from that used in Relational constructs, such as tables, theoretically making the data model easier to understand and use.

Defining objects - Class definitions

In the Object-oriented approach, everything can, in some way, be described as an object. The term usually applies to a person, place or thing that a computer application may need to deal with. In traditional database terms, an object can be likened to an entity in an E-R diagram, but instead of the entity merely containing attributes, it can also contain methods, sometimes known as operations. These methods are fragments of program code, which are used to carry out operations relevant to the object in question. For example, a Customer object, as well as having the traditional data items we might expect to see in a Customer table, may include operations such as CREATE A NEW CUSTOMER INSTANCE (constructor), REMOVE A CUSTOMER INSTANCE (destructor), CHANGE CUSTOMER DETAILS (transformer), etc.

The attributes and methods for groups or classes of objects of the same type are described in a class definition. Each particular object is known as an instance of that class. The class definition is like a template, therefore, which defines the set of data items and methods available to all instances of that class of object. Some Object database systems also permit the definition of database constraints within class definitions, a feature which might be considered to be a specific case of method definition.

Example of class definition

Consider the object type 'book' as might exist in a library database. Information to be held on a book include its title, date of publication, publisher and author. Typical operations on a book might be:

- Take a book out on loan.
- Reserve a book for taking out on loan when available.
- A Boolean function which returns true if the book is currently on loan and false otherwise. The above operations will be implemented as methods of class 'book'.

The class book may be defined by the following structure:

class book

properties

title : string;

date_of_Publication : date;

published_by : publisher;

written_by : author;

operations

create () -> book;

loan (book, borrower, date_due);

reserve (book, borrower, date_reserved);

on_loan (book) -> boolean;

end book;

A method can receive additional information, called parameters, to perform its task. In the above class, loan method expects a book, borrower and date due for it to perform the loan operation. Parameters are put in the parenthesis of a method. When a method performs its task, it can return data back to the caller method.

An important point to note here is that data abstraction as provided by the class mechanism allows one to define properties of entities in terms of other entities. Thus we see from the above example that the properties published_by and written_by are defined in terms of the classes 'publisher' and 'author' respectively. Outline class definitions for author and publisher could be as follows:

class author

properties

surname : string;

initials : string;

nationality : country;

year_of_birth : integer;

year_of_death : integer;

operations

create () -> author;

end author.

class publisher

properties

name : string;

location : city;

operations

create () -> publisher;

end publisher.

Inheritance

When defining a new class, it can either be designed from scratch, or it can extend or modify other classes - this is known as inheritance. For example, the class 'manager' could inherit all the characteristics of the class 'employee', but also be extended to encompass features specific to managers. This is a very powerful feature, as it allows the reuse and easy extension of existing data definitions and methods (note that inheritance is not just restricted to data; it can apply equally to the methods of a class). Some systems only permit the inheritance of the data items (sometimes called the state or properties) of a class definition, while others allow inheritance of both state and behaviour (the methods of a class). Inheritance is a powerful mechanism, as it provides a natural way for applications or systems to evolve. For example, if we wish to create a new class of product, we can easily make use of any previous development work that has gone into the definition of data structures and methods for existing products, by allowing the definition of the new class to inherit them.

Example of class definitions to illustrate inheritance:

As an example, we might take the object classes 'mammal', 'bird' and 'insect', which may be defined as subclasses of 'creature'. The object class 'person' is a subclass of 'mammal', and 'man' and 'woman' are subclasses of 'person'. Class definitions for this hierarchy might take the following form:

class creature

properties

type : string; weight : real;

habitat : (… some habitat type such as swamp, jungle, urban);

operations

create () -> creature;

predators (creature) -> set (creature);

life_expectancy (creature) -> integer;

end creature.

class mammal inherit creature;

properties

gestation_period : real;

operations

end mammal.

class person inherit mammal;

properties

surname, firstname : string;

date_of_birth : date;

origin : country;

end person.

class man inherit person;

properties

wife : woman;

operations

end man.

class woman inherit person;

properties

husband : man;

operations

end woman.

The inheritance mechanism may be used not only for specialisation as described above, but for extending software modules to provide additional services (operations). For example, if we have a class (or module) A with subclass B, then B provides the services of A as well as its own. Thus B may be considered as an extension of A, since the properties and operations applicable to instances of A are a subset of those applicable to instances of B.

This ability of inheritance to specify system evolution in a flexible manner is invaluable for the construction of large software systems. For database applications, inheritance has the added advantage of providing the facility to model natural structure and behaviour.

It is possible in some systems, to inherit state and/or behaviour from more than one class. This is known as multiple inheritance; it is only supported in some Object-oriented systems.

Encapsulation

Encapsulation in object oriented means an object contains both the data structures and the methods to manipulate the data structures. The data structures are internal to the object and are only accessed by other objects through the public methods. Encapsulation ensures that changes in the internal data structure of an object does not affect other objects provided the public methods remains the same. Encapsulation provides a form of data independence.

Review question 2

- Describe the difference between methods and messages in Object-oriented systems.
- Describe a situation in which it may be necessary to provide two different constructor methods for instances of an object.
- Describe the main advantages of inheritance.
- Describe the concept of encapsulation in Object-oriented systems.

Implementing an application of Object databases

Implementing Object databases

An important difference between databases and OO languages is that OO languages create objects in memory, and when an OO application ends, all objects created by the application are destroyed and the data must be written to files in order to be used at a later date. Conversely, databases require access to persistent data. Pure Object-oriented databases make use of Object technology by adding persistence to existing Object-oriented languages; this allows data to be stored as objects even when a program is not running.

In order to implement and manipulate an OO database, it is necessary to use a language that is capable of handling OO concepts. According to Silberschatz (1997) there are several ways in which to do this:

- An existing Relational data-manipulation language can be extended to handle complex data-types and Object-orientation. This leads to Object-Relational systems, discussed later in this chapter.
- A purer OO alternative is to extend an existing OO language to deal with databases, and so it becomes a persistent programming language. C++ and other languages have all had persistent versions implemented.

- The Object database system may be built as such from the beginning. db4objects, DTS/S1, Perst, etc, are examples of pure Object database systems which have been built using this approach.

The use of OO languages allows programmers to directly manipulate data without having to use an embedded data manipulation language such as SQL. This gives programmers a language that is computationally complete and therefore provides greater scope for creating effective business solutions.

Applications for OO databases

There are many fields where it is believed that the OO model can be used to overcome some of the limitations of Relational technology, where the use of complex data types and the need for high performance are essential. These applications include:

- Computer-aided design and manufacturing (CAD/CAM)
- Computer-integrated manufacturing (CIM)
- Computer-aided software engineering (CASE)
- Geographic information systems (GIS)
- Many applications in science and medicine
- Document storage and retrieval

Problems with the OO model

One of the key arguments against OO databases is that databases are usually not designed to solve specific problems, but need the ability to be used to solve many different problems not always apparent at the design stage of the database. It is for this reason that OO technology, and its use of encapsulation, can often limit its flexibility. Indeed the ability to perform ad hoc queries can be made quite difficult, although some vendors do provide a query language to facilitate this.

The use of the same language for both database operations and system operations can provide many advantages, including that of reducing the impedance mismatch: the difference in level between set-at-a-time and record-at-a-time processing. Date (2000), however, does not agree that this is best achieved by making the database language record-at-a-time; he even goes as far as to say that "record-at-a-time is a throwback to the days of pre-Relational systems such as IMS and IDMS". Instead, he proposes that set-at-a-time facilities be added to programming languages. Nonetheless, it could be argued that one of the advantages of pre-Relational systems was their speed. The procedural nature of

OO languages can still lead to serious difficulties when it comes to optimisation, however.

Another problem associated with pure OO databases is that in many cases its use is comparable to that of using a sledgehammer to crack a nut. A large proportion of organisations do not currently deal with the complex data types that OO technology is ideally suited too, and therefore do not require complex data processing. For these companies, there is little incentive for them to move towards Object technology when Relational databases and online analytical processing tools will be sufficient to satisfy their data processing requirements for several years to come. Of course, it is always possible that these companies will find a use for the technology as its popularity becomes more widespread.

The future of OO databases

Many applications falling into the categories cited earlier have been successfully implemented using pure OO techniques. However, the aforementioned problems associated with the OO database model have led to some people doubting as to whether pure OO really is the way forward for databases, particularly with regard to mainstream business applications. Date (2000) is a particularly vehement opponent of pure OO technology, arguing instead that the existing Relational model should evolve to include the best features of Object-orientation and that OO in itself does not herald the dawn of the third generation of database technology.

The Object-Relational model

Perhaps the best hope for the immediate future of database objects is the Object-Relational model. A recent development, stimulated by the advent of the Object-oriented model, the Object-Relational model aims to address some of the problems of pure OO technology - such as the poor support for ad hoc query languages - and open database technology, and provide better support for existing relational products, by extending the Relational model to incorporate the key features of Object-orientation. The Object-Relational model also provides scope for those using existing Relational databases to migrate towards the incorporation of objects, and this perhaps is its key strength, in that it provides a path for the vast number of existing Relational database users gradually to migrate to an Object database platform, while maintaining the support of their Relational vendor.

A major addition to the Relational model is the introduction of a stronger type of system that can accommodate the use of complex data types, which still allow the Relational model to be preserved. Several large database suppliers, including IBM Informix and Oracle, have embraced the Object-Relational model as the way forward.

DB2 Relational Extenders

IBM DB2 Relational Extenders are built on the Object/Relational facilities first introduced in DB2 version2. These facilities form the first part of IBM's implementation of the emerging SQL3 standard. It includes UDTs (User Defined Types), UDFs (User Defined Functions), large objects (LOBs), triggers, stored procedure and checks.

The DB2 Relational Extenders are used to define and implement new complex data types. The Relational Extenders encapsulate the attribute structure and behaviour of these new data types, storing them in table columns of a DB2 database. The new data types can be accessed through SQL statements in the same manner as the standard DB2 data types. The DBMS treats these data types in a strongly typed manner, ensuring that they are only used where data items or columns of the particular data type are anticipated. A DB2 Relational Extender is therefore a package consisting of a number of UDTs, UDFs, triggers, stored procedures and constraints.

When installing a Relational Extender on a database, various files are copied into the server's directories, including the function library containing the UDFs. Then an application is run against the database to define the Relational Extender's database definition to the server. These include scripts to define the UDTs and UDFs making up the Relational Extender.

IBM Informix DataBlades

The DataBlades are standard software modules that plug into the database and extend its capabilities. A DataBlade is like an Object-oriented package, similar to a C++ class library that encapsulates a data object's class definition. The DataBlade not only allows the addition of new and advanced data types to the DBMS, but it also enables specification of new, efficient and optimised access and processing methods for these data types.

A DataBlade includes the data type definition (or structure) as well as the methods (or operations) through which it can be processed. It also includes the rules (or integrity constraints) that should be enforced, similar to a standard built-in data type.

A DataBlade is composed of UDT, a number of UDFs, access methods, interfaces, tables, indexes and client code.

Object-Relational features in Oracle 11

Important

The object features described in the following can only be used with Oracle Enterprise edition. In particular, if you are using Personal Oracle edition for

this module, you will not be able to create the objects described. You will however be able to perform the required activities, as these involve examining sample scripts that are included in the Oracle Personal Edition package. If your Learning Support Centre has a version of Oracle running on a mainframe or minicomputer, it is possible that access to the Enterprise Edition of Oracle can be provided. This is not necessary for completion of the activities and exercises of this chapter, but would be necessary if you wish to consolidate the information given here with some practical experience of Oracle's object features.

We shall examine in some detail the facilities incorporated in Oracle11, as these provide a good example of how one of the major database vendors is seeking to increase the level of Object support within the DBMS, while maintaining support for the Relational model.

Abstract data types

Abstract data types (ADTs) are provided to enable users to define complex data types, which are structures consisting of a number of different elements, each of which uses one of the base data types provided within the Oracle product. For example, an abstract data type could be created to store addresses. Such a data type might consist of three separate base attributes, each of which is of type varchar(30). From the time of its creation, an ADT can be referred to when creating tables in which the ADT is to be used. The address ADT would be established with the following definition in Oracle 8:

CREATE TYPE ADDRESS_TYPE AS OBJECT (STREET VARCHAR2(30),

CITY VARCHAR2(30),

COUNTRY VARCHAR2(30));

ADTs can be nested (their definitions can make use of other ADTs). For example, if we wished to set up an ADT to describe customers, we could make use of the address ADT above as follows:

CREATE TYPE CUSTOMER_TYPE AS OBJECT (CUST_NO NUMBER(6),

NAME VARCHAR2(50),

BIRTHDATE DATE,

GENDER CHAR,

ADDRESS ADDRESS_TYPE);

The advantages of ADTs are that they provide a standard mechanism for defining complex data types within an application, and facilitate reuse of complex data definitions.

Object tables

These are tables created within Oracle11 which have column values that are based on ADTs. Therefore, if we create a table which makes use of the customer and address ADTs described above, the table will be an object table. The code to create such a table would be as follows:

CREATE TABLE CUSTOMER OF CUSTOMER_TYPE;

Note that this CREATE TABLE statement looks rather different to those encountered in the chapter on SQL Data Definition Language (DDL). It is very brief, because it makes use of the previous work we have done in establishing the customer and address ADTs.

It is extremely important to bear in mind the distinction between object tables and ADTs.

ADTs are the building blocks on which object tables can be created. ADTs themselves cannot be queried, in the same way that the built-in data types in Oracle such as number and varchar2 cannot be queried. ADTs simply provide the structure which will be used when objects are inserted into an object table. Object tables are the element which is queried, and these are established using a combination of base data types such as varchar2, date, number and any relevant ADTs as required.

Nested tables

A nested table is a table within a table. It is a collection of rows, represented as a column in the main table. For each record in the main table, the nested table may contain multiple rows. This can be considered as a way of storing a one-to-many relationship within one table. For example, if we have a table storing the details of departments, and each department is associated with a number of projects, we can use a nested table to store details about projects within the department table. The project records can be accessed directly through the corresponding row of the department table, without needing to do a join. Note that the nested table mechanism sacrifices first normal form, as we are now storing a repeating group of projects associated with each department record. This may be acceptable, if it is likely to be a frequent requirement to access departments with their associated projects in this way.

Varying arrays

A varying array, or varray, is a collection of objects, each with the same data type. The size of the array is preset when it is created. The varying array is treated like a column in a main table. Conceptually, it is a nested table, with a preset limit on its number of rows. Varrays also then allow us to store up to a preset number of repeating values in a table. The data type for a varray is determined by the type of data to be stored.

Support for large objects

Large objects, or LOBs as they are known in Oracle8, are provided for by a number of different predefined data types within Oracle11. These predefined data types are as follows:

- Blob: Stores any kind of data in binary format. Typically used for multimedia data such as images, audio and video.
- Clob: Stores string data in the database character set format. Used for large strings or documents that use the database character set exclusively. Characters in the database character set are in a fixed-width format.
- Nclob: Stores string data in National Character Set format. Used for large strings or documents in the National Character Set. Supports characters of varying-width format.
- Bfile: Is a pointer to a binary file stored outside of the database in the host operating system file system, but accessible from database tables.

It is possible to have multiple large objects (including different types) per table.

Summary

Despite the advances made in OO technology and its widespread acceptance in general programming use, pure Object-orientation has only achieved serious acceptance in a limited number of specialised fields and not general, industrial-strength applications. The two main reasons for this appear to be the problems that moving to OO introduces, in addition to the fact that Relational technology still has a great deal to offer. The way forward for the use of objects in databases seems to be the Object-Relational model, extending the existing Relational model to incorporate the best features of OO technology, thus delivering the best of both worlds.

Discussion topic

There are a number of applications, such as engineering design, for which Object-oriented database systems are clearly superior to Relational systems. For a number of commercial applications, however, the advantage is perhaps less clear. Imagine you are starting up a company, which requires to keep data about customers, orders, products and sales. Discuss with your colleagues whether you would prefer to go for a Relational, Object-Relational or Object-oriented database solution. Factors you should take into account are as follows:

- The nature of the products to be sold.
- Whether the database is to be connected to the Internet.

- The volume of the data (both in terms of the numbers of records of each type, and the frequency of transactions to be supported).

Consider in your discussions the way in which each of these factors might affect your decision.

Further work

Polymorphism

Object-orientation contains a number of new concepts and terminology, most of which have been introduced to some extent in this chapter. One important area that has not been covered in detail, is the ability to provide alternative implementations of computer processing. For example, it may be required to calculate the salary of full-time employees in one way, and of part-time employees in another. This facility can be provided in Object-oriented systems through the mechanism of polymorphism. Using the core text for the module, investigate the concept of polymorphism, and identify two further situations where it might be applied.

Chapter 17. Web Database Connectivity

Table of contents

        * Benefits of template-driven packages
        * Shortcomings of template-driven packages
    - GUI application builders
        * The approach
        * Benefits of visual tools
        * Shortcomings of visual tools
- Managing state and persistence in Web applications
    - Technical options
    - The URL approach
        * Benefits of the URL approach
        * Shortcomings of the URL approach
    - URL QUERY_STRING
        * Benefits of the hidden fields approach
        * Shortcomings of the hidden fields approach
    - HTTP cookies
        * Benefits of cookies
        * Shortcomings of cookies
    - Important considerations
        * Managing state on the client
        * Managing state on the server
- Security Issues in Web Database Applications
    - Proxy servers
    - Firewalls
    - Digital signatures
    - Digital certificates
    - Kerberos
    - Secure sockets layer (SSL) and secure HTTP (S-HTTP)
    - Java security
    - ActiveX security
- Performance issues in Web database applications
- Discussion topics

Objectives

At the end of this chapter you should be able to:

- Understand the requirements for connecting database systems to the Web.
- Critically compare a number of approaches that might be used to build the Web database connectivity.
- Make recommendations for a given company and specific scenario regarding which of the commonly used mechanisms is likely to be most appropriate, taking into consideration relative cost, security, likely transaction volumes and required performance.

Introduction

In parallel with this chapter, you should read Chapter 29 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

This chapter introduces you to the exciting topic of combining World Wide Web (WWW) technology with that of databases. It is about bridging a gap between new technologies and 'old' in order to achieve what has never been achievable before. The emergence of the WWW is arguably one of the most important technological advances in this century, and since its birth a decade ago, it has changed many people's lives and had a profound impact on society.

The Web has been expanding at an incredible speed and even while you are reading this, hundreds and thousands of people are getting 'online' and hooked to the Web. Reactions to this technology are understandably mixed. People are excited, shocked, confused, puzzled or even angered by it. Whatever your reaction might be, you are being affected and benefiting from it. Without the Web technology, the creation of a global campus would not have been possible. It is fair to say that the WWW is playing and will continue to play an important role (perhaps the most important role) in shaping the future world of technology, business and industry.

The database technology has been around for a long time now, and for many business and government offices, databases systems have already become an essential and integral part of the organisation. Now the new technology has given the 'old' a shot in the arm, and the combination of the two creates many exciting opportunities for developing advanced database applications, which will in turn produce additional benefits for the traditional database applications. A multinational company, for example, can create a Web-based database application to enable the effective sharing of information among offices around the world.

As far as database applications are concerned, a key aspect of the WWW technology is that it offers a brand new platform to collect, deliver and disseminate information. Via the Web, a database application can be made available, interactively, to users and organisations anywhere in the world.

In this chapter, we are going to examine the impact that the WWW brings to the 'traditional' database technology. We will see how databases can be connected to the Web, and the most effective way of using the new technology to develop database applications. We will also study the most commonly used approaches for creating Web databases, and discuss related issues such as dynamic updating of Web pages inline with the changes in databases, performance, and security concerns.

As the Web contains many large, complex, multimedia documents, the materials covered in this chapter are relevant to the discussion of Object-oriented

databases. The reason is that the Object-oriented model is considered the most suitable for the storage, organisation and retrieval of large sets of Web documents. Also, the need to process high volumes of queries and updates over the Web has an important impact on performance considerations. Traditional techniques need to be adapted or even changed to satisfy performance requirements of Web applications. Lastly, the discussion about client-server applications (Chapter 15) is also relevant to this chapter, because Web databases represent a new type of such applications.

Context

Basic concepts

Before we start to discuss Web database applications, we need to clarify a number of related terms and concepts.

Internet: It is a worldwide collection of interconnected computer networks, which belong to various organisations (e.g. educational, business and governments). It is not synonymous to the WWW. The services that are normally available on the Internet include email, real-time communication (e.g. conferencing and chat), news services, and facilities for accessing remote computers to send and receive documents.

The WWW or simply the Web: The WWW comprises software (e.g. Web servers and browsers) and data (e.g. Web sites). It simply represents a (huge) set of information resources and services that live on the Internet. Each Web site consists of a set of Web pages, which typically contain multimedia data (e.g. text, images, sound and video). In addition, a Web page can include hyperlinks to other Web pages which allow users (also called net surfers) to navigate through the Web of information pages.

Intranet: A Web site or group of sites which belongs to an organisation and can only be accessed by members of that organisation. Between the Internet and an intranet, there is an extra layer of software or hardware called a firewall. Its main function is to prevent unauthorised access to a private network (e.g. an intranet) from the Internet.

Extranet: An intranet which allows partial access by authorised users from outside the organisation via the Internet.

HTTP (HyperText Transfer Protocol): The standard protocol for transferring Web pages through the Internet. HTTP defines how clients (i.e. users) and servers (i.e. providers) should communicate.

HTML (HyperText Markup Language): A simple yet powerful language that is commonly used to format documents which are to be published on the Web.

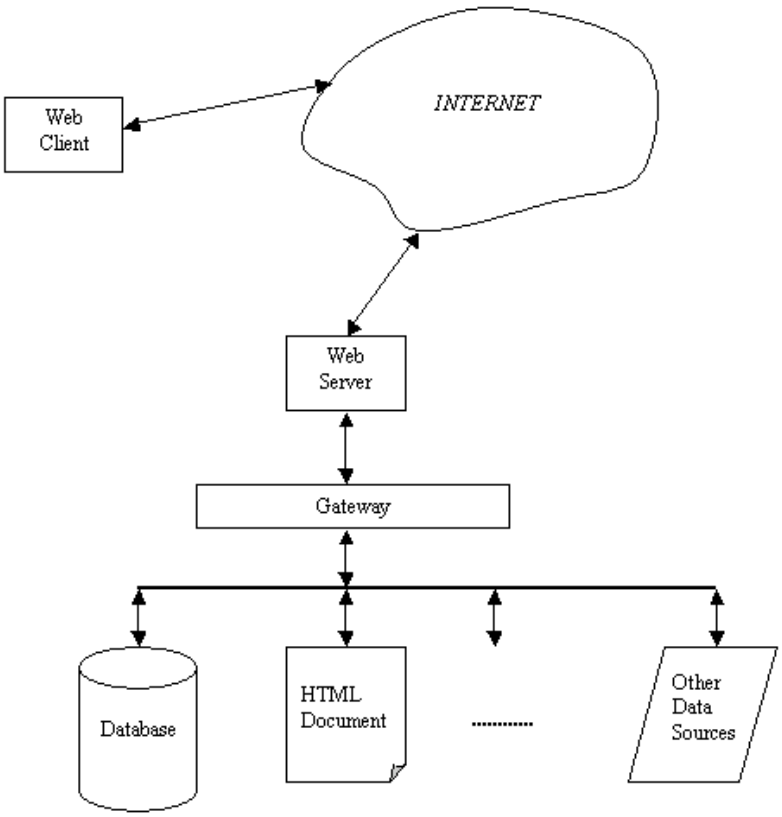
URL (Uniform Resource Locator): A string of alphanumeric characters that represents the location of a resource (e.g. a Web page) on the Internet and how that resource should be accessed.

There are two types of Web pages: static and dynamic.

1. **Static:** An HTML document stored in a file is a typical example of a static Web page. Its contents do not change unless the file itself is changed.
2. **Dynamic:** For a dynamic Web page, its contents are generated each time it is accessed. As a result, a dynamic Web page can respond to user input from the browser by, for example, returning data requested by the completion of a form or returning the result of a database query. A dynamic page can also be customised by and for each user. Once a user has specified some preferences when accessing a particular site or page, the information can be recorded and appropriate responses can be generated according to those preferences.

From the above, it can be seen that dynamic Web pages are much more powerful and versatile than static Web pages, and will be a focus for developing Web database applications. When the documents to be published are dynamic, such as those resulting from queries to databases, the appropriate hypertext needs to be generated by the servers. To achieve this, we must write scripts that perform conversions from different data formats into HTML 'on-the-fly'. These scripts also need to recognise and understand the queries performed by clients through HTML forms and the results generated by the DBMS.

In short, a Web database application normally interacts with an existing database, using the Web as a means of connection and having a Web browser or client program on the front end. Typically such applications use HTML forms for collecting user input (from the client); CGI (Common Gateway Interface, to be discussed later) to check and transfer the data from the server; and a script or program which is or calls a database client to submit or retrieve data from the database. The diagram below gives a graphical illustration of such a scenario. More will be discussed in later parts of this chapter.



Web-based client-server applications

As mentioned earlier in the Introduction, Web-based database applications are a new type of client-server application. Some of the traditional client-server database techniques may still be adapted. However, because of the incorporation of the Web technology, there are important differences as set out in the following table.

Tradition Client-Server Applications	Web-Based Applications
Platform-dependent	Platform-independent
Client is natively compiled and therefore has fast execution speed.	Client is an interpreter (e.g., HTML, Java, Java Script) and therefore is slower.
Installation necessary.	No need for installation.
Complex client; high maintenance cost.	Simple client; minimum maintenance.
New, unfamiliar interface for users.	One common, familiar interface across applications.
Rich, custom GUI constructs possible.	Limited set of GUI constructs; custom ones add to download time.
Difficult to integrate with existing applications.	Easy to integrate.
Difficult to add multimedia.	Easy to add multimedia.
Persistent connection to database.	Non-persistent connection.

Platform independence: Web clients are platform-independent and do not require modification to be run on different operating systems. Traditional database clients, on the other hand, require extensive porting efforts to support multiple platforms. This is arguably one of the most compelling reasons for building a Web-based client-server database application.

Interpreted applications: Web applications are written in interpreted languages (e.g. HTML and Java). This has an adverse effect on performance. In many applications, however, this is a price worth paying to gain the advantage of being platform independent. Time-critical applications may not be good candidates to be implemented on the Web.

No need for installation: Another benefit of Web database applications is that the need for installing special software is eliminated on the clients' side. It is pretty safe to assume that the clients have already had a Web browser installed, which is the only piece of software needed for the clients to run the applications.

Simple client: As a client needs just a browser to run a Web-based database application, the potential complications are minimised.

Common interface across applications: Again, because there is no need for specialised software, users have the benefit of using a browser for possibly different applications.

Limited GUI (Graphical User Interface): This is one area in which Web-based database applications may fall short. Highly customised application interfaces and highly interactive clients may not translate well as Web applications. This is because of the HTML limitations. At the moment, HTML forms do not offer an extensive feature set. Although JavaScript language can extend the functionality of HTML-based applications, it is too complex, adds to download-

ing time, and degrades performance.

Integrate with other applications: Because of the benefit of being platform independent, different applications that adhere to the HTML standard can be integrated without many difficulties.

Non-persistent connection to database: Persistent database connections are highly efficient data channels between a client and the DBMS, and therefore, are ideal for database applications. However, Web-based applications do not have this benefit. A Web-based client maintains its connection to a database only as long as is necessary to retrieve the required data, and then releases it. Thus, Web application developers must address the added overhead for creating new database connections each time a client requires database access.

Apart from the above differences, there are some other important concerns for Web-based applications:

- **Reliability of the Internet:** At the moment, there are reliability problems with the Internet. It may break down; data may be lost on the net; large amounts of data traffic may slow down or even overwhelm the network system.
- **Security:** Security on the Internet is of great concern for any organisation which has developed Web-based database applications. For example, the database may be broken into, or confidential data may be intercepted during transmission by unauthorised parties or even criminals.

At the present, a lot of research and development work is being carried out to address these concerns. There is no doubt that the potential problems can be overcome and over time, the Internet will be more reliable and more secure for connecting the world.

Context summary

In this section, we have drawn an overall picture of Web-based database applications. We have briefly mentioned how a DBMS can be integrated with the Web, and what their advantages and disadvantages are as compared to the traditional client-server applications. In the rest of this chapter, we will study the details of creating Web database applications and discuss the commonly used approaches of linking databases to the Web.

Review question 1

- Are the Internet and WWW (Web) the same concept? Why?
- What are intranets and extranets?
- What is a URL?
- Most Web sites have URLs starting with http://……. Why?

- What is a dynamic Web page? What are its characteristics?
- What are the major features of a Web-based client-server application?

Web database architectures

Components of a database application

Web database applications may be created using various approaches. However, there are a number of components that will form essential building blocks for such applications. In other words, a Web database application should comprise the following four layers (i.e. components):

- Browser layer
- Application logic layer
- Database connection layer
- Database layer

Browser layer

The browser is the client of a Web database application, and it has two major functions. First, it handles the layout and display of HTML documents. Second, it executes the client-side extension functionality such as Java, JavaScript, and ActiveX (a method to extend a browser's capabilities).

The three most popular browsers at the present are Mozilla Firefox (Firefox for short), Google Chrome and Microsoft Internet Explorer (IE).

All three browsers are graphical browsers. During the early days of the Web, a text-based browser, called Lynx, was popular. As loading graphics over the Internet can be a slow and time-consuming process, database performance may be affected. If an application requires a speedy client and does not need to display graphics, then the use of Lynx may be considered.

All browsers implement the HTML standard. The discussion of HTML is beyond this chapter, but you need to know that it is a language used to format data/documents to be displayed on the Web.

Browsers are also responsible for providing forms for the collection of user input, packaging the input, and sending it to the appropriate server for processing. For example, input can include registration for site access, guest books and requests for information. HTML, Java, JavaScript or ActiveX (for IE) may be used to implement forms.

Application logic layer

The application logic layer is the part of a Web database application with which a developer will spend the most time. It is responsible for:

- Collecting data for a query (e.g. a SQL statement).
- Preparing and sending the query to the database via the database connection layer.
- Retrieving the results from the connection layer.
- Formatting the data for display.

Most of the application's business rules and functionality will reside in this layer. Whereas the browser client displays data as well as forms for user input, the application logic component compiles the data to be displayed and processes user input as required. In other words, the application logic generates HTML that the browser renders. Also it receives, processes and stores user input that the browser sends.

Depending on the implementation methods used for the database application, the application logic layer may have different security responsibilities. If the application uses HTML for the front end, the browser and server can handle data encryption (i.e. a security measure to ensure that data will not be able to be intercepted by unauthorised parties). If the application is a Java applet and uses Java for the front end, then it itself must be responsible for adopting transmission encryption.

Database connection layer

This is the component which actually links a database to the Web server. Because manual Web database programming can be a daunting task, many current Web database building tools offer database connectivity solutions, and they are used to simplify the connection process.

The database connection layer provides a link between the application logic layer and the DBMS. Connection solutions come in many forms, such as DBMS net protocols, API (Application Programming Interface [see note below]) or class libraries, and programs that are themselves database clients. Some of these solutions resulted in tools being specifically designed for developing Web database applications. In Oracle, for example, there are native API libraries for connection and a number of tools, such as Web Publishing Assistant, for developing Oracle applications on the Web.

The connection layer within a Web database application must accomplish a number of goals. It has to provide access to the underlying database, and also needs to be easy to use, efficient, flexible, robust, reliable and secure. Different tools and methods fulfil these goals to different extents.

Note

An API consists of a set of interrelated subroutines that provide the functionality required to develop programs for a target operating environment. For example, Microsoft provides different APIs targeted at the construction of 16- and 32-bit Windows applications. An API would provide functions for all aspects of system activity, such as memory, file and process management. Specialised APIs are also supplied by software vendors to support the use of their products, such as database and network management systems.

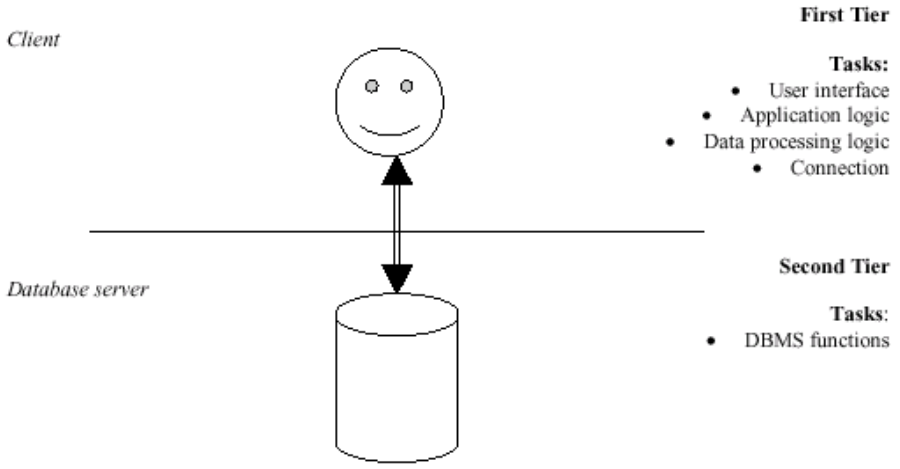
Database layer

This is the place where the underlying database resides within the Web database application. As we have already learned, the database is responsible for storing, retrieving and updating data based on user requirements, and the DBMS can provide efficiency and security measures.

In many cases, when developing a Web database application, the underlying database has already been in existence. A major task, therefore, is to link the database to the Web (the connection layer) and to develop the application logic layer.

2-tier client-server architecture

Traditional client-server applications typically have a 2-tier architecture as illustrated in the figure below. The client (tier 1) is primarily responsible for the presentation of data to the user, and the server (tier 2) is responsible for supplying data services to the client. The client will handle user interfaces and main application logic, and the server will mainly provide access services to the underlying database.

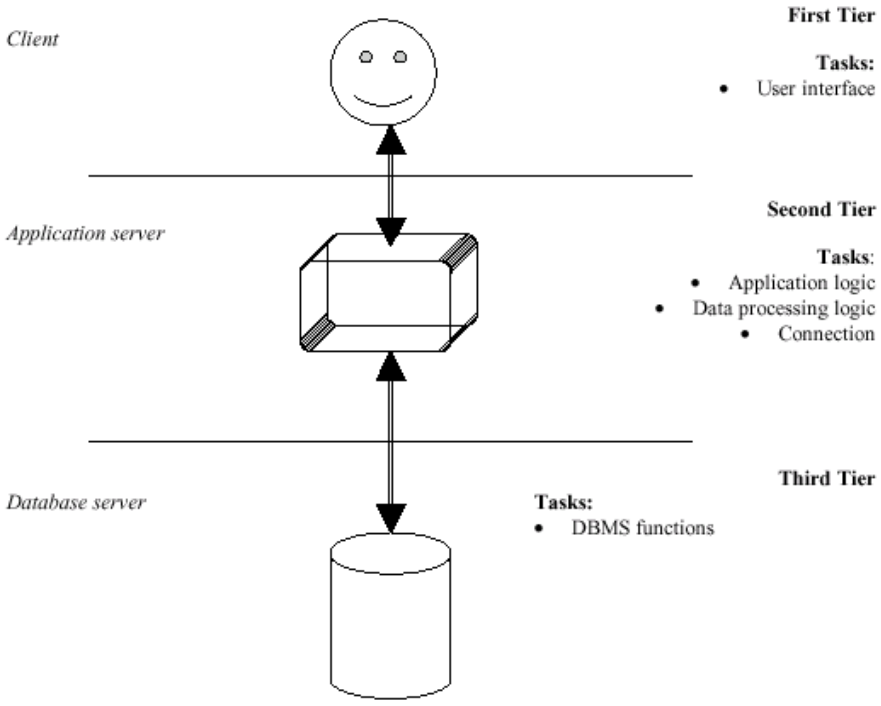


If such a 2-tier architecture is used to implement a Web database application, tier 1 will contain the browser layer, the application logic layer and the connection layer. Tier 2 accommodates the DBMS. This will inevitably result in a fat client.

3-tier client-server architecture

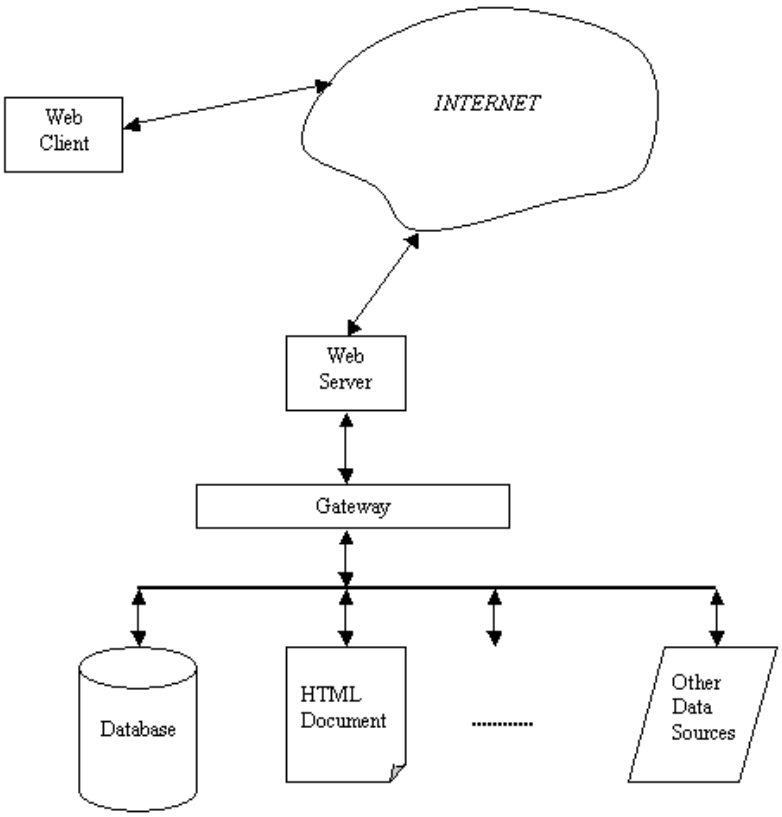
In order to satisfy requirements of increasingly complex distributed database applications, a 3-tier architecture was proposed to replace the 2-tier one. There are three tiers in this new architecture, each of which can potentially run on a different platform.

The first tier is the client, which contains user interfaces. The middle tier accommodates the application server, which provides application logic and data processing functions. The third tier contains the actual DBMS, which may run on a separate server called a database server.



The 3-tier architecture is more suitable for implementing a Web database application. The browser layer can reside in tier 1, together with a small part of the application logic layer. The middle tier implements the majority of the application logic as well as the connection layer. Tier 3 is for the DBMS.

Referring to the figure below, for example, it can be seen that the Web Client is in the first tier. The Web Server and Gateway are in the middle tier and they form the application server. The DBMS and possibly other data sources are in the third tier.



Having studied the Web database architectures, we should understand that the most important task in developing a Web database application is to build the database connection layer. In other words, we must know how to bridge the gap between the application logic layer and the database layer.

Review question 2

- What is the typical architecture of a Web database application?
- How can a 3-tier client-server architecture be used to implement a Web database application?

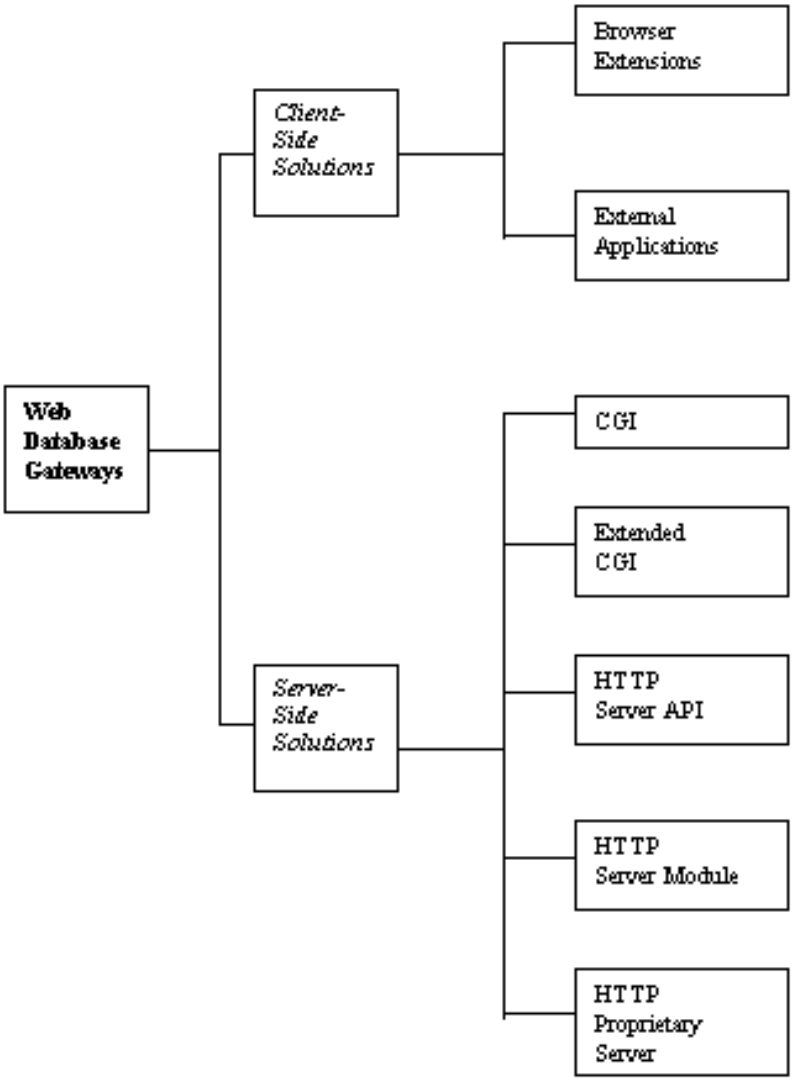
Database gateways

A Web database gateway is a bridge between the Web and a DBMS, and its objective is to provide a Web-based application the ability to manipulate

data stored in the database. Web database gateways link stateful systems (i.e. databases) with a stateless, connectionless protocol (i.e. HTTP). HTTP is a stateless protocol in the sense that each connection is closed once the server provides a response. Thus, a Web server will not normally keep any record about previous requests. This results in an important difference between a Web-based client-server application and a traditional client-server application:

- In a Web-based application, only one transaction can occur on a connection. In other words, the connection is created for a specific request from the client. Once the request has been satisfied, the connection is closed. Thus, every request involving access to the database will have to incur the overhead of making the connection.
- In a traditional application, multiple transactions can occur on the same connection. The overhead of making the connection will only occur once at the beginning of each database session.

There are a number of different ways to create Web database gateways. Generally, they can be grouped into two categories: client-side solutions and server-side solutions, as illustrated below:



Client-side solutions

The client-side solutions include two types of approaches for connections: browser extensions and external applications.

Browser extensions are add-ons to the core Web browser that enhance and augment the browser's original functionality. Specific methods include plug-ins for Firefox, Chrome and IE, and ActiveX controls for IE. Also, all the three types of browsers (Firefox, Chrome and IE) support Java and JavaScript languages (i.e. Java applets and JavaScript can be used to extend browsers' capabilities).

External applications are helper applications or viewers. They are typically existing database clients that reside on the client machine and are launched by the Web browser in a particular Web application. Using external applications is a quick and easy way to bring legacy database applications online, but the resulting system is neither open nor portable. Legacy database clients do not take advantages of the platform independence and language independence available through many Web solutions. Legacy clients are resistant to change, meaning that any modification to the client program must be propagated via costly manual installations throughout the user base.

Server-side solutions

Server-side solutions are more widely adopted than the client-side solutions. A main reason for this is that the Web database architecture requires the client to be as thin as possible. The Web server should not only host all the documents, but should also be responsible for dealing with all the requests from the client.

In general, the Web server should be responsible for the following:

- Listening for HTTP requests.
- Checking the validity of the request.
- Finding the requested resource.
- Requesting authentication if necessary.
- Delivering requested resource.
- Spawning programs if required.
- Passing variables to programs.
- Delivering output of programs to the requester.
- Displaying error message if necessary.

The client (browser) should be responsible for some of the following:

- Rendering HTML documents.
- Allowing users to navigate HTML links.
- Displaying image.
- Sending HTML form data to a URL.

- Interpreting Java applets.
- Executing plug-ins.
- Executing external helper applications.
- Interpreting JavaScript and other scripting language programs.
- Executing ActiveX controls in the case of IE.

In the following sections, we are going to discuss both client-side and server-side solutions in some detail.

Review question 3

- What is a gateway in a Web database application and why is it needed?
- Where can we implement a gateway for a Web database application?

Client-side Web database programming

Major tasks of client-side Web database application programming include the creation of browser extensions and the incorporation of external applications. These types of gateways take advantage of the resources of the client machine, to aid server-side database access. Remember, however, it is advantageous to have a thin client. Thus, the scope of such programming on the client-side should be limited. A very large part of the database application should be on the server side.

Browser extensions

Browser extensions can be created by incorporating script language interpreters to support script languages (e.g. JavaScript), bytecode interpreters to support Java, and dynamic object linkers to support various plug-ins.

JavaScript

JavaScript is a scripting language that allows programmers to create and customise applications on the Internet and intranets. On the client side, it can be used to perform simple data manipulation such as mathematical calculations and form validation. JavaScript code is normally sent as a part of an HTML document and is executed by the browser upon receipt (the browser must have the script language interpreter).

Note that JavaScript has little to do with Java language. JavaScript was originally called LiveScript, but it was changed to benefit from the excitement surrounding Java. The only relationship between JavaScript and Java is a gateway between the former and Java applets (Web applications written in Java).

JavaScript provides developers with a simple way to access certain properties and methods of Java applets on the same page, without having to understand or modify the Java source code of the applet.

Connection to databases

As a database gateway, JavaScript on the client side does not offer much without the aid of a complementary approach such as Java, plug-ins and CGI (Common Gateway Interface, to be discussed later). For example:

- If a Java applet on a page of HTML has access to a database, a programmer can write JavaScript code using LiveConnect to manipulate the applet.
- If there is a form on the HTML document and if an action parameter for that form refers to a CGI program that has access to a database, a programmer can write JavaScript code to manipulate the data elements within the form and then submit it (i.e. submit a kind of request to a DBMS).

Performance

JavaScript can improve the performance of a Web database application if it is used for client-side state management. It can eliminate the need to transfer state data repeatedly between the browser and the Web server. Instead of sending an HTTP request each time it updates an application state, it sends the state only once as the final action. However, there are some side effects resulting from this gain in performance. For example, it may result in the application becoming less robust if state management is completely on the client side. If the client accidentally or deliberately exits, the session state is lost.

Java

As mentioned earlier, Java applets can be manipulated by JavaScript functions to access databases. In general, Java applets can be downloaded into a browser and executed on the client side (the browser should have the bytecode interpreter). The connection to the database is made through appropriate APIs (Application Programming Interface, such as JDBC and ODBC). We will discuss the details in the next section: Server-Side Web Database Programming.

ActiveX

ActiveX is a way to extend Microsoft IE's (Internet Explorer) capabilities. An ActiveX control is a component on the browser that adds functionality which cannot be obtained in HTML, such as access to a file on the client side, other applications, complex user interfaces, and additional hardware devices. ActiveX is similar to Microsoft OLE (Object Linking and Embedding), and ActiveX controls can be developed by any organisation and individual. At the present,

more than one thousand ActiveX controls, including controls for database access, are available for developers to incorporate into Web applications.

Connection to databases

A number of commercial ActiveX controls offer database connectivity. Because ActiveX has abilities similar to OLE, it supports most or all the functionality available to any Windows program.

Performance

Like JavaScript, ActiveX can aid in minimising network traffic. In many cases, this technique results in improved performance. ActiveX can also offer rich GUIs. The more flexible interface, executed entirely on the client side, makes operations more efficient for users.

Plug-ins

Plug-ins are Dynamic Link Libraries (DLL) that give browsers additional functionality. Plug-ins can be installed to run seamlessly inside the browser window, transparent to the user. They have full access to the client's resources, because they are simply programs that run in an intimate symbiosis with the Web browser.

To create a plug-in, the developer writes an application using the plug-in API and native calls. The code is then compiled as a DLL. Installing a plug-in is just a matter of copying the DLL into the directory where the browser looks for plug-ins. The next time that the browser is run, the MIME type(s) that the new plug-in supports will be opened with the plug-in. One plug-in may support multiple MIME types.

There are a number of important issues concerning plug-ins:

- Plug-ins incur installation requirements. Because they are native code, not packaged with the browser itself, plug-ins must be installed on the client machine.
- Plug-ins are platform dependent. Whenever a change is made, it must be made on all supported platforms.

Connection to databases

Plug-ins can operate like any stand-alone applications on the client side. They can be used to create direct socket connections to databases via the DBMS net protocols (such as SQL *Net for Oracle). Plug-ins can also use JDBC, ODBC, OLE and any other methods to connect to databases.

Performance

Plug-ins are loaded on demand. When a user starts up a browser, the installed plug-ins are registered with the browser along with their supported MIME types,

but the plug-ins themselves are not loaded. When a plug-in for a particular MIME type is requested, the code is then loaded into memory. Because plug-ins use native code, their executions are fast.

External applications

External helper applications can be new or legacy database clients, or a terminal emulator. If there are existing traditional client-server database applications which reside on the same machine as the browser, then they can be launched by the browser and execute as usual.

This approach may be an appropriate interim solution for migrating from an existing client-server application to a purely Web-based one. It is straightforward to configure the browser to launch existing applications. It just involves the registration of a new MIME type and the associated application name. For organisations that cannot yet afford the time and funds needed to transfer existing database applications to the Web, launching legacy applications from the browser provides a first step that requires little work.

Maintenance issues

Using the external applications approach, the existing database applications need not be changed. However, it means that all the maintenance burdens associated with traditional client-server applications will remain. Any change to the external application will require a very costly reinstallation on all client machines. Because this is not a pure Web-based solution, many advantages offered by Web-based applications cannot be realised.

Performance

Traditional client-server database applications usually offer good performance. They do not incur the overhead of requiring repeated connections to the database. External database clients can make one connection to the remote database and use that connection for as many transactions as necessary for the session, closing it only when finished.

Review question 4

What are the major tasks involved in client-side Web database programming?

Server-side Web database programming

CGI (Common Gateway Interface)

CGI is a protocol for allowing Web browsers to communicate with Web servers, such as sending data to the servers. Upon receiving the data, the Web server can then pass them to a specified external program (residing on the server

host machine) via environment variables or standard input stream (STDIN). The external program is called a CGI program or CGI script. Because CGI is a protocol, not a library of functions written specifically for any particular Web server, CGI programs/scripts are language independent. As long as the program/script conforms to the specification of the CGI protocol, it can be written in any language such as C, C++ or Java. In short, CGI is the protocol governing communications among browsers, servers and CGI programs.

In general, a Web server is only able to send documents and to tell a browser what kinds of documents it is sending. By using CGI, the server can also launch external programs (i.e. CGI programs). When the server recognises that a URL points to a file, it returns the contents of that file. When the URL points to a CGI program, the server will execute it and then send back the output of the program's execution to the browser as if it were a file.

Before the server launches a CGI program, it prepares a number of environment variables representing the current state of the server which is requesting the action. The program collects this information and reads STDIN. It then carries out the necessary processing and writes its output to STDOUT (the standard output stream). In particular, the program must send the MIME header information prior to the main body of the output. This header information specifies the type of the output.

Refer to the figure under the Basic Concepts section. The CGI approach enables access to databases from the browser. The Web client can invoke a CGI program/script via a browser, and then the program performs the required action and accesses the database via the gateway. The outcome of accessing the database is then returned to the client via the Web server. Invoking and executing CGI programs from a Web browser is mostly transparent to the user. The following steps need to be taken in order for a CGI program to execute successfully:

- The user (Web client) calls the CGI program by clicking on a link or by pressing a button. The program can also be invoked when the browser loads an HTML document (hence being able to create a dynamic Web page).
- The browser contacts the Web server, asking for permission to run the CGI program.
- The server checks the configuration and access files to ensure that the program exists and the client has access authorisation to the program.
- The server prepares the environment variables and launches the program.
- The program executes and reads the environment variables and STDIN.
- The program sends the appropriate MIME headers to STDOUT, followed by the remainder of the output, and terminates.

- The server sends the data in STDOUT (i.e. the output from the program's execution) to the browser and closes the connection.
- The browser displays the information received from the server.

As mentioned earlier, when preparing data for the browser to display, the CGI program has to include a header as the first line of output. It specifies how the browser should display the output. This header may be one of the following types:

Header	Type of output (document)
Content-type: text/html	an HTML document
Content-type: text/plain	ordinary text
Content-type: image/gif	a GIF file (Graphics Interchange Format)
Content-type: image/jpeg	a JPEG file (Joint Photographic Experts Group)
Content-type: image/png	a Portable Network Graph
Content-type: application/postscript	postscript document
Content-type: video/avi Microsoft Audio	Visual Interleave file
Content-type: video/mov	Apple QuickTime Movie file
Content-type: video/mpeg	Moving Picture Experts Group file

Primarily, there are four methods available for passing information from the browser to a CGI program. In this way, clients' input (representing users' specific requirements) can be transmitted to the program for actions.

1. Passing parameters on the command line.
2. Passing environment variables to CGI programs.
3. Passing data to CGI programs via STDIN.
4. Using extra path information.

Detailed discussions on these methods are beyond the scope of this chapter. Please refer to any book dealing specifically with the CGI topic.

Advantages and disadvantages of CGI

Advantages

CGI is the de facto standard for interfacing Web clients and servers with external applications, and is arguably the most commonly adopted approach for interfacing Web applications to data sources (such as databases). The main advantages of CGI are its simplicity, language independence, Web server independence and its wide acceptance.

Disadvantages

The first notable drawback of CGI is that the communication between a client (browser) and the database server must always go through the Web server in the middle, which may cause a bottleneck if there is a large number of users accessing the Web server simultaneously. For every request submitted by a Web client or every response delivered by the database server, the Web server has to convert data from or to an HTML document. This incurs a significant overhead to query processing.

The second disadvantage of CGI is the lack of efficiency and transaction support in a CGI program. For every query submitted through CGI, the database server has to perform the same logon and logout procedure, even for subsequent queries submitted by the same user. The CGI program could handle queries in batch mode, but then support for online database transactions that contain multiple interactive queries would be difficult.

The third major shortcoming of CGI is due to the fact that the server has to generate a new process or thread for each CGI program. For a popular site (like Yahoo), there can easily be hundreds or even thousands of processes competing for memory, disk and processor time. This situation can incur significant overhead.

Last but not least, extra measures have to be taken to ensure server security. CGI itself does not provide any security measures, and therefore developers of CGI programs must be security conscious. Any request for unauthorised action must be spotted and stopped.

Extended CGI

As discussed in the previous section, one of the major concerns with CGI is its performance. With CGI, a process is spawned on the server each time a request is made for a CGI program. There is no method for keeping a spawned process alive between successive requests, even if they are made by the same user. Furthermore, CGI does not inherently support distributed processing, nor does it provide any mechanism for sharing commonly used data or functionality among active and future CGI requests. Any data that exists in one instance of a CGI program cannot be accessed by another instance of the same program.

In order to overcome these problems, an improved version of CGI, called FastCGI, has been developed with the following features:

- Language independence: As with CGI, FastCGI is a protocol and not dependent on any specific language.
- Open standard: Like CGI, FastCGI is positioned as an open standard. It can be implemented by anyone. The specifications, documentation and source code (in different languages) can be obtained at the Web site https://soramimi.jp/fastcgi/fastcgispec.html.

- Independence from the Web server architecture: A FastCGI application need not be modified when an existing Web server architecture changes. As long as the new architecture supports the FastCGI protocol, the application will continue to work.
- Distributed computing: FastCGI allows the Web application to be run on a different machine from the Web server. In this way, the hardware can be tuned optimally for the software.
- Multiple, extensible roles: In addition to the functionality offered by CGI (i.e. receiving data and returning responses), FastCGI can fill multiple roles such as a filter role and an authoriser role. A FastCGI application can filter a requested file before sending it to the client; the authoriser program can make an access control decision for a request, such as looking up a username and password pair in a database. If more roles are needed, more definitions and FastCGI programs can be written to fulfil them.
- Memory sharing: In some cases, a Web application might need to refer to a file on disk. Under CGI, the file would have to be read into the memory space of that particular instance of the CGI program; if the CGI program were accessed by multiple users simultaneously, the file would be loaded and duplicated into different memory locations. With FastCGI, different instances of the same application can access the same file from the same section of memory without duplication. This approach improves performance.
- Allocating processes: FastCGI applications do not require the Web server to start a new process for each application instance. Instead, a certain number of processes are allotted to the FastCGI application. The number of processes dedicated for an application is user-definable. These processes can be initiated when the Web server is started or on demand.

FastCGI seems to be a complete solution for Web database programming, as it includes the best features of CGI and server APIs. In the following sections, a number of CGI-alternative approaches are discussed.

HTTP server APIs and server modules

HTTP server (Web server) APIs and modules are the server equivalent of browser extensions. The central theme of Web database sites created with HTTP server APIs or modules is that the database access programs coexist with the server. They share the address space and run-time process of the server. This approach is in direct contrast to the architecture of CGI, in which CGI programs run as separate processes and in separate memory spaces from the HTTP server.

Instead of creating a separate process for each CGI program, the API offers a way to create an interface between the server and the external programs using

dynamic linking or shared objects. Programs are loaded as part of the server, giving them full access to all the I/O functions of the server. In addition, only one copy of the program is loaded and shared among multiple requests to the server.

Server vendor modules

Server modules are just prefabricated applications written in some server APIs. Developers can often purchase commercial modules to aid or replace the development of an application feature. Sometimes, the functionality required in a Web database application can be found as an existing server module.

Vendors of Web servers usually provide proprietary server modules to support their products. There are a very large number of server modules that are commercially available, and the number is still rising. For example, Oracle provides the Oracle PL/SQL module, which contains procedures to drive database-backed Web sites. The Oracle module supports both NSAPI and ISAPI.

Advantages of server APIs and modules

Having database access programs coexist with the HTTP server improves Web database access due to improved speed, resource sharing, and the range of functionality.

- **Server speed**

  API programs run as dynamically loaded libraries or modules. A server API program is usually loaded the first time the resource is requested, and therefore, only the first user who requests that program will incur the overhead of loading the dynamic libraries. Alternatively, the server can force this first instantiation so that no user will incur the loading overhead. This technique is called preloading. Either way, the API approach is more efficient than CGI.

- **Resource sharing**

  Unlike a CGI program, a server API program shares address space with other instances of itself and with the HTTP server. This means that any common data required by the different threads and instances need exist only in one place. This common storage area can be accessed by concurrent and separate instances of the server API program.

  The same principle applies to common functions and code. The same set of functions and code are loaded just once and can be shared by multiple server API programs. The above techniques save space and improve performance.

- **Range of functionality**

A CGI program has access to a Web transaction only at certain limited points. It has no control over the HTTP authentication scheme. It has no contact with the inner workings of the HTTP server, because a CGI program is considered external to the server.

In contrast, server API programs are closely linked to the server; they exist in conjunction with or as part of the server. They can customise the authentication method as well as transmission encryption methods. Server API programs can also customise the way access logging is performed, providing more detailed transaction logs than are available by default.

Overall, server APIs provide a very flexible and powerful solution to extending the capabilities of Web servers. However, this approach is much more complex than CGI, requiring specialised programmers with a deep understanding of the Web server and sophisticated programming skills.

Important issues

- **Server architecture dependence**

  Server APIs are closely tied to the server they work with. The only way to provide efficient cross-server support is for vendors to adhere to the same API standard. If a common API standard is used, programs written for one server will work just as well with another server. However, setting up standards involves compromises among competitors. In many cases, they are hard to come by.

- **Platform dependence**

  Server APIs and modules are also dependent on computing platforms. Some servers are supported on multiple platforms. Nevertheless, each supporting version is dependent on that platform. Similarly, the Microsoft server is only available for various versions of Windows.

- **Programming language**

  Most Web servers can be extended using a variety of programming languages and facilities. In addition, Microsoft provides an application environment called Active Server Pages. Active Server Pages is an open, compile-free application environment in which developers can combine HTML, scripts and reusable ActiveX server components to create dynamic and powerful Web-based business solutions.

Comparison of CGI, server APIs and modules, and FastCGI

The following table provides a straightforward comparison among approaches of CGI, server APIs and modules, and FastCGI:

Features	CGI	APIs and Modules	APIs and Modules
Language-independent	Yes		Yes
Runs in different process from core Web server	Yes		Yes
Open standard	Yes		Yes
Web server architecture-independent	Yes		Yes
Supports distributed computing		Yes	Yes
Multiple, extensible roles		Yes	Yes
Memory sharing with other processes		Yes	Yes
Does NOT create new process for each instance		Yes	Yes
Easy to use	Yes		Yes

Proprietary HTTP servers

A proprietary HTTP server is defined as a server application that handles HTTP requests and provides additional functionality that is not standard or common among available HTTP servers. The functionality includes access to a particular database or data source, and translation from a legacy application environment to the Web.

Examples of proprietary servers include IBM Domino, Oracle Application Express Listener and Hyper-G. These products were created for specific needs. For Domino, the need is tight integration with legacy Lotus Notes applications, allowing them to be served over the Web. Oracle Application Express Listener was designed to provide highly efficient and integrated access to back-end Oracle databases. For Hyper-G, the need is to have easily maintainable Web sites with automatic link update capabilities.

The main objectives of creating proprietary servers are to meet specialised and customised needs, and to optimise performance. However, the benefits of proprietary servers must be carefully weighed against their exclusive ties to a Web database product (which may bring many shortcomings). It requires a thorough understanding of the business requirements in order to determine whether or not a proprietary Web server is appropriate in a project.

Review question 5

- What is CGI?
- What are the typical steps in the procedure by a Web client of invoking a CGI program?
- What are Web server APIs and server modules?
- Compare the features of CGI, FastCGI, and server APIs and modules.

Connecting to the database

In previous sections, we have studied various approaches that enable browsers (Web clients) to communicate with Web servers, and in turn allow Web clients to have access to databases. For example, CGI, FastCGI or API programs can be invoked by the Web client to access the underlying database. In this section, we are going to discuss how database connections can actually be made via those CGI/FastCGI/API programs. We will learn what specific techniques, tools and languages are available for making the connections. In short, we will see how the database connection layer is built for the underlying database.

In general, database connectivity solutions include the use of:

- Native database APIs
- Database-independent APIs
- Template-driven database access packages
- Third-party class libraries

Do not be confused with the concepts of Web server APIs and database APIs. Web server APIs are used to write server applications, in which database APIs are used specifically for connecting to and accessing the database. Also, database APIs can be used to write a CGI program which allows developers to create a Web application with a database back end. Similarly, a template-driven database access package, along with a program written in a Web server's API (e.g. NSAPI, ISAPI), is another way to link a Web front end to a database back end.

Database API libraries

Before we look at specific API database connectivity solutions, let's give background to database API libraries.

Database API libraries are at the core of every Web database application and gateway. Regardless how a Web database application is built (whether by manually coding CGI programs or by using a visual application builder), database API libraries are the foundation of database access.

The approach

Database API libraries are collections of functions or object classes that provide source code access to databases. They offer a method of connecting to the database engine (under a username and password if user authentication is supported by the DBMS), sending queries across the connection, and retrieving the results and/or error messages in a desired format.

Traditional client-server database applications have already employed database connectivity libraries supplied by vendors and third-party software companies (i.e., third party class libraries). Because of this fact of wider user base, database APIs have the advantage over other gateway solutions for Web database connectivity.

The Web database applications that require developers to use database API libraries are mainly CGI, FastCGI or server API programs. Web database application building tools, including template-driven database access packages and visual GUI builders, use database APIs as well as the supporting gateways (such as CGI and server API), but all these interactivities are hidden from the developers.

Native database APIs

Native database APIs are platform-dependent as well as programming language dependent. However, most popular databases (such as Oracle) support multiple platforms in the first place, and therefore, the porting between different platforms should not require excessive effort.

In general, programs that use native database APIs are faster than those using other methods, because the libraries provide direct and low-level access. Other database access methods tend to be slower, because they add another layer of programming to provide the developer a different, easier, or more customised programming interface. These additional layers slow the overall transaction down.

Native database API programming is not inherently dependent on a Web server. For example, a CGI program using native API calls to Oracle that works with the Netscape server should also work with other types of servers. However, if the CGI program also incorporates Web server-specific functions or modules, it will be dependent on that Web server.

Database-independent APIs: ODBC

The most popular standard database-independent API was pioneered by Microsoft. It is called ODBC (Open Database Connectivity) and is supported by all of the most popular databases such as Microsoft Access and Oracle.

ODBC requires a database-specific driver or client to be loaded on the database client machine. In a Java application that accesses Oracle, for example, the server that hosts the Java application would need to have an Oracle ODBC client installed. This client would allow the Java application to connect to the ODBC data source (the actual database) without knowing anything about the Oracle database.

In addition to the database-specific ODBC driver being installed on the client machine, Java requires that a JDBC-ODBC bridge (i.e. another driver. JDBC stands for Java Database Connectivity) be present on the client machine. This JDBC-ODBC driver translates JDBC to ODBC and vice versa, so that Java programs can access ODBC-compliant data sources but still use their own JDBC class library structure.

Having the database-specific ODBC driver on the client machine dictates that Web database Java applications or applets using ODBC be 3-tiered. The database client of the Web application must reside on a server: either the same server as the Web server or a remote server. Otherwise, the database-specific ODBC driver would have to exist on every user's computer, which is a very undesirable situation. The diagram below provides a graphical illustration of such an architecture.

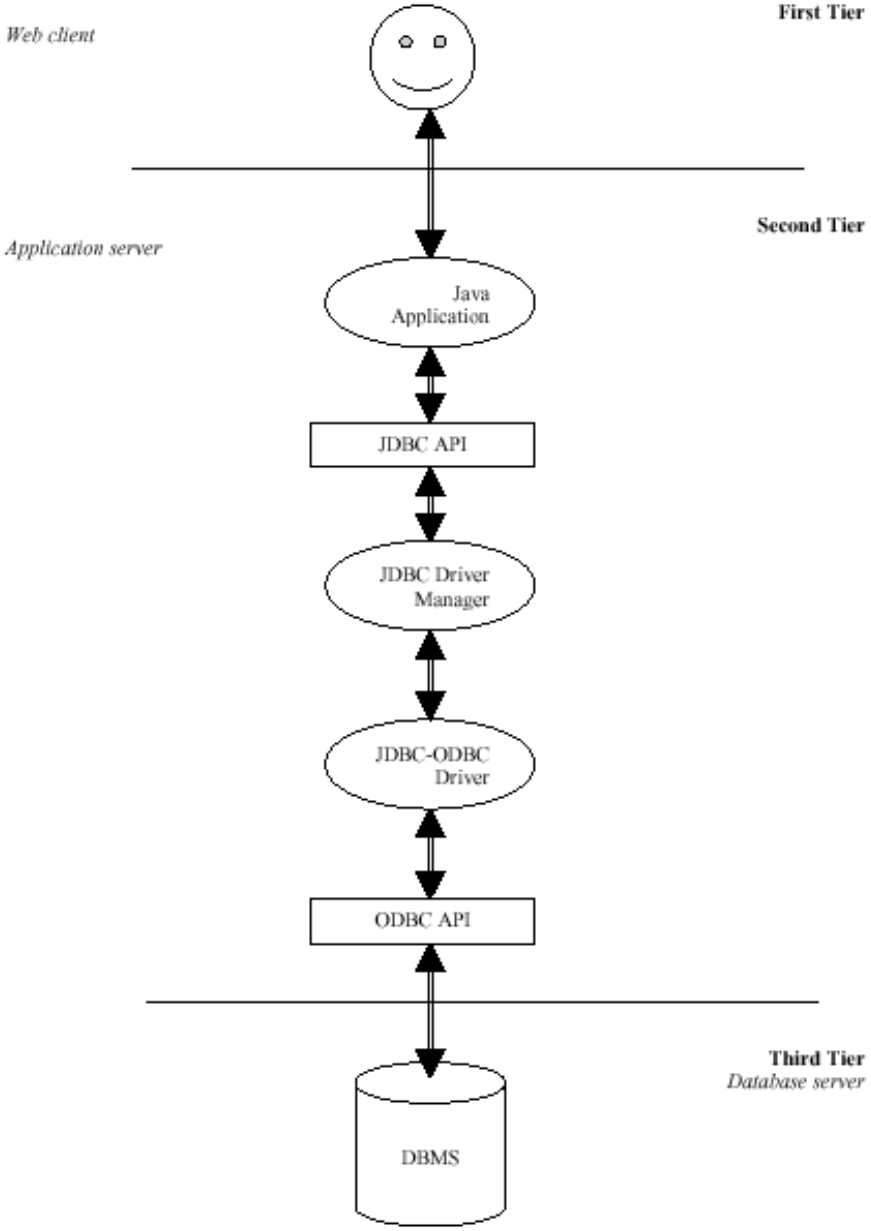
Benefits of database APIs

Database APIs (native or independent) arguably offer the most flexible way in which Web database applications are created. Applications created with native database APIs are more efficient than those with database-independent APIs. This database connectivity solution is the fastest way to access database functionality and has been tested rigorously in the database software industry. It is worth noting that database APIs have been used successfully for years even before the invention of the Web.

Shortcomings of database APIs

The most notable disadvantage of programming in database API is complexity. For rapid application development and prototyping, it is better to use a high-level tool, such as template-driven database access software or visual application builders.

Another disadvantage is with ODBC. Because ODBC standardises access to databases from multiple vendors, applications using ODBC do not have access to native SQL database calls that are not supported by the ODBC standard. In some cases, this can be inconvenient and may even affect application performance.


First Tier
Web client
Second Tier
Application server
Java Application
JDBC API
JDBC Driver Manager
JDBC-ODBC Driver
ODBC API
Third Tier
Database server
DBMS

Template-driven packages

The approach

Template-driven database connectivity packages are offered by database vendors and third-party developers to simplify Web database application programming. Such a package usually consists of the following components:

- Template consisting of HTML and non-standard tags or directives
- Template parser
- Database client daemons

Template-driven packages are very product dependent. Different DBMSs require database access templates in different formats. An application developed for one product will be strongly tied to it. Migrating from one product to another is very difficult and requires a rewrite of all the database access, flow control and output-formatting commands.

An example of a template-driven package is PHP.

Benefits of template-driven packages

The most important benefit from using a template-driven package is speed of development. Assuming an available package has been installed and configured properly, it takes as little time as a few hours to create a Web site that displays information directly from the database.

Shortcomings of template-driven packages

The structures of templates are normally predetermined by vendors or third-party developers. As a result, they only offer a limited range of flexibility and customisability. Package vendors provide what they feel is important functionality, but, as with most off-the-shelf tools, such software packages may not let you create applications requiring complex operations.

Although templates offer a rapid path to prototyping and developing simple Web database applications, the ease of development is obtained for the cost of speed and efficiency. Because the templates must be processed on demand and require heavy string manipulation (templates are of a large text type or string type; they must be parsed by the parser), using them is slow compared with using direct access such as native database APIs.

The actual performance of an application should be tested and evaluated before the usefulness of such a package is ruled out. The overhead of parsing templates may be negligible if using high-performance machines. Other factors, such as development time or development expertise, may be more important than a higher operational speed.

GUI application builders

Visual Web database building tools offer an interesting development environment for creating Web database applications. For developers accustomed to point-and-click application programming, these tools help speed the development process. For instance, Visual Basic and/or Microsoft Access developers should find such a tool intuitive and easy to use.

The approach

The architectures of visual building tools vary. In general, they include a user-friendly GUI (Graphical User Interface), allowing developers to build a Web database application with a series of mouse clicks and some textual input. These tools also offer application management so that a developer no longer needs to juggle multiple HTML documents and CGI, NSAPI or ISAPI programs manually.

At the end of a building session, the tool package can generate applications using various techniques. Some applications are coded using ODBC; some use native database APIs for the databases they support; and others may use database net protocols.

Some of these tools create their own API, which can be used by other developers. Some generate code that works but can still be modified and customised by developers using various traditional IDEs, compilers and debuggers.

A building tool may generate a CGI program or a Web server API program (such as NSAPI and ISAPI). Some sophisticated tools even offer all the options. The developer can choose what he/she wants.

Unlike native database APIs or template-driven database connectivity packages, visual Web database development tools tend to be as open as possible. Many offer development support for the popular databases.

Benefits of visual tools

Visual development tools can be of great assistance to developers who are familiar and comfortable with visual application development techniques. They offer rapid application development and prototyping, and an organised way to manage the application components. Visual tools also shield the developer from low-level details of Web database application development. As a result, a developer can create a useful Web application without the need to know what is happening in the code levels.

Shortcomings of visual tools

Depending on the sophistication of the package used, the resulting programs may be slower to execute than similar programs coded by an experienced programmer. Visual application building tools, particularly Object-oriented ones, tend to generate fat programs with a lot of unnecessary sub-classing.

Another potential drawback is cost. A good visual tool may be too expensive for a small one-off development budget.

Review question 6

- What are database APIs? Who uses them and why?
- Why are template-driven packages useful for building database connections? What are the shortcomings?
- How can we benefit from using visual development tools to build database connections?

Managing state and persistence in Web applications

State is an abstract concept of being, which can be explained by a set of rules, facts or truisms. A state in a database application includes a set of variables and/or other means to record who the user/client is, what tasks he/she has been doing, at what position he/she is at a particular instance in time, and many other useful pieces of information about a database session. Persistence is the capability of remembering a state and tracking state changes across different applications or different periods of time within an instance of an application or multiple instances.

The requirement of state maintenance in Web database applications results in the increased complexity. As mentioned before in the Context section, HTTP is connectionless, which means that once an HTTP request is sent and a response is received, the connection to the server is closed. If a connection were to be kept open between client and server, the server could at any time query the client for state information and vice versa. The server would be able to know the identity of the user throughout the session once the user logged in. However, the reality is that there is no constant connection throughout the session. Thus, the server cannot have memory of the user's identity even after user login. In this situation, programmers must find a way to make session state persist.

Technical options

There are several options available to programmers to maintain state. They range from open systems options defined in HTTP and CGI standards, to proprietary mechanisms written from scratch.

The most important task in maintaining persistence is to keep track of the identity of the user. If the identity can persist, any other data/information can usually be made to persist in exactly the same manner.

The URL approach

It works as follows:

- A registration or login page is delivered to the user.
- The user types in a username and password, and then submits the page.
- The username and password pair are sent to a server-side CGI program, which extracts the values from the QUERY_STRING environment variable.
- The values are checked by the server to determine whether or not the user is authenticated.
- If he/she is authenticated, the authenticated state is reflected in a randomly generated session ID (SID), which is stored in a database along with other necessary data to describe the state of the user session.
- The SID can then be stored in all URLs within HTML documents returned by the server to the client, therefore tracking the identity of the user throughout the session.

Benefits of the URL approach

The URL approach is easy to use to maintain state. To retrieve a state, the receiving CGI program need only collect the data from environment variables in the GET method and act on it as necessary. To pass on, set or change the state, the program simply creates new URLs with the appropriate data.

Shortcomings of the URL approach

If the state information has to be kept in the URL, the URL becomes very long and can be very messy. Also, such a URL displays part of the application code and low-level details. This causes security concerns, and may be used by hackers.

If an application manages state on the client side using the URL method, the state will be lost when the user quits the browser session unless the user bookmarks the URL. A bookmark saves the URL in the browser for future retrieval. If state is maintained solely in the URL without any server-side state data management, bookmarking is sufficient to recreate the state in a new browser session. However, having the user perform this maintenance task is obviously undesirable.

URL QUERY_STRING

This is another popular method of maintaining state. A registered user in a site has a hidden form appended to each page visited within the site. This form contains the username and the name of the current page. When the user moves from one page to another, the hidden form moves as well and is appended to the end of the succeeding HTML page.

Benefits of the hidden fields approach

Like the URL approach, it is easy to use to maintain state. In addition, because the fields are hidden, the user has a seamless experience and sees a clean URL.

Another advantage of using this approach is that, unlike using URLs, there is no limit on the size of data that can be stored.

Shortcomings of the hidden fields approach

As with the URL approach, users can fake states by editing their own version of the HTML hidden fields. They can bring up the document source in an editor, change the data stored, and then submit the tampered form to the server. This raises serious security concerns.

Data is also lost between sessions. If the entire session state is stored in hidden fields, that state will not be accessible after the user exits the browser unless the user specifically saves the HTML document to disk or with a bookmark. Again, it is undesirable to involve users in this kind of maintenance task.

HTTP cookies

An HTTP cookie is a technique that helps maintain state in Web applications. A cookie is in fact a small text file containing:

- Name of the cookie
- Domains for which the cookie is valid
- Expiration time in GMT
- Application-specific data such as user information

Cookies are sent by the server to the browser, and saved to the client's disk. Whenever necessary, the server can request a desired cookie from the client. The client browser will check whether it has it. If it does, the browser will send the information stored in the cookie to the server.

Benefits of cookies

Cookies can be completely transparent. As long as a user does not choose the browser option to be alerted before accepting cookies, his/her browser will handle incoming cookies and place them on the client disk without user intervention.

Cookies are stored in a separate file, whose location is handled by the browser and difficult for the user to find. Also, cookies are difficult to tamper with. This increases security.

Because cookies are stored on the client disk, the cookie data is accessible even in a new browser session. It does not require theuser to do anything.

If a programmer chooses to set an expiration date or time for a cookie, the browser will invalidate the cookie at the appropriate time.

Shortcomings of cookies

The amount of data that can be stored with a cookie is usually limited to 4 kilobytes. If an application has very large state data, other techniques must be considered.

Because cookies are physically stored on the client disk, they cannot move with the user. This side effect is important for applications whose users often change machines.

Although cookies are difficult to tamper with, it is still possible for someone to break into them. Remember a cookie is just a text file. If a user can find it and edit it, it can still cause security problems.

Important considerations

Managing state on the client

An application can maintain all of its state on the client-side with any of the methods discussed in the previous section.

- **Benefits of the client-side maintenance**

  On attraction of maintaining state on the client is simplicity. It is easier to keep all the data in one place, and by doing it on the client, it eliminates the need for server database programming and maintenance.

  If an application uses client-side extensions to maintain state, it can also provide a faster response to the user because the need to network access is eliminated.

- **Shortcomings of the client-side maintenance**

If all the state data is on the client-side, there is a danger that users can somehow forge state information by editing URLs, hidden fields, and cookies. This leads to security risks in server programs.

With the exception of the cookie approach to maintaining state, there is no guarantee that the necessary data will be saved when the client exits unexpectedly. Thus, the robustness of the application is compromised.

Managing state on the server

This approach for maintaining state actually involves using both the client and the server. Usually a small piece of information, either a user ID or a session key is stored on the client-side. The server program uses this ID or key to look up the state data in a database.

- **Benefits of the server-side maintenance**

  Maintaining state on the server is more reliable and robust than the client-side maintenance. As long as the client can provide an ID or a key, the user's session state can be restored, even between different browsing sessions.

  Server-side maintenance can result in thin clients. The less dependent a Web database application is on the client, the less code needs to exist on or be transmitted to the client.

  Server-side maintenance also leads to better network efficiency, because only small amounts of data need to be transmitted between the client and the server.

- **Shortcomings of the server-side maintenance**

  The main reason an application would not be developed using server-side state maintenance is its complexity, because it requires the developer to write extensive code. However, the benefits of implementing server-side state management outweigh the additional work required.

Review question 7

- What is state and persistence management in Web database applications?
- What are the technical options available for managing state and persistence?

Security Issues in Web Database Applications

Security risks exist in many areas of a Web database application. This is because the very foundations of the Internet and Web – TCP/IP and HTTP – are very weak with respect to securities. Without special software, all Internet traffic travels in the open and anyone with a little bit skill can intercept data

transmission on the Internet. If no measures are taken, there will be many security loopholes that can be explored by malicious users on the Internet.

In general, security issues in Web database applications include the following:

- Data transmission (communication) between the client and the server is not accessible to anyone else except the sender and intended receiver (privacy).
- Data cannot be changed during transmission (integrity).
- The receiver can be sure that the data is from the authenticated sender (authenticity).
- The sender can be sure the receiver is the genuinely intended one (non-fabrication).
- The sender cannot deny he/she sent it (non-repudiation).
- The request from the client should not ask the server to perform illegal or unauthorised actions.
- The data transmitted to the client machine from the server must not be allowed to contain executables that will perform malicious actions.

At the present, there are a number of measures that can be taken to address some of the above issues. These measures are not perfect in the sense that they cannot cover every eventuality, but they should help get rid of some of the loopholes. It must be stressed that security is the most important but least understood aspect of Web database programming. More work still needs to be done to enhance security.

Proxy servers

A proxy server is a system that resides between a Web browser and a Web server. It intercepts all requests to the Web server to determine if it can fulfil the requests itself. If not, it forwards the requests to the Web server.

Due to the fact that the proxy server is between browsers and the Web server, it can be utilised to be a defence for the server.

Firewalls

Because a Web server is open for access by anyone on the Internet, it is normally advised that the server should not be connected to the intranet (i.e., an organisation's internal network). This way, no one can have access to the intranet via the Web server.

However, if a Web application has to use a database on the intranet, then the firewall approach can be used to prevent unauthorised access.

A firewall is a system designed to prevent unauthorised access to or from a private network (intranet). It can be implemented in either hardware, software, or both. All data entering or leaving the intranet (connected to the Internet) must pass through the firewall. They are checked by the firewall system and anything that does not meet the specified security criteria is blocked.

A proxy server can act as a firewall because it intercepts all data in and out, and can also hide the address of the server and intranet.

Digital signatures

A digital signature consists of two pieces of information: a string of bits that is computed from the data (message) that is being signed along with the private key of the requester for the signature.

The signature can be used to verify that the data is from a particular individual or organisation. It has the following properties:

- Its authenticity is verifiable using a computation based on a corresponding public key.
- If the private key is kept secret, the signature cannot be forged.
- It is unique for the data signed. The computation will not produce the same result for two different messages.
- The signed data cannot be changed, otherwise the signature will no longer verify the data as being authentic.

The digital signature technique is very useful for verifying authenticity and maintaining integrity.

Digital certificates

A digital certificate is an attachment to a message used for verifying the sender's authenticity. Such a certificate is obtained from a Certificate Authority (CA), which must be a trust-worthy organisation.

When a user wants to send a message, he/she can apply for a digital certificate from the CA. The CA issues an encrypted certificate containing the applicant's public key and other identification information. The CA makes its own key publicly available.

When the message is received, the recipient uses the CA's public key to decode the digital certificate attached to the message, verifies it as issued by the CA, and then obtains the sender's public key and identification information held within the certificate. With this information, the recipient can send an encrypted reply.

Kerberos

Kerberos is a server of secured usernames and passwords. It provides one centralised security server for all data and resources on the network. Database access, login, authorisation control, and other security measures are centralised on trusted Kerberos servers. The main function is to identify and validate a user.

Secure sockets layer (SSL) and secure HTTP (S-HTTP)

SSL is an encryption protocol developed by Netscape for transmitting private documents over the Internet. It works by using a private key to encrypt data that is to be transferred over the SSL connection. Netscape, Firefox, Chrome and Microsoft IE support SSL.

Another protocol for transmitting data securely over the Internet is called Secure HTTP, a modified version of the standard HTTP protocol. Whereas SSL creates a secure connection between a client and a server, over which any amount of data can be sent securely, S-HTTP is designed to transmit individual messages securely.

In general, the SSL and S-HTTP protocols allow a browser and a server to establish a secure link to transmit information. However, the authenticity of the client (the browser) and the server must be verified. Thus, a key component in the establishment of secure Web sessions using SSL or S-HTTP protocols is the digital certificate. Without authentic and trustworthy certificates, the protocols offer no security at all.

Java security

If Java is used to write the Web database application, then many security measures can be implemented within Java. Three Java components can be utilised for security purposes:

- The class loader: It not only loads each required class and checks it is in the correct format, but also ensures that the application/applet does not violate system security by allocating a namespace. This technique can effectively define security levels for each class and ensure that a class with a lower security clearance can never be in place of a class with a higher clearance.
- The bytecode verifier: Before the Java Virtual Machine (JVM) will allow an application/applet to execute, its code must be verified to ensure: compiled code is correctly formatted; internal stacks will not overflow or underflow; no illegal data conversions will occur; bytecode instructions are correctly typed; and all class member accesses are valid.

- The security manager: An application-specific security manager can be defined within a browser, and any applets downloaded by this browser are subject to its (security manager's) security policies. This can prevent a client from being attacked by dangerous methods.

ActiveX security

For Java, security for the client machine is one of the most important design factors. Java applet programming provides as many features as possible without compromising the security of the client. In contrast, ActiveX's security model places the responsibility for the computer's safety on the user (client). Before a browser downloads an ActiveX control that has not been digitally signed or has been certified by an unknown CA, it displays a dialog box warning the user that this action may not be safe. It is up to the user to decide whether to abort the downloading, or continue and accept a potential damaging consequence.

Review question 8

- What are the major security concerns in Web database applications?
- What are the measures that can be taken to address the security concerns?

Performance issues in Web database applications

Web database applications are very complex, more so than stand-alone or traditional client-server applications. They are a hybrid of technology, vendors, programming languages, and development techniques.

Many factors work together in a Web database application and any one of them can hamper the application's performance. It is crucial to understand the potential bottlenecks in the application as well as to know effective, well-tested solutions to address the problems.

The following is a list of issues concerning performance:

- **Network consistency:** The availability and speed of network connections can significantly affect performance.
- **Client and server resources:** This is the same consideration as in the traditional client-server applications. Memory and CPU are the scarce resources.
- **Database performance:** It is concerned with the overhead for establishing connections, database tuning, and SQL query optimisation.
- **Content delivery:** This is concerned with the content's download time and load time. The size of any content should be minimised to reduce download time; and appropriate format should be chosen for a certain

document (mainly images and graphics) so that load time can be minimised.

- **State maintenance:** It should always minimise the amount of data transferred between client and server and minimise the amount of processing necessary to rebuild the application state.
- **Client-side processing:** If some processing can be carried out on the client-side, it should be done so. Transmitting data to the server for processing that can be done on the client-side will degrade performance.
- **Programming language:** A thorough understanding of the tasks at hand and techniques available can help choose the most suitable language for implementing the application.

Discussion topics

In this chapter, we have studied various approaches for constructing the Web database connectivity, including GUI-based development tools. Both Oracle and Microsoft offer visual development tools. Discuss:

1. What are the pros and cons of using GUI-based tools?
2. Do you prefer programming using APIs or visual tools? Why?

Chapter 18. Temporal Databases

Table of contents

- Temporal database design
  - Entity Relationship Time model
- The ERT-SQL language

Almost all database applications are concerned with the modelling and storage of data that varies with time. It is not surprising therefore, that a great deal of research and development, both in industry and in universities, has gone into developing database systems that support the time-related or temporal aspects of data processing. In this chapter we shall examine the major issues in the provision of support for temporal data. We shall explore some of the most important research that has been done in the area, and identify the influence of this research on query languages.

Objectives

At the end of this chapter you should be able to:

- Define and use important temporal concepts, such as time point, time interval, and time-interval operators such as before, after and overlaps.
- Explain the issues involved in modelling a number of time-varying features of data, such as transaction time, valid time and time granularity.
- Understand the temporal data model at the conceptual level.
- Describe some of the extensions to conventional query languages that have been proposed to support temporal query processing.

Introduction

Detailed concepts of temporal databases can be found in the book titled "Time and Relational Theory (Temporal Databases in the Relational Model and SQL), 2nd Edition by C.J. Date, Hugh Darwen and Nikos Lorentzos"

Temporal databases: The complexities of time

A temporal database is generally understood as a database capable of supporting storage and reasoning of time-based data. For example, medical applications may be able to benefit from temporal database support — a record of a patient's medical history has little value unless the test results, e.g. the temperatures, are associated to the times at which they are valid, since we may wish to do reasoning about the periods in time in which the patient's temperature changed.

Temporal databases store temporal data, i.e. data that is time dependent (timevarying). Typical temporal database scenarios and applications include time-dependent/time-varying economic data, such as:

- Share prices
- Exchange rates
- Interest rates
- Company profits

The desire to model such data means that we need to store not only the respective value but also an associated date or a time period for which the value is valid. Typical queries expressed informally might include:

- Give me last month's history of the Dollar-Pound Sterling exchange rate.
- Give me the share prices of the NYSE on October 17, 1996.

Many companies offer products whose prices vary over time. Daytime telephone calls, for example, are usually more expensive than evening or weekend calls. Travel agents, airlines or ferry companies distinguish between high and low seasons. Sports centres offer squash or tennis courts at cheaper rate during the day. Hence, prices are time dependent in these examples. They are typically summarised in tables with prices associated with a time period.

As well as the user of a database recording when certain data values are valid, we may wish to store (for backup, or analysis reasons) historical records of changes made to the database. So each time a change to a database is made the system may automatically store a transaction timestamp. Therefore a temporal database may be storing two different pieces of time data for a tuple — the user-defined period of time for which the data is valid (e.g. October to April [winter season] rental of tennis courts are 1 US dollar per hour), and the system-generated transaction timestamp for when the tuple (or part of a tuple) was changed (e.g. 14:55 on 03/01/1999). A temporal database ability to store these different kinds of data makes possible many different kinds of temporal-based queries, as long as its query language and data model are sophisticated enough to formally model and allow reasoning about temporal data. It is the possible gains from such temporal querying facilities that has provided the motivation for research and development into extending the Relational database model for temporal data (and suggesting alternatives to the Relational model…).

Our everyday life is very often influenced by timetables for buses, trains, flights, university lectures, laboratory access and even cinema, theatre or TV programmes. As one consequence, many people plan their daily activities by using diaries, which themselves are a kind of timetable. Timetables or diaries can be regarded as temporal relations in terms of a Relational temporal data model. Medical diagnosis often draws conclusions from a patient's history, i.e. from the evolution of his/her illness. The latter is described by a series of values, such as the body temperature, cholesterol concentration in the blood, blood pressure, etc. As in the first example, each of these values is only valid during a certain period of time (e.g. a certain day). Typically a doctor would retrieve a patient's values' history, analyse trends and base diagnosis on such observations. Similar

examples can be found in many areas that rely on the observation of evolutionary processes, such as environmental studies, economics and many natural sciences.

Concepts of time

Continuous or discrete

From a philosophical point of view we might argue either that time passes continuously, flowing as if a stream or river of water, or we can think of time passing in discrete units of time, each with equal duration, as we think of time when listening to the ticking of a clock. For the purposes of recording and reasoning about time, many people prefer to work with a conceptual model of time as being discrete and passing in small, measurable units; however, there are some occasions and applications where a continuous model of time is most appropriate.

Granularity

When we think about time as passing in discrete units, depending on the purpose or application, different-sized units may be appropriate. So the age of volcanoes may be measured in years, or decades, or hundreds of years. The age of motor cars may be measured in years, or perhaps months for 'young' cars. The age of babies may be measured in years, or months, or weeks, or days. The age of bacteria in seconds or milliseconds. The size of the units of time used to refer to a particular scenario is referred to as the granularity of the temporal units — small temporal grains refer to short units of time (days, hours, seconds, milliseconds, etc), and large temporal grains refer to longer units of time (months, years, decades, etc).

Time quanta

For a particular situation we may wish to define the smallest unit of time which can be recorded or reasoned about. One way to refer to the chosen, indivisible unit of time is as 'time quanta'. Sometimes the term 'chronon' or 'time granule' is used to refer to the indivisible units of time quanta for a situation or system. In this chapter we shall use the term time quanta.

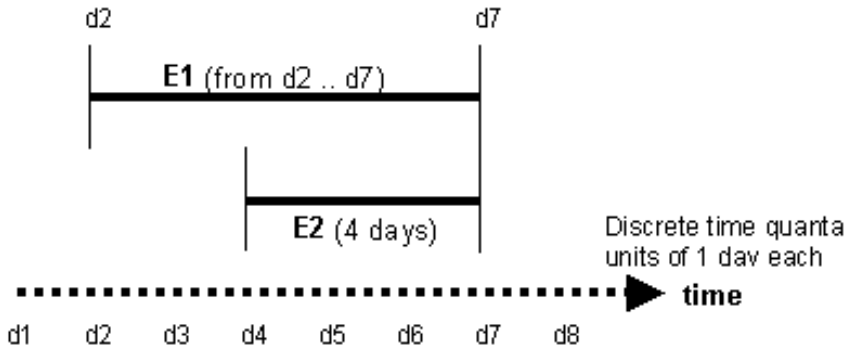
Timelines, points, duration and intervals

When attempting to represent and reason about time, four important concepts are:

- **Points:** Formally a point in time has no duration; it simply refers to a particular position in the timeline under discussion. We can talk of the point in time at which some event begins or ends.

- **Duration:** A duration refers to a number of time quanta; for example, a week, two months, three years and 10 seconds are all durations. A duration refers to a particular magnitude (size) of a part of a timeline, but not the direction (so whether we talk of a week ago or a week beginning in three days' time, we are still referring to a length of time of a week).
- **Interval:** An interval has a start time point and an end time point. Using more formal notation, we can refer to an interval I(s, e) with start point 's' and end point 'e', and for which all points referring to time from s to e (inclusive) make up the interval. Note that there is an assumption (constraint) that the timepoint 's' does not occur after the timepoint 'e' (an interval of zero time quanta would have a start point and end point that were equal).
- **Timeline:** Conceptually we can often imagine time as moving along a line in one direction. When graphically representing time, it is usual to draw a line (often with an arrow to show time direction), where events shown towards the end of the timeline have occurred later than those shown towards the beginning of the line. Often a graphical timeline is draw like an axis on a graph (by convention, the X-axis represents time) and the granularity of the time units is marked (and perhaps labelled) along the X-axis.

You may find the following diagram useful in illustrating these four concepts:



Looking at the figure, there are two events, E1 and E2. We see that event E1 starts at the time point 'd2' and ends at time point 'd7', therefore event E1 is an example of an interval. All that is written for event E2 is that it lasts four days, so event E2 is a duration (although we might attempt to infer from the X-axis that E2 appears to start at d4 and end at d7, but perhaps E2 is four days measured back from d7). The X-axis is a labelled arrow 'time', and represents a timeline. There are units labelled on the X-axis from 'd1' to 'd8', and a note indicates that these units are time quanta representing one day each.

Since the time quanta are a day each for this model, we are not able to model any time units of less that one day (so even if it looked like an event started halfway between two units on the diagram, we could not meaningfully talk of day 1 plus 12 hours or whatever). This does raise an issue when attempting to convert from a model of one time granularity to anothe. For example, if all data from the model above were to be added to a database where the time quanta was in terms of hours, what hour of the day would 'd3' represent? Would it be midnight when the day started, or midday, or 09:00 in the morning (the start of a working day)? Such questions must be answered by any data analyst if converting between different temporal models, which is one reason why it is so important to choose an appropriate granularity of time quanta. It might seem reasonable, just in case, to choose a very small time quanta, such as seconds or milliseconds,but there may be significant memory implications for such a choice (e.g. if the database system had to use twice as much memory to record millisecond timestamps for each attribute of each tuple, rather than just recording the day timestamp).

The important temporal work of Allen (1983)

Much of the recent work in the fields of time-based computing (both for databases and other areas such as artificial intelligence) is based on the work of J. F. Allen. A publication by Allen that is frequently cited in literature about time-based reasoning is:

Maintaining Knowledge about Temporal Intervals, J. F. Allen, CACM (Communications of the Association for Computing Machinery) Volume 16 number 11, November 1983.

Allen's contribution to those wishing to represent and reason about time-based information consisted of the formalisation of possible relationships between pairs of intervals. Although the following, informal overview may seem obvious from an intuitive perspective, Allen presented these concepts in a rigorous, formal way which has been the basis for much temporal reasoning and computer system design since.

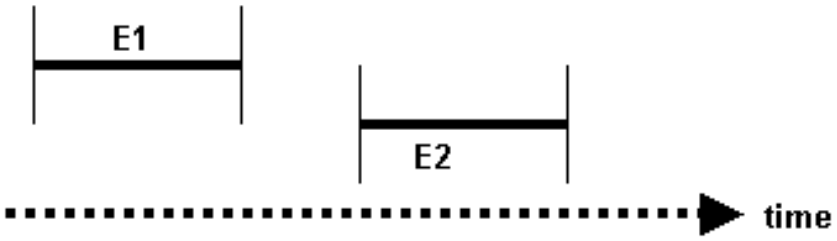
Consider two events, E1 and E2. Each event has a starting point in time and an ending point in time — therefore each event takes place as an interval in time. The following are the possible relationships between two intervals (events):

- E1 starts and ends before E2 begins.
- E1 ends at the same point in time that E2 begins — the two events are temporally contiguous. We can say that the end of E1 meets the beginning of E2.
- E1 starts before E2 starts, but the end of E1 overlaps with the beginning of E2. E2 ends after E1 has ended.

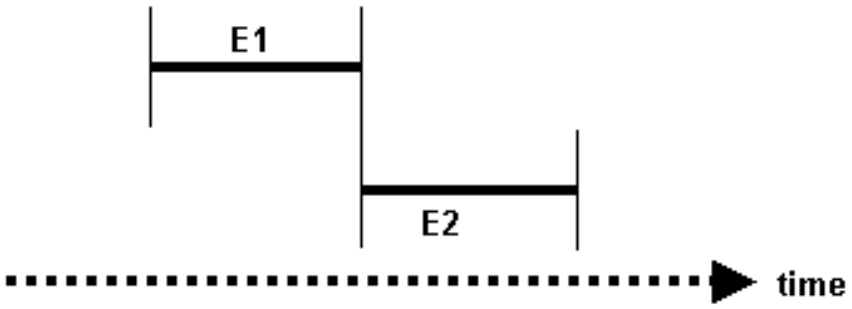
- E1 takes place entirely during the period that E2 exists, i.e. E1 starts after E2 and E1 finishes before E2.
- E1 occurs after E2 has ended.
- Both E1 and E2 may start at the same time (likewise, both E1 and E2 might finish at the same time).

Each of these relationships can be formally defined as an operation between intervallic events, and these operators are often referred to as 'Allen's operators'. Each of the above is perhaps more easily understood in graphical representations.

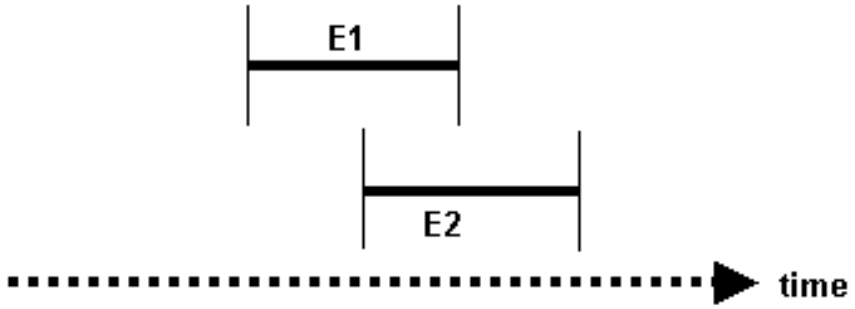
- E1 BEORE E2 — E1 occurs before E2



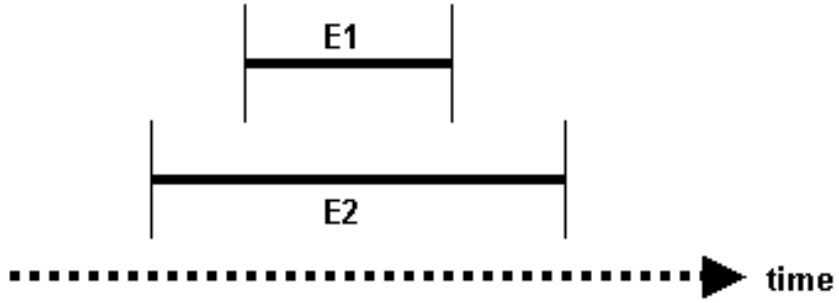
- E1 MEETS E2 — The end of E1 meets the beginning of E2



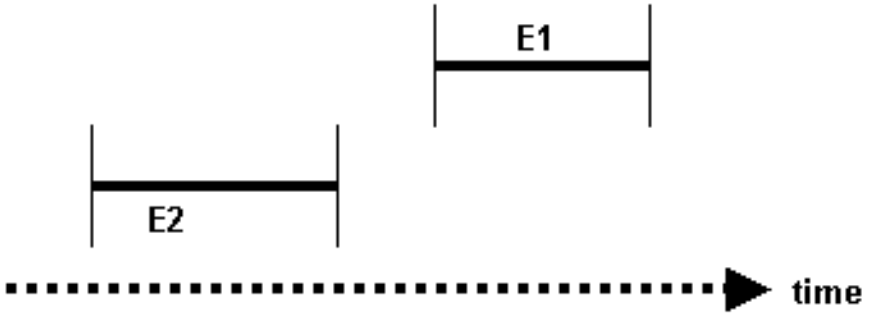
- E1 OVERLAPS E2 — E1 overlaps with E2



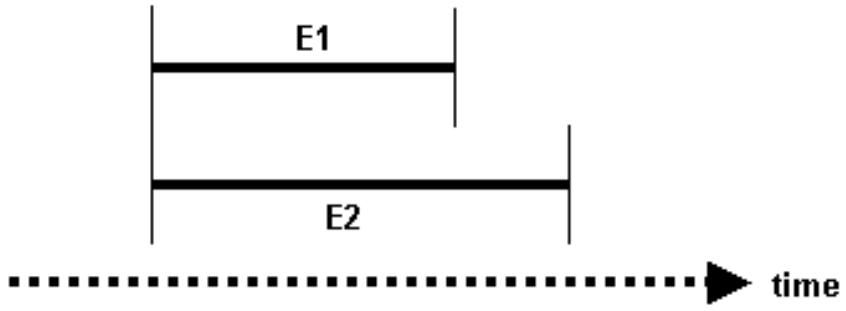
• E1 DURING E2 — E1 takes place during E2 (E1 and E1 may start and finish at the same time)



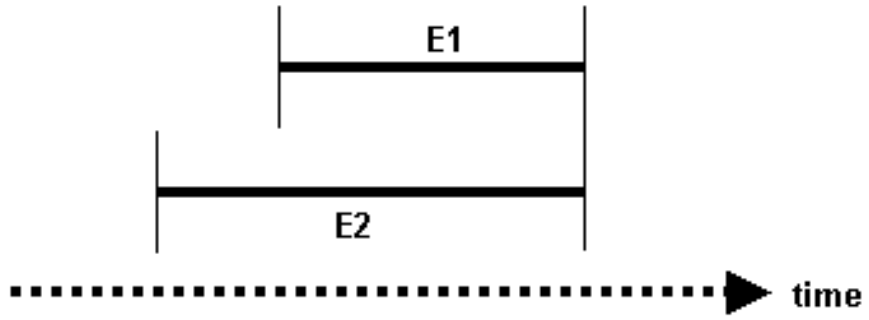
• E1 AFTER E2 — E1 occurs after E2 has ended



• E1 STARTS E2 — E1 starts at the same time that E2 starts (and E1 does not end after E2)



• E1 FINISHES E2 — E1 finishes at the same time that E2 ends (and E1 does not start before E2)



Unary intervals

For certain formal reasoning and temporal relational operators, the concept of a unary interval is important. A unary interval has a duration of one time unit (one time quantum). For example:

Interval_A(t4, t4)

The interval Interval_A has a start time of 't4' and an end time of 't4'. Therefore it starts at the beginning of the time quantum t4, and ends at the end of the time quantum t4, and so has a duration of 1 time quantum. Other examples of unary intervals include:

Interval_B(t1, t1) Interval_C(t9, t9) and so on.

Any interval with start time 's' and end time 'e' can be 'unfolded' into a sequence of unary intervals. For example, some Interval_D(t3, t7), which starts at t3 and ends at t7, can be thought of as being the same as the set of unary intervals:

(t3, t3) (t4, t4) (t5, t5) (t6, t6) (t7, t7)

We shall return to the use of the unary interval concept when relational temporal operators are investigated.

Relative and absolute times

A reference to a duration or interval can be absolute or relative to some time point or other interval. The position on the time axis of a period or of an instant can be given as an absolute position, such as the calendric time (e.g. "Blood pressure taken on 3November 1996"). This is a common approach adopted by data models underlying temporal medical databases.

However, it also is common in medicine to reason with relative time references: "Heartbeat measurement taken after a long walk", "the day after tomorrow", etc. The relationship between times can be qualitative (before, after, etc) as well as quantitative (three days before, 397 years after, etc.). Examples include:

- Mary's salary was raised yesterday.
- It happened sometime last week.
- It happened within three days of Easter.
- The French revolution occurred 397 years after the discovery of America.

Temporal data behaviour

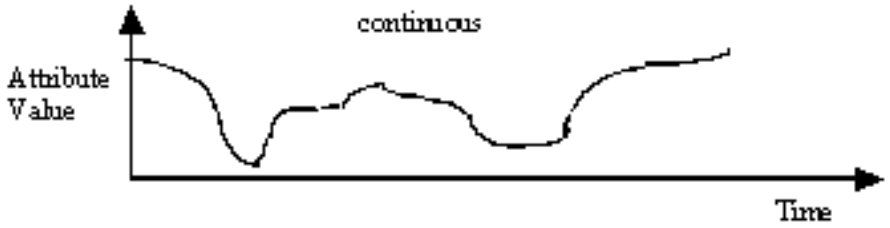
In general, the behaviour of temporal entities can be classified into one of four basic categories, namely:

- Discrete
- Continuous
- Stepwise constant
- Period based

These can be depicted graphically as shown in the figures below. We shall consider each category of temporal data individually.

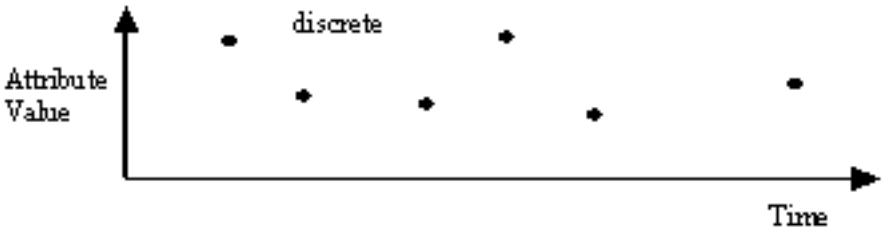
Continuous temporal data

Continuous behaviour is observed where an attribute value is recorded constantly over time such that its value may be constantly changing. Continuous behaviour can often be found in monitoring systems recording continuous characteristics - for example, a speedometer of a motorcar.



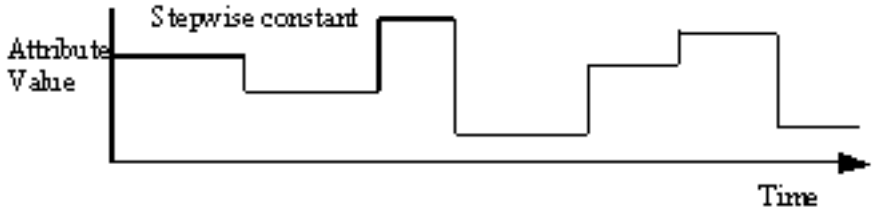
Discrete temporal data

Discrete data attributes are recorded at specific points in time but have no definition at any other points in time. Discrete data is associated with individual events such as "A complete check-up was on a particular date".



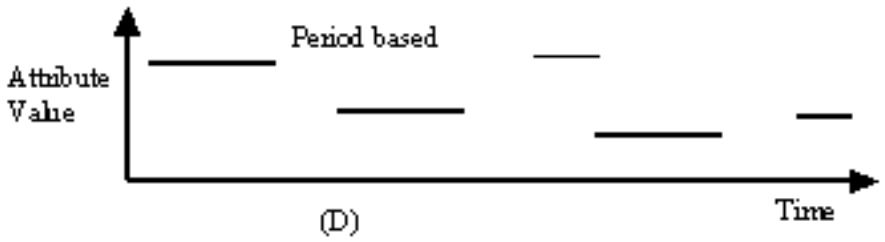
Stepwise constant temporal data

Stepwise constant data consists of values that change at a point in time, then remain constant until being changed again - for example, blood pressure measurement.



Period-based temporal data

Period-based data models the behaviour of events that occur over some period of time but, at the end of this period, become undefined. An example of period-based data would be patient drug usage records, where a patient takes a drug for a prescribed period of time and then stops taking it.



(D)

Exercises

Exercise 1 - Temporal terms and concepts

State whether each of the following is a point, duration or interval:

- 10 seconds
- 14.45 on 3/Feb/2007
- Three days
- From 1/3/99 to 5/1/99

Exercise 2 - Granularity

Which is the smaller level of temporal granularity: seconds or days?

Exercise 3 - Time quanta

What are time quanta? Are seconds or days examples of time quanta?

Exercise 4 - Interval operators

Consider the following scenario:

- Event E1 (s1, e1), where

1. the time E1 starts is t2, i.e. s1=t2
2. the time E1 ends is t7, i.e. e1=t7

- event E2 (s2, e2), where

1. the time E2 starts is t5=t4, i.e. s2=t7
2. the time E2 ends is t7, i.e. e2=t7

- event E2 starts at t4 and ends at t7

In terms of Allen's operators, what can we say about the relationship(s) between E1 and E2?

Draw a diagram showing the two events on the timeline t1, t2, … t8, to illustrate the relationship between the intervals.

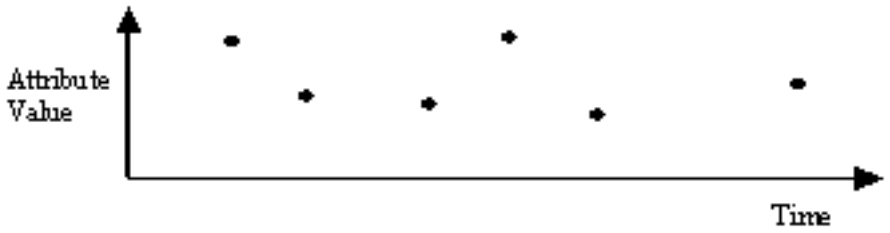
Exercise 5 - Relative and absolute time

Which of the following are absolute and which are relative times?

- 7th May 1991
- The day after tomorrow
- 14:13 on Monday 14th April 2010
- A week after last Tuesday
- 10 minutes after we finish work

Exercise 6 - Category of temporal data behaviour

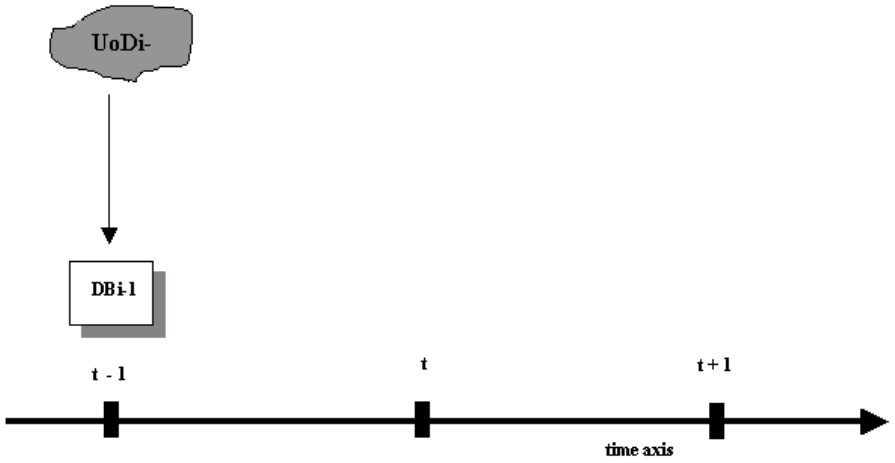
What kind of temporal data behaviour does the following graph represent?



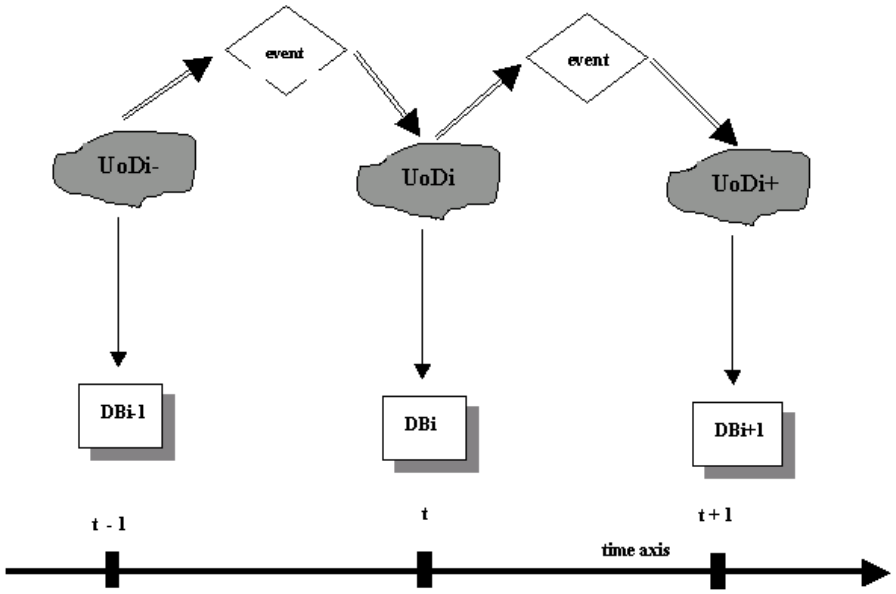
Temporal database concepts

A database might be informally defined as a collection of related data. This data corresponds to some piece of the Universe of Discourse (UoD) — i.e. the data we record represents a model of those parts of the real world (or an imagined world) which we are interested in, and wish to reason about. An example of a Universe of Discourse might be the patients, doctors, operating theatres, booked operations and available drugs in a hospital. Another UoD might be a basketball competition made up of each team, player, set of fixtures and results of games played to date.

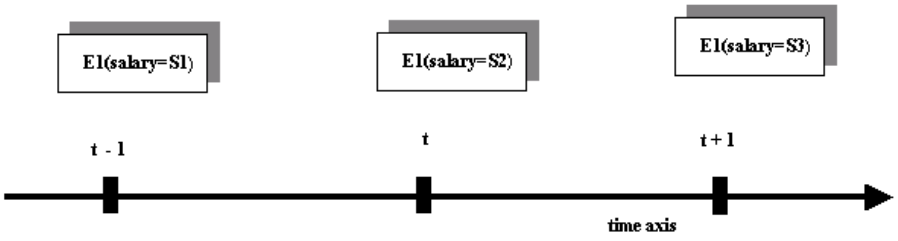
The figure below illustrates that at a particular point in time, the Universe of Discourse is in a particular state, and we create a database at this point in time recording details of the state of the UoD:



As time passes, the UoD is subject to events that change its state (i.e. events that change one or more of the component objects that populate our UoD). These changes are also reflected within the database, as shown in the next figure - assuming, of course, that we maintain the equivalence of the UoD with the database. So for example, a new patient arriving in the hospital means a change in our UoD. We will wish to update our database (with new patient details) to record and model the changes in our UoD. Therefore the state of the UoD and the state of our database changes with events that occur over time.



To give another example, assume that the UoD, and thus the database, contains information about employees and their salaries (see figure below). At time t-1 employee E1 has salary S1. At time t, the employee is given a salary increase and thus, his salary becomes S2. Later, at time t+1, the employee actually changes department and manager and his new salary becomes S3.



A temporal database is a database that deals with not only the 'current' state (in this example, t+1) but also with all the previous states of the salary history. To achieve that, we need to be able to model and reason about the time axis, and also be able to model and reason about the evolution of the data over time (usually referred to as the data or database history).

In order to deal with the time axis, the temporal database should have constructs to model the different notions of time (e.g. point, intervals, granularity

and calendar units) and reason about them through temporal operators such as DURING and BEFORE. In the last two decades, the Relational data model has become the most popular database model because of its simplicity and solid mathematical foundation. However, the Relational data model originally proposed does not address the temporal dimension of data. Since there is a need for temporal data, many temporal extensions to Relational data models and query languages have been proposed. The incorporation of the temporal dimension has taken a number of different forms. One approach is the strategy of extending the schema of the relation to include one or more distinguished temporal attributes (e.g. START TIME, END TIME) to represent the intervals of time a tuple was 'true' for the database. This approach is called tuple timestamping. Another approach, referred to as attribute timestamping, involves adding additional attributes to the schema, with the domain of each attribute extended from simple values to complex values to incorporate the temporal dimension.

Some important concepts

In this section we shall define some important concepts resulting from the previous discussions.

Valid time

The valid time of a fact is the time when the fact is true in the modelled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user.

An example would be that Fred Bloggs was employed as marketing director at Matrix Incorporated from 1/3/1999 to 5/6/1999 — i.e. the valid time interval for a tuple recording that Fred Bloggs was marketing director is from 1/3/1999 to 5/6/1999.

The valid time has nothing to do with the time that data has been added to the database, so for example, we may have recorded this data and valid time about Fred Bloggs on 28/2/1999.

The valid time for data can be changed — e.g. perhaps Fred Bloggs comes out of retirement, and so a second interval from 12/10/2000 to 1/6/2001 is added to the valid times for his employment as marketing director.

Transaction time

A database fact is stored in a database at some point in time. A transaction time of a data fact is the time at which the information about a modelled object or event is stored in the database. The transaction time is automatically recorded by the DBMS and cannot be changed. If a fact is updated to a database at

10:15 on 4/2/1999, this transaction time never changes. If the data is changed at a later time, then a second transaction time is generated for the change, and so on. In this way a history of database changes based on transaction time timestamps is built up.

Timestamp

A timestamp is a time value associated with some object, e.g. an attribute value or a tuple. The concept may be specialised to valid timestamp, transaction timestamp or, for a particular temporal data model, some other kind of timestamp.

Calendar

A calendar provides a human interpretation of time. As such, calendars ascribe meaning to temporal values where the particular meaning or interpretation is relevant to the user — in particular, the mapping between human-meaningful time values and underlying timeline. Calendars are most often cyclic, allowing human-meaningful time values to be expressed succinctly. For example, dates in the common Gregorian calendar may be expressed in the form <day, month, year> (for the UK) or <month, day, year> (for the US), where each of the fields 'day' and 'month' cycle as time passes (although year will continue to increase as time passes).

Time order

Different properties can be associated with a time axis composed from instants. Time is linear when the set of time points is completely ordered, also branching for the tasks of diagnosis, projection or forecasting (such as prediction of a medical evolution over time). Circular time describes recurrent events, such as "taking regular insulin every morning".

Review question 1

- What are the differences between valid time and transaction time?
- What are the differences between an absolute time and relative time? Give examples.
- What are the differences between a time period and time interval? Give examples.
- What are the differences between a time point and time period?

Database representation and reasoning with time

Four types of databases can be identified, according to the ability of a database to support valid time and/or transaction time, and the extent to which the database can be updated with regard to time.

Snapshot databases

A snapshot database can support either valid time or transaction time, but not both. It contains a 'snapshot' of the state of the database at a point in time. The term 'snapshot' is used to refer to the computational state of a database at a particular point in time.

If the data stored in the database represents a correct model of the world at this current point in time, then the database state represents transaction time — i.e. the database contents are the representation of the world for the current timestamp of the DBMS.

If the data stored in the database is valid or true at this current point in time, then the database state represents valid time.

Results from database operations take effect from commit time and past states are forgotten (overwritten). Snapshot databases are what one would usually think of as a database (with no recording of changes or past data values).

Rollback databases

A rollback database has a sequence of states that are indexed by transaction time, i.e. the time the information was stored in the database. A rollback database preserves the past states of the database but not those of the real world. Therefore, it cannot correct errors in past information, nor represent that some facts will be valid for some future time.

Each time a change is made to a database, the before and after states are recorded for the transaction timestamp at the time the change takes place — so that at a later date, the database can be returned to a previous state.

Historical databases

These support only valid time. As defined earlier,valid time of a fact is the time when the fact is true in the modelled reality. The semantics of valid time are closely related to reality and, therefore, historical databases can maintain the history of the modelled Universe of Discourse and represent current knowledge about the past. However, historical databases cannot view the database as it was at some moment in the past.

In other words, a historical database can record that some fact F was valid from 1/1/1996 to 4/5/1998, but it is not recorded when these valid times were added to the database, so it is not possible to state that on 1/1/1997 it was recorded that fact F was true — perhaps we have retrospectively recorded when fact F was true at some time after 1/1/1997. Thus it is possible to reason about what we think was true for a given point of time, but not possible to answer questions about when we knew the facts were true, since no changes to the database have been recorded.

A historical database could be thought of as a snapshot of our beliefs about the past — at this point in time, we believe fact F was valid from 1/1/1996 to 4/5/1998 and that is all we can say.

Temporal databases

These represent both transaction time and valid time and thus are both historical and rollback. So temporal databases record data about valid time (e.g. that we believe fact F is valid from 1/1/1996 to 4/5/1998) and the transaction time when such data was entered into the database (e.g. that we added this belief on 4/6/1997). This means that we can now rollback our temporal database to find out what our valid time beliefs were for any given past transaction time (e.g. what did we believe about fact F on 2/3/1997?).

Temporal databases allows retroactive update — i.e. coming into effect after the time to which the data was referenced. Temporal databases also support proactive update — i.e. coming into effect before the time to which the data was referenced. Thus, in a temporal database, we can represent explicitly a scenario such as the following:

- On January 17, 1997 (transaction time), the physician entered in the patient's record the fact that on January 12, 1997 (valid time) the patient had an allergic reaction to a sulfa-type drug, and that the physician believed that the patient would take a non-sulfa type drug from January 20 to February 20. If on February 1st the physician decides the patient no longer needs the drug, the database will be amended to show that the patient only took the non-sulfa drug from January 20th to February 1st.

Review question 2

- What are the differences between relative and absolute times?
- What are the differences between a snapshot database and a historical database?
- What are the differences between a snapshot database and a temporal database?

Incorporating time in Relational databases

In this final section, we briefly look at ways in which time data may be recorded and queried using extensions to the Relational model. It is not necessary to understand the fine details of temporal Relational query statements, but you should understand the different kinds of timestamping approaches, and the concepts around the two suggested temporal Relational operators, COALESCE and UNFOLD.

Recording changes to databases

Let us now examine how the different types of temporal databases may be represented in the Relational model. A conventional relation can be visualised as a two-dimensional structure of tuples and attributes. Adding time as a third dimension to a conventional relation will change the relation into a three-dimensional structure. There have been many attempts in the past to find a suitable design structure that would be able to cope with handling the extra time dimension. We will describe these attempts in general, pointing out their successes and failures.

Archiving

One of the earliest methods of maintaining time-based data for a database was to backup (archive) all the data stored in the database at regular intervals; i.e. the entire database was copied, with a timestamp, weekly or daily. However, information between backups is lost and retrieval of achieved information is slow and clumsy, since an entire version of the database needs to be loaded/searched.

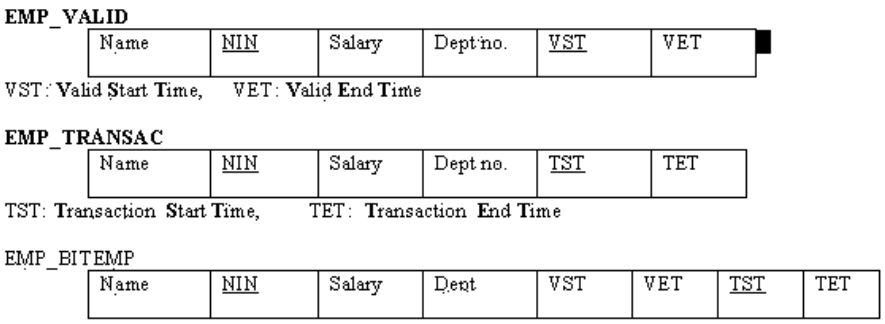
Time-slicing

The time-slicing method works if the database is stored as tables, such as in the Relational database model. When a change to the database occurs, at least one attribute of at least one tuple (record) from a particular table is changed. The time-slicing approach simply stores the entire table prior to the event and gives it a timestamp. Then a duplicate but updated copy is created and becomes part of the 'live' database state. Time-slicing is more efficient and easier to implement than archiving, since only those tables which are changed are copied with a timestamp. However, there still a lot of data redundancy in the time-slicing approach. This data redundancy is a result of duplicating a whole table when, for example, only one attribute value of one tuple was changed. With time-slicing, it is not possible to know the lifespan of a particular database state.

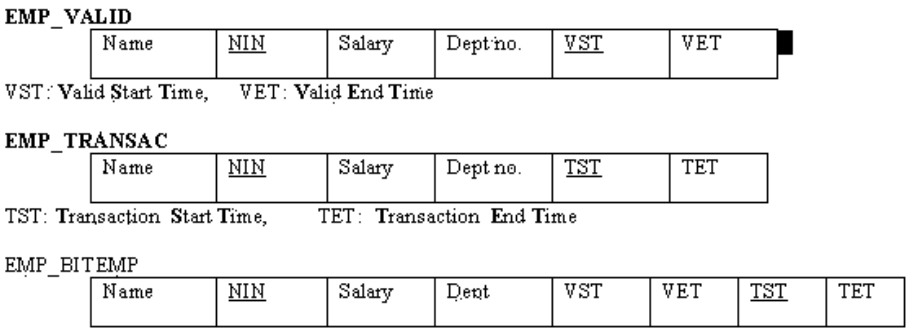
Tuple timestamping

Tuple timestamping means that each relation is augmented with two time attributes representing a time interval, as illustrated in the figure below (the two attributes are the time points a tuple was 'live' for the database, i.e. starting time point and ending time point).

The entire table does not need to be duplicated; new tuples are simply added to the existing table when an event occurs. These new tuples are appended at the end of the table.



Examples:



Example with data:

Name	NIN	Salary	Dept.	VST	VET
Patel	SW2313	26000	Accounting	1998-06-01	now
Smith	MS2344	13000	Marketing	1994-08-21	1996-01-31
Smith	WE6741	26000	Accounting	1996-02-01	1997-03-31
Smith	JK2987	37000	Accounting	1997-04-01	now
Jackson	KH2322	31000	Marketing	1996-05-01	1997-08-10
Khan	PW2121	39000	Accounting	1998-08-01	now

Attribute timestamping

Attribute timestamping is implemented by attribute values consisting of two components, a timestamp and the data value. Some approaches to attribute timestamping use time intervals instead of timestamps, which express lifespan better than other constructs. Using time intervals can avoid the main problem of tuple timestamping, which breaks tuples into unmatched versions within or across tables. Retrieval is fast for single attributes, but poor for complete tuples.

UNFOLD and COALESCE: Two useful temporal Relational operators

The temporal Relational operators UNFOLD and COALESCE are important core concepts in most suggested extensions to the Relational model for temporal databases. The UNFOLD operator makes use of the concept of unary intervals introduced earlier in the chapter, and the COALECSE operator is the logical opposite of UNFOLD. We shall investigate each below.

UNFOLD

The temporal Relational operator UNFOLD works on a set of tuples with valid or transaction time intervals (i.e. start and end time attributes) and expands the relation so that the data has a tuple for each unary interval appearing in any of the intervals for the data. This is probably best understood with an example. Consider the following temporal relation, NIGHTSHIFT, recording employees who are security guards for particular factory sites for particular dates (we will use valid dates in this example):

EmployeeNumber	SiteNumber	StartVDate	FinishVDate
#0027	S03	2/3/99	5/3/99
#0027	S03	8/3/99	9/3/99
#0102	S02	1/3/99	2/3/99
#0102	S02	4/3/99	7/3/99

We can see intuitively from the table that employee #0027 has been on duty at site S03 for 2, 3, 4, 5 and 8, 9 March 1999. The UNFOLD operator makes this formal and explicit by replacing the tuples with intervals greater than one time quantum with multiple, unary intervals as follows:

EmployeeNumber	SiteNumber	StartVDate	FinishVDate
#0027	S03	2/3/99	2/3/99
#0027	S03	3/3/99	3/3/99
#0027	S03	4/3/99	4/3/99
#0027	S03	5/3/99	5/3/99
#0102	S02	1/3/99	1/3/99
#0102	S02	2/3/99	2/3/99
#0102	S02	4/3/99	4/3/99
#0102	S02	5/3/99	5/3/99

EmployeeNumber	SiteNumber	StartVDate	FinishVDate
#0102	S02	6/3/99	6/3/99
#0102	S02	7/3/99	7/3/99

The usefulness of such an operator means that once a table has been UNFOLDED it becomes a simple query to find out whether a tuple was valid for any given time quantum.

COALESCE

The COALESCE temporal Relational operator is the logical opposite of UNFOLD in that it attempts to reduce the number of tuples to the minimum, i.e. wherever a sequence of intervals can be summarised in a single interval, this is done. Consider the following set of tuples for our NIGHTSHIFT relation:

EmployeeNumber	SiteNumber	StartVDate	FinishVDate
#0027	S03	2/3/99	3/3/99
#0027	S03	4/3/99	4/3/99
#0027	S03	8/3/99	9/3/99
#0027	S03	10/3/99	11/3/99
#0102	S02	1/3/99	2/3/99
#0102	S02	4/3/99	4/3/99
#0102	S02	5/3/99	8/3/99

We might assume that employee #0027 came in for extras days on 4/3/99 to cover a colleague's sick leave; likewise for employee #0102 on 4/3/99.

We can see intuitively that employee #0027 was working from 2/3/99 until 4/3/99, and from 8/3/99 until 11/3/99. The COALESCE temporal Relational operator formalises this concept of using the minimum number of tuples to represent the intervals when data is valid. The result of the COALESCE query on the relation is as follows:

Result of query: NIGHTSHIFT COALESCE StartVDate, FinishVDate

EmployeeNumber	SiteNumber	StartVDate	FinishVDate
#0027	S03	2/3/99	4/3/99
#0027	S03	8/3/99	11/3/99
#0102	S02	1/3/99	2/3/99
#0102	S02	4/3/99	8/3/99

You may wish to refer to the recommended reading for this chapter to investigate further the details of these two temporal Relational operators.

Further work and application

Review question

Consider the following statements spoken in 2015.

"Two days ago, I was seven years old," said a little girl. "Next year, I'll be 10!"

Can this be true? If so, what was the date the little girl was born, and what was the date when she was speaking?

Discussion topic

Discuss how time instants and time periods can be use to model time-oriented medical data.

Additional content and activities

Temporal database design

Amongst many existing data models, we have chosen the Entity Relationship Time (ERT) model for its simplicity and the support of the SQL language.

Entity Relationship Time model

The Entity Relationship Time model (ERT) is an extended entity-relationship type model that additionally accommodates 'complex objects' and 'time'. The ERT model consists of the following concepts:

- **Entity:** Anything, concrete or abstract, uniquely identifiable and being of interest during a certain time period.
- **Entity class:** The collection of all the entities to which a specific definition and common properties apply at a specific time period.
- **Relationship:** Any permanent or temporary association between two entities or between an entity and a value.
- **Relationship class:** The collection of all the relationships to which a specific definition applies at a specific time period.
- **Value:** An object perceived individually, which is only of interest when it is associated with an entity. That is, values cannot exist on their own.
- **Value class:** The proposition establishing a domain of values.
- **Time period:** A pair of time points expressed at the same abstraction level.
- **Time period class:** A collection of time periods.
- **Complex object:** A complex value or a complex entity. A complex entity is an abstraction (aggregation or grouping) of entities and relationships. A complex value is an abstraction (aggregation or grouping) of simple values.
- **Complex object class:** A collection of complex objects. That is, it can be a complex entity or a complex value class.

The time structure denotes the granularity of the timestamp. Various types of granularity are provided including: second, minute, hour, day, month and year. The default granularity type is the second. Obviously, if the user specifies a granularity type for a timestamped ERT object, this granularity type supports all the super types of it. For example, if the day granularity type is specified, the actual timestamp values will have a year, a month and a day reference. Furthermore, the time structure can be user-defined, e.g. a granularity of a week may be necessary.

Besides the temporal dimension, ERT differs from the entity-relationship model in that it regards any association between objects as the unified form of a relationship, thus avoiding the unnecessary distinction between attributes and relationships. A relationship class denotes a set of associations between two entity classes or between an entity class and a value class, and in that, all relationships are binary and each relationship has its inverse. Additionally, for every relationship class, apart from the objects participating, the relationship involvements of the objects and the cardinality constraints should be specified.

The relationship involvements specify the roles of the object in the specified relationship e.g. Employee works_for Department, Department employs Employee. Furthermore, for each relationship involvement, a user-supplied constraint rule must be defined which restricts the number of times an entity or a value can participate in this involvement. This constraint is called a cardinality constraint and it is applied to the instances of the relationship by restricting its population. As an example, consider the case where the constraint for relationship (Employee, works_for, Department) is 1:N. This is interpreted as: one Employee can be associated to at least one instance of Department, while there is no upper limit to the number of Departments an Employee is associated with.

Another additional feature of the ERT model is the concept of complex object (entity or value class). The basic motivation for the inclusion of the complex object class in the external formalism, is to abstract away detail, which in several cases is not of interest. In addition, no distinction is made between aggregation and grouping, but rather a general composition mechanism is considered which involves relationships/attributes. For the modelling of complex objects, a special type of relationship class is provided, named IsPartOf.

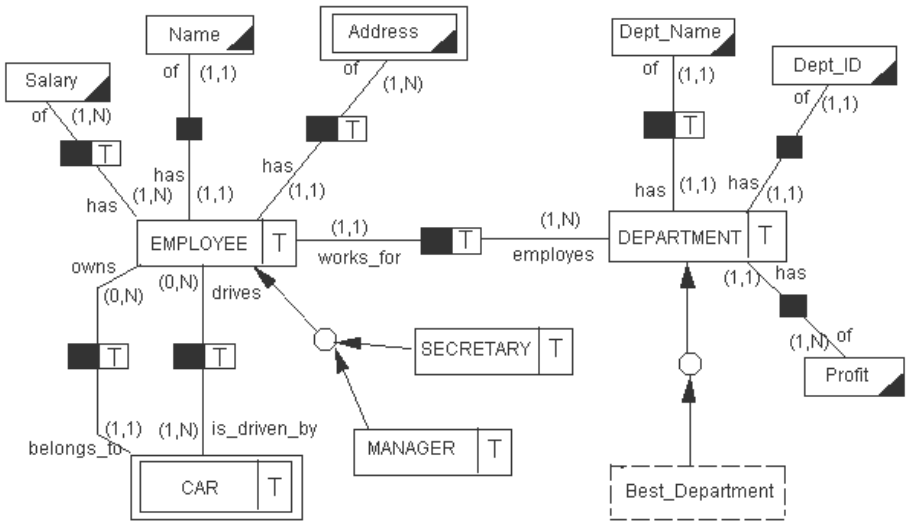
An entity or relationship class can be derived. This implies that its instances are not stored by default, but for each such derivable component, there is a corresponding derivation rule which gives the instances of this class at any time. Derived schema components are one of the fundamental mechanisms in semantic models for data abstraction and encapsulation.

The graphical representation of ERT constructs is depicted below.

The construct of timestamp is modelled as a pair (a,b), where a denotes the temporal semantic category (TSC) while b denotes the time structure. Three different temporal semantic categories have been identified, namely:

- Decomposable
- Nondecomposable
- time points

The ERT notation for the three concepts is TPI for the decomposable time periods, TI for the nondecomposable time interval and TP for the time point. The time structure denotes the granularity of the timestamp.



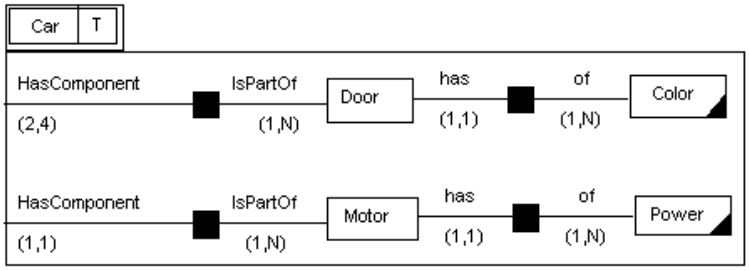
An example ERT schema

An example ERT schema is depicted in the figure above. Several entity classes can be found in the diagram, namely: Employee, Manager, Secretary, Department and Car. These are represented using a rectangle with the addition of a 'time box', which shows that the entity is time varying. Complex entity classes are represented using double rectangles, e.g. Car, while for derived entity class the rectangle is dashed, e.g. Best_Department. For the entity classes participating in a hierarchy, an arrow is used for specifying the parent entity class.

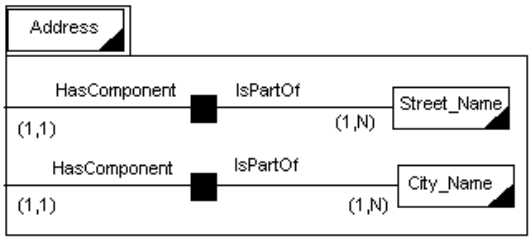
Relationship classes are represented using a small filled rectangle and can be time varying (with the addition of 'T') or not. In addition, for every relationship class, relationship involvements and cardinality constraints are specified.

Value classes e.g. Name, Address or Dept_ID, are represented with rectangles which have a small black triangle at the bottom right corner. Complex value classes e.g. Address are represented using double rectangles.

Every complex object can be further analysed into its components. The notation used is the same as in the top-level schema, adding the special type relationship IsPartOf. The complex entity class Car is illustrated in the first figure below, while the complex value class Address is illustrated in the second figure below.



Complex entity class CAR

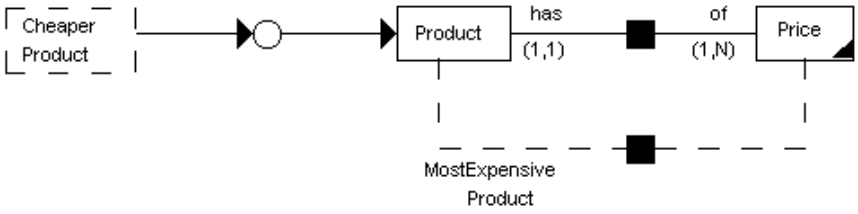


Complex value class Address

An example of a derived entity class is illustrated in the figure below, and population of entity class Employee and entity class Department are shown in the two tables below that.

E$	Timestamp	Name	Salary	Department (works_for)
1	[1/1/85,1/1/90)	{Ali, John}	20000	2: [1/1/85,1/1/90)
2	[1/1/89,1/1/91)	David	23000	1: [1/1/89,1/1/91)
3	[1/1/70,1/1/98)	{Ali, Wong}	18000	2: [1/1/70,1/1/98)
4	[1/9/93,1/1/95]	Rakesh	15000	3: [1/9/93,1/1/95]

Population of entity class Employee



An ERT schema with derived objects

D$	Dept_Name	Dept_ID	Employee (employs)	Profit
1	Toys: [1/1/60,1/1/99)	D1A	2: [1/1/89,1/1/91)	100000
2	Menswear: [1/1/70,1/1/99)	D2A	1: [1/1/85,1/1/90), 3: [1/1/70,1/1/92)	230000
3	{(Sports: [1/1/88,1/1/91), (SportsDept: [1/1/91,1/1/99)}	D2B	4: [1/9/93,1/7/95)	450000

Population of entity class Department

The ERT-SQL language

In this section, we describe the temporal query language ERT-SQL primarily used for manipulating an ERT-based database. The ERT-SQL is based on the standard SQL2 and on the valid time SQL (VT-SQL) language.

The ERT-SQL statements are divided into three language groups:

- **Data Definition Language (DDL):** These statements are used to define the ERT schema, by specifying one by one all ERT components (for example, CREATE ENTITY, CREATE RELATIONSHIP).
- **Data Manipulation Language (DML):** DML statements are used to query or update data of an ERT database (for example, SELECT, INSERT).
- **Schema Manipulation Language (SML):** These statements are used to alter the ERT schema (for example DROP ENTITY, ALTER ENTITY).

Full description of ERT-SQL is beyond the scope of this chapter. Thus, in the

rest of this section, the capabilities of ERT-SQL are illustrated with a number of examples.

The CREATE statement has the structure and facilities similar to that of the standard SQL, but it is extended in order to be able to capture the temporal and structural semantics of the ERT model.

"Create entity employee class, employee and also relationship class (employee, department)."

CREATE ENTITY Employee (TPI,DAY)

(VALUE,Name,CHAR(20),has,1,1,of,1,1)

(VALUE,Salary,INTEGER,has,1,N,of,1,N(TPI,DAY))

(COMPLEX VALUE,Address,has,1,1,of,1,N(TPI,DAY))

CREATE RELATIONSHIP (Employee,Department,works_for,1,1,employs,1,N(TI,MONTH))

The SELECT statement also has the structure and facilities similar to that of the standard, with temporal capability.

"Give the periods during which the employee 'Ali' had been working for the 'Toys' department."

SELECT [Employee, Department, works_for].TIMESTAMP

FROM Employee, Department

WHERE Dept_Name = 'Toys' AND Name='Ali'

The INSERT statement is used to add instances both to entity and to relationship classes.

"Insert the 'Toys' department with existence period from the first day of 1994 and Profit £10000."

INSERT INTO Department

VALUES

Dept_ID = 'D151294'

Dept_Name = 'Toys' '[1/1/1994,)'

Profit = 10000

TIMESTAMP = '[1/1/1994,)'

"Insert the information that the employee with name 'Ali' has been working for the 'Toys' department from 5/4/1994 to 1/5/1996."

INSERT INTO RELATIONSHIP (Employee, Department, works_for)

TIMESTAMP = '[5/4/1994, 1/5/1996)'

WHERE (Name = ('Ali')) AND (Dept_Name = 'Toys')

The DELETE statement is used to delete particular instances from an entity class or from a relationship class.

"Delete the information that the employee 'Ali' had worked for the 'Toys' department for the period [1/1/1990,1/2/1990)."

DELETE FROM RELATIONSHIP (Employee,Department,works_for)

WHERE (TIMESTAMP = '[1/1/1990, 1/2/1990)') AND (Name = 'Ali') AND (Dept_Name = 'Toys')

The UPDATE statement is used to alter the contents of an instance either of an entity class or of a relationship class.

"The department 'Toys' had this name for the period '[1/1/1970,1/1/1987)' and not for the '[1/1/1960,1/1/1990)'. Enter the correct period."

UPDATE RELATIONSHIP(Department, Dept_Name, has)

SET TIMESTAMP = '[1/1/1970,1/1/1987)'

WHERE(TIMESTAMP='[1/1/1960,1/1/1990)') AND (Dept_Name='Toys')

The DROP statement is used to remove entity classes (simple, complex or derived) or relationship classes from an ERT schema.

"Remove the Department entity class."

DROP ENTITY Department

"Remove the relationship class between entities Employee and Car with role name 'owns'."

DROP RELATIONSHIP (Employee, Car, owns)

The ALTER statement is used to add and remove value classes or to add a new component to a complex object.

"Add to the entity Employee, value class 'Bank_Account'."

ALTER ENTITY Employee

ADD (VALUE,Bank_Account,INTEGER,has,1,1,of,1,N)

"Alter timestamped relationship between Employee and Department. Set temporal semantic category to TPI and granularity to DAY."

ALTER RELATIONSHIP

(Employee, Department, works_for)

TIMESTAMP (TPI,DAY)

Additional review questions

1. Using the employee database example in the Extend section, express the following query in ERT-SQL language:

- "Insert the information that the employee with name 'Ali' has been working for the 'Toys' department from 5/4/1994 to 1/5/1996."
- "Alter timestamped relationship between Employee and Department. Set temporal semantic category to TPI and granularity to DAY."

2. What are the three language groups of ERT-SQL?

Chapter 19. Data Warehousing and Data Mining

Table of contents

    - Staging area
    - Checkpoint restart logic
    - Data loading
- Data warehousing and data mining
- General introduction to data mining
    - Data mining concepts
    - Benefits of data mining
- Comparing data mining with other techniques
    - Query tools vs. data mining tools
    - OLAP tools vs. data mining tools
    - Website analysis tools vs. data mining tools
    - Data mining tasks
    - Techniques for data mining
    - Data mining directions and trends
- Data mining process
    - The process overview
    - The process in detail
        * Business objectives determination
        * Data preparation
            · Data selection
            · Data pre-processing
            · Data transformation
        * Data mining
        * Analysis of results
        * Assimilation of knowledge
- Data mining algorithms
    - From application to algorithm
    - Popular data mining techniques
        * Decision trees
        * Neural networks
        * Supervised learning
            · Preparing data
        * Unsupervised learning - self-organising map (SOM)
- Discussion topics

Objectives

At the end of this chapter you should be able to:

- Distinguish a data warehouse from an operational database system, and appreciate the need for developing a data warehouse for large corporations.
- Describe the problems and processes involved in the development of a data warehouse.
- Explain the process of data mining and its importance.

- Understand different data mining techniques.

Context

Rapid developments in information technology have resulted in the construction of many business application systems in numerous areas. Within these systems, databases often play an essential role. Data has become a critical resource in many organisations, and therefore, efficient access to the data, sharing the data, extracting information from the data, and making use of the information stored, has become an urgent need. As a result, there have been many efforts on firstly integrating the various data sources (e.g. databases) scattered across different sites to build a corporate data warehouse, and then extracting information from the warehouse in the form of patterns and trends.

A data warehouse is very much like a database system, but there are distinctions between these two types of systems. A data warehouse brings together the essential data from the underlying heterogeneous databases, so that a user only needs to make queries to the warehouse instead of accessing individual databases. The co-operation of several processing modules to process a complex query is hidden from the user.

Essentially, a data warehouse is built to provide decision support functions for an enterprise or an organisation. For example, while the individual data sources may have the raw data, the data warehouse will have correlated data, summary reports, and aggregate functions applied to the raw data. Thus, the warehouse is able to provide useful information that cannot be obtained from any individual databases. The differences between the data warehousing system and operational databases are discussed later in the chapter.

We will also see what a data warehouse looks like – its architecture and other design issues will be studied. Important issues include the role of metadata as well as various access tools. Data warehouse development issues are discussed with an emphasis on data transformation and data cleansing. Star schema, a popular data modelling approach, is introduced. A brief analysis of the relationships between database, data warehouse and data mining leads us to the second part of this chapter - data mining.

Data mining is a process of extracting information and patterns, which are previously unknown, from large quantities of data using various techniques ranging from machine learning to statistical methods. Data could have been stored in files, Relational or OO databases, or data warehouses. In this chapter, we will introduce basic data mining concepts and describe the data mining process with an emphasis on data preparation. We will also study a number of data mining techniques, including decision trees and neural networks.

We will also study the basic concepts, principles and theories of data warehousing and data mining techniques, followed by detailed discussions. Both

theoretical and practical issues are covered. As this is a relatively new and popular topic in databases, you will be expected to do some extensive searching, reading and discussion during the process of studying this chapter.

General introduction to data warehousing

In parallel with this chapter, you should read Chapter 31, Chapter 32 and Chapter 34 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

What is a data warehouse?

A data warehouse is an environment, not a product. The motivation for building a data warehouse is that corporate data is often scattered across different databases and possibly in different formats. In order to obtain a complete piece of information, it is necessary to access these heterogeneous databases, obtain bits and pieces of partial information from each of them, and then put together the bits and pieces to produce an overall picture. Obviously, this approach (without a data warehouse) is cumbersome, inefficient, ineffective, error-prone, and usually involves huge efforts of system analysts. All these difficulties deter the effective use of complex corporate data, which usually represents a valuable resource of an organisation.

In order to overcome these problems, it is considered necessary to have an environment that can bring together the essential data from the underlying heterogeneous databases. In addition, the environment should also provide facilities for users to carry out queries on all the data without worrying where it actually resides. Such an environment is called a data warehouse. All queries are issued to the data warehouse as if it is a single database, and the warehouse management system will handle the evaluation of the queries.

Different techniques are used in data warehouses, all aimed at effective integration of operational databases into an environment that enables strategic use of data. These techniques include Relational and multidimensional database management systems, client-server architecture, metadata modelling and repositories, graphical user interfaces, and much more.

A data warehouse system has the following characteristics:

- It provides a centralised utility of corporate data or information assets.
- It is contained in a well-managed environment.
- It has consistent and repeatable processes defined for loading operational data.
- It is built on an open and scalable architecture that will handle future expansion of data.

- It provides tools that allow its users to effectively process the data into information without a high degree of technical support.

A data warehouse is conceptually similar to a traditional centralised warehouse of products within the manufacturing industry. For example, a manufacturing company may have a number of plants and a centralised warehouse. Different plants use different raw materials and manufacturing processes to manufacture goods. The finished products from the plants will then be transferred to and stored in the warehouse. Any queries and deliveries will only be made to and from the warehouse rather than the individual plants.

Using the above analogy, we can say that a data warehouse is a centralised place to store data (i.e. the finished products) generated from different operational systems (i.e. plants). For a big corporation, for example, there are normally a number of different departments/divisions, each of which may have its own operational system (e.g. database). These operational systems generate data day in and day out, and the output from these individual systems can be transferred to the data warehouse for further use. Such a transfer, however, is not just a simple process of moving data from one place to another. It is a process involving data transformation and possibly other operations as well. The purpose is to ensure that heterogeneous data will conform to the same specification and requirement of the data warehouse.

Building data warehouses has become a rapidly expanding requirement for most information technology departments. The reason for growth in this area stems from many places:

- With regard to data, most companies now have access to more than 20 years of data on managing the operational aspects of their business.
- With regard to user tools, the technology of user computing has reached a point where corporations can now effectively allow the users to navigate corporation databases without causing a heavy burden to technical support.
- With regard to corporate management, executives are realising that the only way to sustain and gain an advantage in today's economy is to better leverage information.

Operational systems vs. data warehousing systems

Before we proceed to detailed discussions of data warehousing systems, it is beneficial to note some of the major differences between operational and data warehousing systems.

Operational systems

Operational systems are those that assist a company or an organisation in its day-to-day business to respond to events or transactions. As a result, operational system applications and their data are highly structured around the events they manage. These systems provide an immediate focus on business functions and typically run in an online transaction processing (OLTP) computing environment. The databases associated with these applications are required to support a large number of transactions on a daily basis. Typically, operational databases are required to work as fast as possible. Strategies for increasing performance include keeping these operational data stores small, focusing the database on a specific business area or application, and eliminating database overhead in areas such as indexes.

Data warehousing systems

Operational system applications and their data are highly structured around the events they manage. Data warehouse systems are organised around the trends or patterns in those events. Operational systems manage events and transactions in a similar fashion to manual systems utilised by clerks within a business. These systems are developed to deal with individual transactions according to the established business rules. Data warehouse systems focus on business needs and requirements that are established by managers, who need to reflect on events and develop ideas for changing the business rules to make these events more effective.

Operational systems and data warehouses provide separate data stores. A data warehouse's data store is designed to support queries and applications for decision-making. The separation of a data warehouse and operational systems serves multiple purposes:

- It minimises the impact of reporting and complex query processing on operational systems.
- It preserves operational data for reuse after that data has been purged from the operational systems.
- It manages the data based on time, allowing the user to look back and see how the company looked in the past versus the present.
- It provides a data store that can be modified to conform to the way the users view the data.
- It unifies the data within a common business definition, offering one version of reality.

A data warehouse assists a company in analysing its business over time. Users of data warehouse systems can analyse data to spot trends, determine problems and compare business techniques in a historical context. The processing that these systems support include complex queries, ad hoc reporting and static reporting (such as the standard monthly reports that are distributed to managers).

The data that is queried tends to be of historical significance and provides its users with a time-based context of business processes.

Differences between operational and data warehousing systems

While a company can better manage its primary business with operational systems through techniques that focus on cost reduction, data warehouse systems allow a company to identify opportunities for increasing revenues, and therefore, for growing the business. From a business point of view, this is the primary way to differentiate these two mission-critical systems. However, there are many other key differences between these two types of systems.

- **Size and content:** The goals and objectives of a data warehouse differ greatly from an operational environment. While the goal of an operational database is to stay small, a data warehouse is expected to grow large – to contain a good history of the business. The information required to assist us in better understanding our business can grow quite voluminous over time, and we do not want to lose this data.
- **Performance:** In an operational environment, speed is of the essence. However, in a data warehouse, some requests – 'meaning-of-life' queries – can take hours to fulfil. This may be acceptable in a data warehouse environment, because the true goal is to provide better information, or business intelligence. For these types of queries, users are typically given a personalised extract of the requested data so they can further analyse and query the information package provided by the data warehouse.
- **Content focus:** Operational systems tend to focus on small work areas, not the entire enterprise; a data warehouse, on the other hand, focuses on cross-functional subject areas. For example, a data warehouse could help a business understand who its top 20 at-risk customers are – those who are about to drop their services – and what type of promotions will assist in not losing these customers. To fulfil this query request, the data warehouse needs data from the customer service application, the sales application, the order management application, the credit application and the quality system.
- **Tools:** Operational systems are typically structured, offering only a few ways to enter or access the data that they manage, and lack a large amount of tools accessibility for users. A data warehouse is the land of user tools. Various tools are available to support the types of data requests discussed earlier. These tools provide many features that transform and present the data from a data warehouse as business intelligence. These features offer a high flexibility over the standard reporting tools that are offered within an operational systems environment.

Benefits of data warehousing systems

Driven by the need to gain competitive advantage in the marketplace, organisations are now seeking to convert their operational data into useful business intelligence – in essence fulfilling user information requirements. The user's questioning process is not as simple as one question and the resultant answer. Typically, the answer to one question leads to one or more additional questions. The data warehousing systems of today require support for dynamic iterative analysis – delivering answers in a rapid fashion. Data warehouse systems, often characterised by query processing, can assist in the following areas:

- **Consistent and quality data:** For example, a hospital system had a severe data quality problem within its operational system that captured information about people serviced. The hospital needed to log all people who came through its door regardless of the data that was provided. This meant that someone who checked in with a gunshot wound and told the staff his name was Bob Jones, and who subsequently lost consciousness, would be logged into the system identified as Bob Jones. This posed a huge data quality problem, because Bob Jones could have been Robert Jones, Bobby Jones or James Robert Jones. There was no way of distinguishing who this person was. You may be saying to yourself, big deal! But if you look at what a hospital must do to assist a patient with the best care, this is a problem. What if Bob Jones were allergic to some medication required to treat the gunshot wound? From a business sense, who was going to pay for Bob Jones' bills? From a moral sense, who should be contacted regarding Bob Jones' ultimate outcome? All of these directives had driven this institution to a proper conclusion: They needed a data warehouse. This information base, which they called a clinical repository, would contain quality data on the people involved with the institution – that is, a master people database. This data source could then assist the staff in analysing data as well as improving the data capture, or operational system, in improving the quality of data entry. Now when Bob Jones checks in, they are prompted with all of the patients called Bob Jones who have been treated. The person entering the data is presented with a list of valid Bob Joneses and several questions that allow the staff to better match the person to someone who was previously treated by the hospital.
- **Cost reduction:** Monthly reports produced by an operational system could be expensive to store and distribute. In addition, very little content in the reports is typically universally useful, and because the data takes so long to produce and distribute, it's out of sync with the users' requirements. A data warehouse implementation can solve this problem. We can index the paper reports online and allow users to select the pages of importance to be loaded electronically to the users' personal workstations. We could save a bundle of money just by eliminating the distribution of massive paper reports.

- **More timely data access:** As noted earlier, reporting systems have become so unwieldy that the data they present is typically unusable after it is placed in users' hands. What good is a monthly report if you do not get it until the end of the following month? How can you change what you are doing based on data that old? The reporting backlog has never dissipated within information system departments; typically it has grown. Granting users access to data on a more timely basis allows them to better perform their business tasks. It can also assist in reducing the reporting backlog, because users take more responsibility for the reporting process.
- **Improved performance and productivity:** Removing information systems professionals from the reporting loop and empowering users results in internal efficiency. Imagine that you had no operational systems and had to hunt down the person who recorded a transaction to better understand how to improve the business process or determine whether a promotion was successful. The truth is that all we have done is automate this nightmare with the current operational systems. Users have no central sources for information and must search all of the operational systems for the data that is required to answer their questions. A data warehouse assists in eliminating information backlogs, reporting backlogs, information system performance problems and so on by improving the efficiency of the process, eliminating much of the information search missions.

It should be noted that even with a data warehouse, companies still require two distinct kinds of reporting: that which provides notification of operational conditions needing response, and that which provides general information, often summarised, about business operations. The notification-style reports should still be derived from operational systems, because detecting and reporting these conditions is part of the process of responding to business events. The general information reports, indicating operational performance typically used in analysing the business, are managed by a data warehouse.

Review question 1

Analyse the differences between data warehousing and operational systems, and discuss the importance of the separation of the two systems.

Activity 1

Research how a business in your area of interest has benefited from the data warehousing technology.

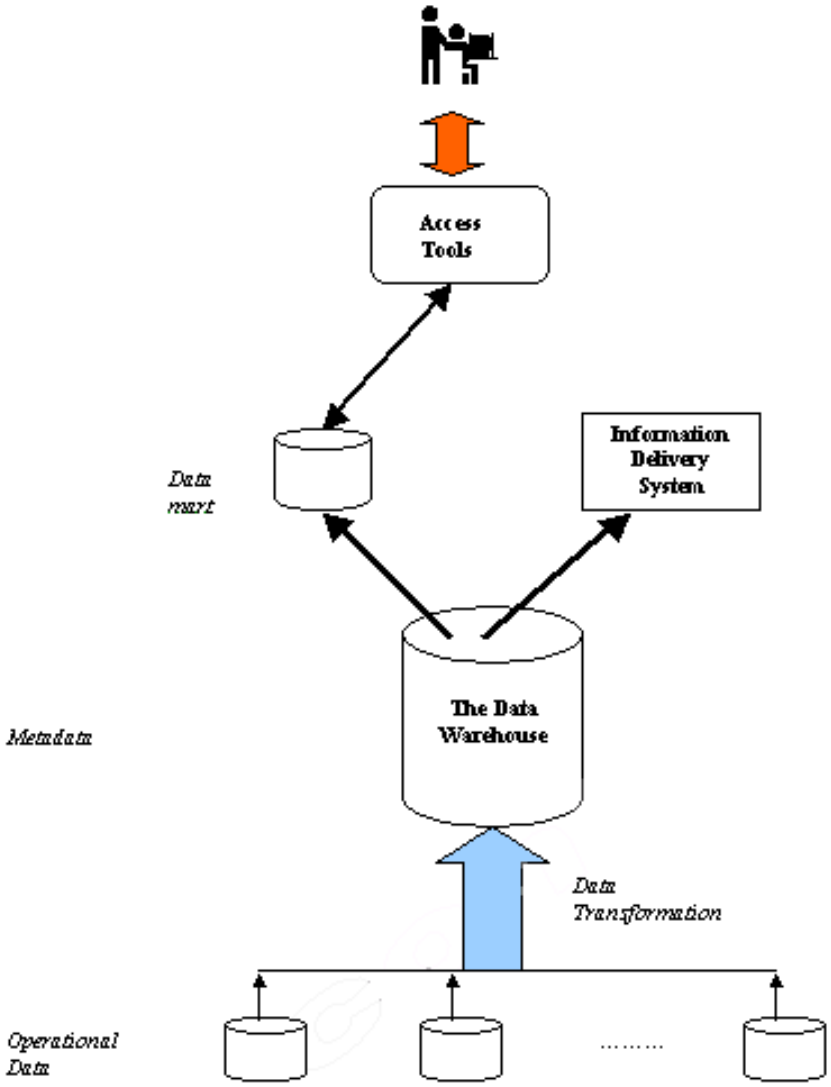
Data warehouse architecture

Data warehouses provide a means to make information available for decision-making. An effective data warehousing strategy must deal with the complexities of modern enterprises. Data is generated everywhere, and controlled by different operational systems and data storage mechanisms. Users demand access to

data anywhere and any time, and data must be customised to their needs and requirements. The function of a data warehouse is to prepare the current transactions from operational systems into data with a historical context, required by the users of the data warehouse.

Overall architecture

The general data warehouse architecture is based on a Relational database management system server that functions as the central repository for informational data. In the data warehouse architecture, operational data and processing is completely separate from data warehouse processing. This central information repository is surrounded by a number of key components designed to make the entire environment functional, manageable and accessible by both the operational systems that source data into the warehouse and by end-user query and analysis tools. The diagram below depicts such a general architecture:



Typically, the source data for the warehouse is coming from the operational applications, or an operational data store (ODS). As the data enters the data warehouse, it is transformed into an integrated structure and format. The transformation process may involve conversion, summarisation, filtering and condensation of data. Because data within the data warehouse contains a large historical component (sometimes over 5 to 10 years), the data warehouse must

be capable of holding and managing large volumes of data as well as different data structures for the same database over time.

The data warehouse

The central data warehouse database is a cornerstone of the data warehousing environment. This type of database is mostly implemented using a Relational DBMS (RDBMS). However, a warehouse implementation based on traditional RDBMS technology is often constrained by the fact that traditional RDBMS implementations are optimised for transactional database processing. Certain data warehouse attributes, such as very large database size, ad hoc query processing and the need for flexible user view creation, including aggregates, multi-table joins and drill-downs, have become drivers for different technological approaches to the data warehouse database.

Data transformation

A significant portion of the data warehouse implementation effort is spent extracting data from operational systems and putting it in a format suitable for information applications that will run off the data warehouse. The data-sourcing, clean-up, transformation and migration tools perform all of the conversions, summarisation, key changes, structural changes and condensations needed to transform disparate data into information that can be used by the decision support tool. It also maintains the metadata. The functionality of data transformation includes:

- Removing unwanted data from operational databases.
- Converting to common data names and definitions.
- Calculating summaries and derived data.
- Establishing defaults for missing data.
- Accommodating source data definition changes.

The data-sourcing, clean-up, extraction, transformation and migration tools have to deal with some important issues as follows:

- Database heterogeneity: DBMSs can vary in data models, data access languages, data navigation operations, concurrency, integrity, recovery, etc.
- Data heterogeneity: This is the difference in the way data is defined and used in different models – there are homonyms, synonyms, unit incompatibility, different attributes for the same entity, and different ways of modelling the same fact.

Metadata

A crucial area of data warehousing is metadata, which is a kind of data that describes the data warehouse itself. Within a data warehouse, metadata describes and locates data components, their origins (which may be either the operational systems or the data warehouse), and their movement through the data warehouse process. The data access, data stores and processing information will have associated descriptions about the data and processing – the inputs, calculations and outputs – documented in the metadata. This metadata should be captured within the data architecture and managed from the beginning of a data warehouse project. The metadata repository should contain information such as that listed below:

- Description of the data model.
- Description of the layouts used in the database design.
- Definition of the primary system managing the data items.
- A map of the data from the system of record to the other locations in the data warehouse, including the descriptions of transformations and aggregations.
- Specific database design definitions.
- Data element definitions, including rules for derivations and summaries.

It is through metadata that a data warehouse becomes an effective tool for an overall enterprise. This repository of information will tell the story of the data: where it originated, how it has been transformed, where it went and how often – that is, its genealogy or artefacts. Technically, the metadata will also improve the maintainability and manageability of a warehouse by making impact analysis information and entity life histories available to the support staff.

Equally important, metadata provides interactive access to users to help understand content and find data. Thus, there is a need to create a metadata interface for users.

One important functional component of the metadata repository is the information directory. The content of the information directory is the metadata that helps users exploit the power of data warehousing. This directory helps integrate, maintain and view the contents of the data warehousing system. From a technical requirements point of view, the information directory and the entire metadata repository should:

- Be a gateway to the data warehouse environment, and therefore, should be accessible from any platform via transparent and seamless connections.
- Support an easy distribution and replication of its content for high performance and availability.

- Be searchable by business-oriented keywords.
- Act as a launch platform for end-user data access and analysis tools.
- Support the sharing of information objects such as queries, reports, data collections and subscriptions between users.
- Support a variety of scheduling options for requests against the data warehouse, including on-demand, one-time, repetitive, event-driven and conditional delivery (in conjunction with the information delivery system).
- Support the distribution of query results to one or more destinations in any of the user-specified formats (in conjunction with the information delivery system).
- Support and provide interfaces to other applications such as e-mail, spreadsheet and schedules.
- Support end-user monitoring of the status of the data warehouse environment.

At a minimum, the information directory components should be accessible by any Web browser, and should run on all major platforms, including MS Windows, Windows NT and UNIX. Also, the data structures of the metadata repository should be supported on all major Relational database platforms.

These requirements define a very sophisticated repository of metadata information. In reality, however, existing products often come up short when implementing these requirements.

Access tools

The principal purpose of data warehousing is to provide information to business users for strategic decision-making. These users interact with the data warehouse using front-end tools. Although ad hoc requests, regular reports and custom applications are the primary delivery vehicles for the analysis done in most data warehouses, many development efforts of data warehousing projects are focusing on exceptional reporting also known as alerts, which alert a user when a certain event has occurred. For example, if a data warehouse is designed to access the risk of currency trading, an alert can be activated when a certain currency rate drops below a predefined threshold. When an alert is well synchronised with the key objectives of the business, it can provide warehouse users with a tremendous advantage.

The front-end user tools can be divided into five major groups:

1. Data query and reporting tools.
2. Application development tools.
3. Executive information systems (EIS) tools.

4. Online analytical processing (OLAP) tools.
5. Data mining tools.

Query and reporting tools

This category can be further divided into two groups: reporting tools and managed query tools. Reporting tools can be divided into production reporting tools and desktop report writers.

Production reporting tools let companies generate regular operational reports or support high-volume batch jobs, such as calculating and printing pay cheques. Report writers, on the other hand, are affordable desktop tools designed for end-users.

Managed query tools shield end-users from the complexities of SQL and database structures by inserting a metalayer between users and the database. The metalayer is the software that provides subject-oriented views of a database and supports point-and-click creation of SQL. Some of these tools proceed to format the retrieved data into easy-to-read reports, while others concentrate on on-screen presentations. These tools are the preferred choice of the users of business applications such as segment identification, demographic analysis, territory management and customer mailing lists. As the complexity of the questions grows, these tools may rapidly become inefficient.

Application development tools

Often, the analytical needs of the data warehouse user community exceed the built-in capabilities of query and reporting tools. Organisations will often rely on a true and proven approach of in-house application development, using graphical data access environments designed primarily for client-server environments. Some of these application development platforms integrate well with popular OLAP tools, and can access all major database systems, including Oracle and IBM Informix.

Executive information systems (EIS) tools

The target users of EIS tools are senior management of a company. The tools are used to transform information and present that information to users in a meaningful and usable manner. They support advanced analytical techniques and free-form data exploration, allowing users to easily transform data into information. EIS tools tend to give their users a high-level summarisation of key performance measures to support decision-making.

OLAP

These tools are based on concepts of multidimensional database and allow a sophisticated user to analyse the data using elaborate, multidimensional and complex views. Typical business applications for these tools include product performance and profitability, effectiveness of a sales program or a marketing campaign, sales forecasting and capacity planning. These tools assume that the data is organised in a multidimensional model, which is supported by a special multidimensional database or by a Relational database designed to enable multidimensional properties.

Data mining tools

Data mining can be defined as the process of discovering meaningful new correlation, patterns and trends by digging (mining) large amounts of data stored in a warehouse, using artificial intelligence (AI) and/or statistical/mathematical techniques. The major attraction of data mining is its ability to build predictive rather than retrospective models. Using data mining to build predictive models for decision-making has several benefits. First, the model should be able to explain why a particular decision was made. Second, adjusting a model on the basis of feedback from future decisions will lead to experience accumulation and true organisational learning. Finally, a predictive model can be used to automate a decision step in a larger process. For example, using a model to instantly predict whether a customer will default on credit card payments will allow automatic adjustment of credit limits rather than depending on expensive staff making inconsistent decisions. Data mining will be discussed in more detail later on in the chapter.

Data visualisation

Data warehouses are causing a surge in popularity of data visualisation techniques for looking at data. Data visualisation is not a separate class of tools; rather, it is a method of presenting the output of all the previously mentioned tools in such a way that the entire problem and/or the solution (e.g. a result of a Relational or multidimensional query, or the result of data mining) is clearly visible to domain experts and even casual observers.

Data visualisation goes far beyond simple bar and pie charts. It is a collection of complex techniques that currently represent an area of intense research and development, focusing on determining how to best display complex relationships and patterns on a two-dimensional (flat) computer monitor. Similar to medical imaging research, current data visualisation techniques experiment with various colours, shapes, 3D imaging and sound, and virtual reality to help users really see and feel the problem and its solutions.

Data marts

The concept of data mart is causing a lot of excitement and attracts much attention in the data warehouse industry. Mostly, data marts are presented as an inexpensive alternative to a data warehouse that takes significantly less time and money to build. However, the term means different things to different people. A rigorous definition of data mart is that it is a data store that is subsidiary to a data warehouse of integrated data. The data mart is directed at a partition of data (often called subject area) that is created for the use of a dedicated group of users. A data mart could be a set of denormalised, summarised or aggregated data. Sometimes, such a set could be placed on the data warehouse database rather than a physically separate store of data. In most instances, however, a data mart is a physically separate store of data and is normally resident on a separate database server, often on the local area network serving a dedicated user group.

Data marts can incorporate different techniques like OLAP or data mining. All these types of data marts are called dependent data marts because their data content is sourced from the data warehouse. No matter how many are deployed and what different enabling technologies are used, different users are all accessing the information views derived from the same single integrated version of the data (i.e. the underlying warehouse).

Unfortunately, the misleading statements about the simplicity and low cost of data marts sometimes result in organisations or vendors incorrectly positioning them as an alternative to the data warehouse. This viewpoint defines independent data marts that in fact represent fragmented point solutions to a range of business problems. It is missing the integration that is at the heart of the data warehousing concept: data integration. Each independent data mart makes its own assumptions about how to consolidate data, and as a result, data across several data marts may not be consistent.

Moreover, the concept of an independent data mart is dangerous – as soon as the first data mart is created, other organisations, groups and subject areas within the enterprise embark on the task of building their own data marts. As a result, you create an environment in which multiple operational systems feed multiple non-integrated data marts that are often overlapping in data content, job scheduling, connectivity and management. In other words, you have transformed a complex many-to-one problem of building a data warehouse from operational data sources into a many-to-many sourcing and management nightmare. Another consideration against independent data marts is related to the potential scalability problem.

To address data integration issues associated with data marts, a commonly recommended approach is as follows. For any two data marts in an enterprise, the common dimensions must conform to the equality and roll-up rule, which states that these dimensions are either the same or that one is a strict roll-up

of another.

Thus, in a retail store chain, if the purchase orders database is one data mart and the sales database is another data mart, the two data marts will form a coherent part of an overall enterprise data warehouse if their common dimensions (e.g. time and product) conform. The time dimension from both data marts might be at the individual day level, or conversely, one time dimension is at the day level but the other is at the week level. Because days roll up to weeks, the two time dimensions are conformed. The time dimensions would not be conformed if one time dimension were weeks and the other a fiscal quarter. The resulting data marts could not usefully coexist in the same application.

The key to a successful data mart strategy is the development of an overall scalable data warehouse architecture, and the key step in that architecture is identifying and implementing the common dimensions.

Information delivery system

The information delivery system distributes warehouse-stored data and other information objects to other data warehouses and end-user products such as spreadsheets and local databases. Delivery of information may be based on time of day, or on the completion of an external event. The rationale for the delivery system component is based on the fact that once the data warehouse is installed and operational, its users don't have to be aware of its location and maintenance. All they may need is the report or an analytical view of data, at a certain time of the day, or based on a particular, relevant event. And of course, such a delivery system may deliver warehouse-based information to end users via the Internet. A Web-enabled information delivery system allows users dispersed across continents to perform sophisticated business-critical analysis, and to engage in collective decision-making that is based on timely and valid information.

Review question 2

- Discuss the functionality of data transformation in a data warehouse system.
- What is metadata? How is it used in a data warehouse system?
- What is a data mart? What are the drawbacks of using independent data marts?

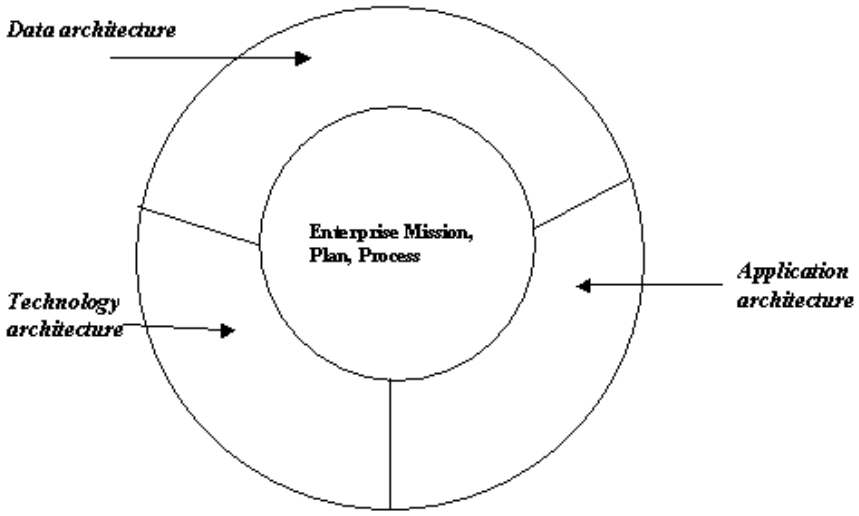
Data Warehouse Development

Data warehouse blueprint

A data warehouse blueprint should include clear documentation of the following items:

- **Requirements:** What does the business want from the data warehouse?
- **Architecture blueprint:** How will you deliver what the business wants?
- **Development approach:** What is a clear definition of phased delivery cycles, including architectural review and refinement processes?

The blueprint document essentially translates an enterprise's mission, goals and objectives for the data warehouse into a logical technology architecture composed of individual sub-architectures for the application, data and technology components of a data warehouse, as shown below:

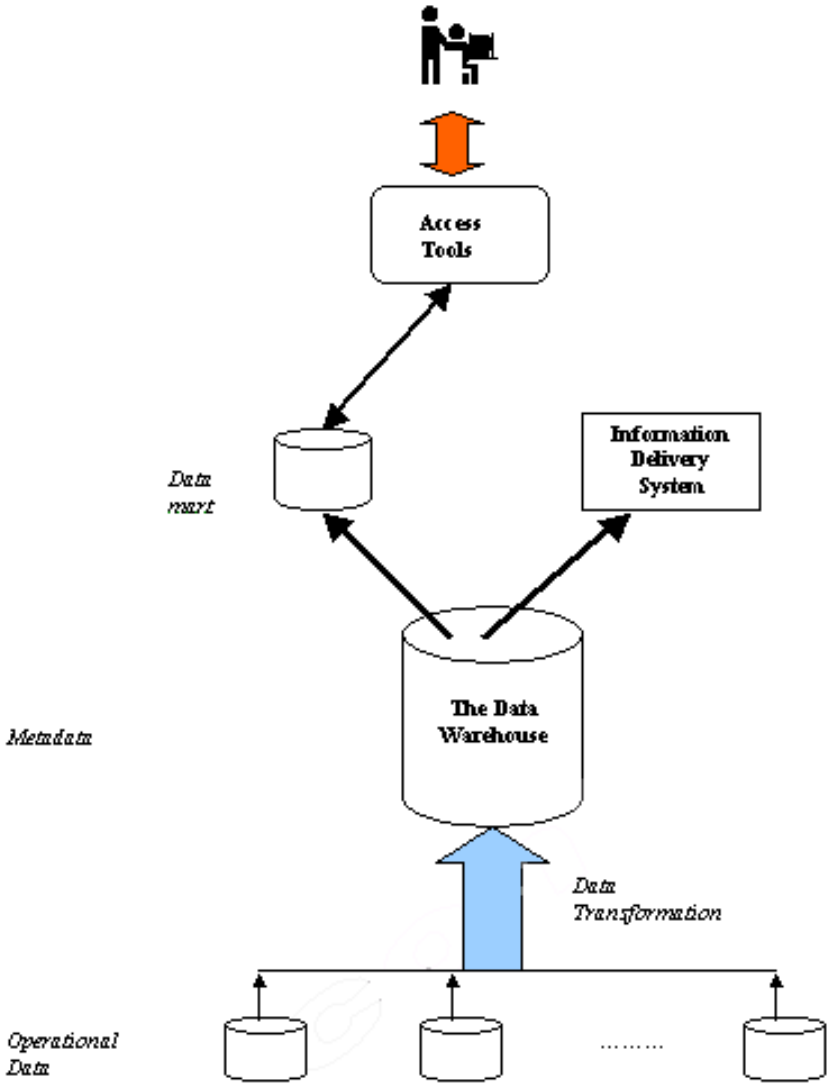


An architecture blueprint is important, because it serves as a road map for all development work and as a guide for integrating the data warehouse with legacy systems. When the blueprint is understood by the development staff, decisions become much easier. The blueprint should be developed in a logical sense rather than in a physical sense. For the database components, for example, you will state things like "the data store for the data warehouse will support an easy-to-use data manipulation language that is standard in the industry, such as SQL". This is a logical architecture-product requirement. When you implement

the data warehouse, this could be Sybase SQL Server or Oracle. The logical definition allows your implementations to grow as technology evolves. If your business requirements do not change in the next three to five years, neither will your blueprint.

Data architecture

As shown in the 'Overall architecture' section earlier, a data warehouse is presented as a network of databases. The sub-components of the data architecture will include the enterprise data warehouse, metadata repository, data marts and multidimensional data stores. These sub-components are documented separately, because the architecture should present a logical view of them. It is for the data warehouse implementation team to determine the proper way to physically implement the recommended architecture. This suggests that the implementation may well be on the same physical database, rather than separate data stores, as shown below:



Volumetrics

A number of issues need to be considered in the logical design of the data architecture of a data warehouse. Metadata, which has been discussed earlier, is the first issue, followed by the volume of data that will be processed and

housed by a data warehouse. The latter is probably the biggest factor that determines the technology utilised by the data warehouse to manage and store the information. The volume of data affects the warehouse in two aspects: the overall size and ability to load.

Too often, people design their warehouse load processes only for mass loading of the data from the operational systems to the warehouse system. This is inadequate. When defining your data architecture, you should devise a solution that allows mass loading as well as incremental loading. Mass loading is typically a high-risk area; the database management systems can only load data at a certain speed. Mass loading often forces downtime, but we want users to have access to a data warehouse with as few interruptions as possible.

Transformation

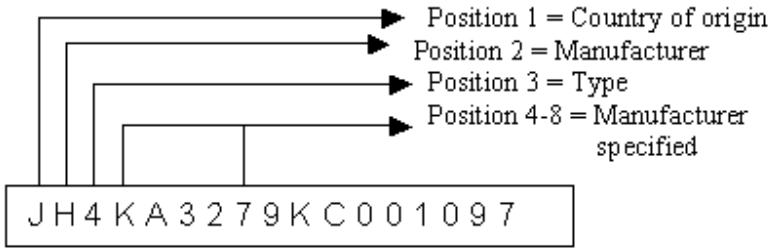
A data architecture needs to provide a clear understanding of transformation requirements that must be supported, including logic and complexity. This is one area in which the architectural team will have difficulty finding commercially available software to manage or assist with the process. Transformation tools and standards are currently immature. Many tools were initially developed to assist companies in moving applications away from mainframes. Operational data stores are vast and varied. Many data stores are unsupported by these transformation tools. The tools support the popular database engines, but do nothing to advance your effort with little-known or unpopular databases. It is better to evaluate and select a transformational tool or agent that supports a good connectivity tool, such as Information Builder's EDA/SQL, rather than one that supports a native file access strategy. With an open connectivity product, your development teams can focus on multiplatform, multidatabase transformations.

Data cleansing

In addition to finding tools to automate the transformation process, the developers should also evaluate the complexity behind data transformations. Most legacy data stores lack standards and have anomalies that can cause enormous difficulties. Again, tools are evolving to assist you in automating transformations, including complex issues such as buried data, lack of legacy standards and non-centralised key data.

- **Buried data**

  Often, legacy systems use composite keys to uniquely define data. Although these fields appear as one in a database, they represent multiple pieces of information. The diagram below illustrates buried data by showing a vehicle identification number that contains many pieces of information.



- **Lack of legacy standards**

  Items such as descriptions, names, labels and keys have typically been managed on an application-by-application basis. In many legacy systems, such fields lack clear definition. For example, data in the name field sometimes is haphazardly formatted (Brent Thomas; Elizabeth A. Hammergreen; and Herny, Ashley). Moreover, application software providers may offer user-oriented fields, which can be used and defined as required by the customer.

- **Non-centralised key data**

  As companies have evolved through acquisition or growth, various systems have taken ownership of data that may not have been in their scope. This is especially true for companies that can be characterised as heavy users of packaged application software and those that have grown through acquisition. Notice how the non-centralised cust_no field varies from one database to another for a hypothetical company represented below:

Cust_no	Company	Location
001ABC	XYZ Ltd	North
01-XYZ	XYZ Ltd	East
ABC001	XYZ Ltd	South west

The ultimate goal of a transformation architecture is to allow the developers to create a repeatable transformation process. You should make sure to clearly define your needs for data synchronisation and data cleansing.

Data architecture requirements

As a summary of the data architecture design, this section lists the main requirements placed on a data warehouse.

- **Subject-oriented data:** Data that is contained within a data warehouse should be organised by subject. For example, if your data warehouse focuses on sales and marketing processes, you need to generate data about customers, prospects, orders, products, and so on. To completely define a subject area, you may need to draw upon data from multiple operational systems. To derive the data entities that clearly define the sales and marketing process of an enterprise, you might need to draw upon an order entry system, a sales force automation system, and various other applications.
- **Time-based data:** Data in a data warehouse should relate specifically to a time period, allowing users to capture data that is relevant to their analysis period. Consider an example in which a new customer was added to an order entry system with a primary contact of John Doe on 2/11/99. This customer's data was changed on 4/11/99 to reflect a new primary contact of Jane Doe. In this scenario, the data warehouse would contain the two contact records shown in the following table:

Cust_ID	Contact_ID	Last_Name	First_Name	Time_Stamp
1999120601	01	Doe	John	2/11/99
1999120601	01	Doe	Jane	4/11/99

- **Update processing:** A data warehouse should contain data that represents closed operational items, such as fulfilled customer order. In this sense, the data warehouse will typically contain little or no update processing. Typically, incremental or mass loading processes are run to insert data into the data warehouse. Updating individual records that are already in the data warehouse will rarely occur.
- **Transformed and scrubbed data:** Data that is contained in a data warehouse should be transformed, scrubbed and integrated into user-friendly subject areas.
- **Aggregation:** Data needs to be aggregated into and out of a data warehouse. Thus, computational requirements will be placed on the entire data warehousing process.
- **Granularity:** A data warehouse typically contains multiple levels of granularity. It is normal for the data warehouse to be summarised and contain less detail than the original operational data; however, some data warehouses require dual levels of granularity. For example, a sales manager

may need to understand how sales representatives in his or her area perform a forecasting task. In this example, monthly summaries that contain the data associated with the sales representatives' forecast and the actual orders received are sufficient; there is no requirement to see each individual line item of an order. However, a retailer may need to wade through individual sales transactions to look for correlations that may show people tend to buy soft drinks and snacks together. This need requires more details associated with each individual purchase. The data required to fulfil both of these requests may exist, and therefore, the data warehouse might be built to manage both summarised data to fulfil a very rapid query and the more detailed data required to fulfil a lengthy analysis process.

- **Metadata management:** Because a data warehouse pools information from a variety of sources and the data warehouse developers will perform data gathering on current data stores and new data stores, it is required that storage and management of metadata be effectively done through the data warehouse process.

Application architecture

An application architecture determines how users interact with a data warehouse. To determine the most appropriate application architecture for a company, the intended users and their skill levels should be assessed. Other factors that may affect the design of the architecture include technology currently available and budget constraints. In any case, however, the architecture must be defined logically rather than physically. The classification of users will help determine the proper tools to satisfy their reporting and analysis needs. A sampling of user category definitions is listed below:

- **Power users:** Technical users who require little or no support to develop complex reports and queries. This type of user tends to support other users and analyse data through the entire enterprise.
- **Frequent users:** Less technical users who primarily interface with the power users for support, but sometimes require the IT department to support them. These users tend to provide management reporting support up to the division level within an enterprise, a narrower scope than for power users.
- **Casual users:** These users touch the system and computers infrequently. They tend to require a higher degree of support, which normally includes building predetermined reports, graphs and tables for their analysis purpose.

Requirements of tools

Tools must be made available to users to access a data warehouse. These tools should be carefully selected so that they are efficient and compatible with other parts of the architecture and standards.

- **Executive information systems (EIS):** As mentioned earlier, these tools transform information and present that information to users in a meaningful and usable manner. They support advanced analytical techniques and free-form data exploration, allowing users to easily transform data into information. EIS tools tend to give their users a high-level summarisation of key performance measures to support decision-making. These tools fall into the big-button syndrome, in which an application development team builds a nice standard report with hooks to many other reports, then presents this information behind a big button. When a user clicks the button, magic happens.
- **Decision support systems (DSS):** DSS tools are intended for more technical users, who require more flexibility and ad hoc analytical capabilities. DSS tools allow users to browse their data and transform it into information. They avoid the big button syndrome.
- **Ad hoc query and reporting:** The purpose of EIS and DSS applications is to allow business users to analyse, manipulate and report on data using familiar, easy-to-use interfaces. These tools conform to presentation styles that business people understand and with which they are comfortable. Unfortunately, many of these tools have size restrictions that do not allow them to access large stores or to access data in a highly normalised structure, such as a Relational database, in a rapid fashion; in other words, they can be slow. Thus, users need tools that allow for more traditional reporting against Relational, or two-dimensional, data structures. These tools offer database access with limited coding and often allow users to create read-only applications. Ad hoc query and reporting tools are an important component within a data warehouse tool suite. Their greatest advantage is contained in the term 'ad hoc'. This means that decision makers can access data in an easy and timely fashion.
- **Production report writer:** A production report writer allows the development staff to build and deploy reports that will be widely exploited by the user community in an efficient manner. These tools are often components within fourth generation languages (4GLs) and allow for complex computational logic and advanced formatting capabilities. It is best to find a vendor that provides an ad hoc query tool that can transform itself into a production report writer.
- **Application development environments (ADE):** ADEs are nothing new, and many people overlook the need for such tools within a data warehouse tool suite. However, you will need to develop some presentation system for your users. The development, though minimal, is still a requirement, and it is advised that data warehouse development projects

standardise on an ADE. Example tools include Microsoft Visual Basic and Powersoft Powerbuilder. Many tools now support the concept of cross-platform development for environment such as Windows, Apple Macintosh and OS/2 Presentation Manager. Every data warehouse project team should have a standard ADE in its arsenal.

- **Other tools:** Although the tools just described represent minimum requirements, you may find a need for several other speciality tools. These additional tools include OLAP, data mining and managed query environments.

Technology architecture

It is in the technology architecture section of the blueprint that hardware, software and network topology are specified to support the implementation of the data warehouse. This architecture is composed of three major components - clients, servers and networks – and the software to manage each of them.

- **Clients:** The client technology component comprises the devices that are utilised by users. These devices can include workstations, personal computers, personal digital assistants and even beepers for support personnel. Each of these devices has a purpose being served by a data warehouse. Conceptually, the client either contains software to access the data warehouse (this is the traditional client in the client-server model and is known as a fat client), or it contains very little software and accesses a server that contains most of the software required to access a data warehouse. The later approach is the evolving Internet client model, known as a thin client and fat server.
- **Servers:** The server technology component includes the physical hardware platforms as well as the operating systems that manage the hardware. Other components, typically software, can also be grouped within this component, including database management software, application server software, gateway connectivity software, replication software and configuration management software.
- **Networks:** The network component defines the transport technologies needed to support communication activities between clients and servers. This component includes requirements and decisions for wide area networks (WANs), local area networks (LANs), communication protocols and other hardware associated with networks, such as bridges, routers and gateways.

Review question 3

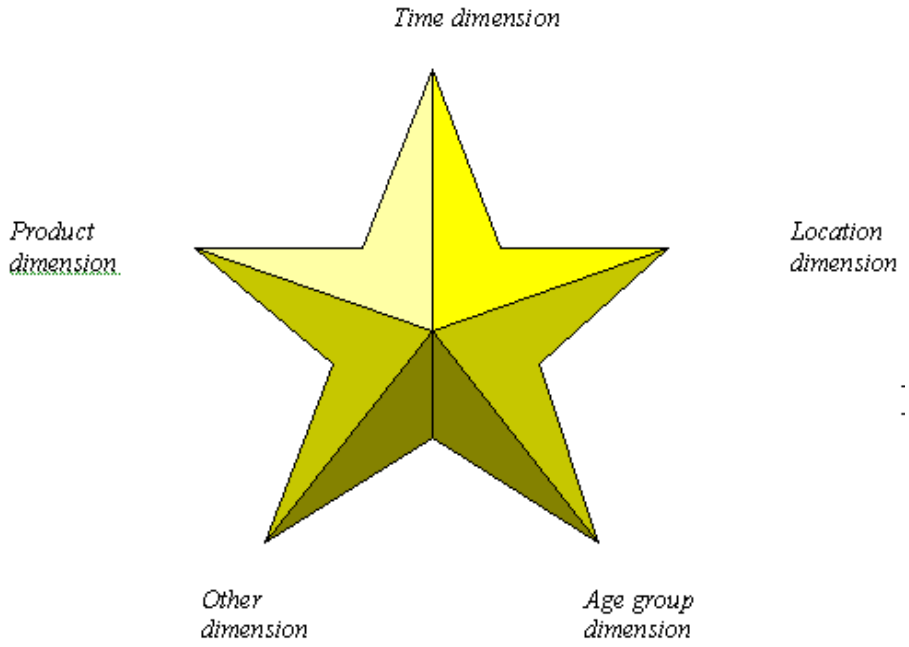
- What are the problems that you may encounter in the process of data cleansing?

- Describe the three components of the technology architecture of a data warehousing system.

Star schema design

Data warehouses can best be modelled using a technique known as star schema modelling. It defines data entities in a way that supports the decision-makers' view of a business and that reflects the important operational aspects of the business. A star schema contains three logical entities: dimension, measure and category detail (or category for short).

A star schema is optimised to queries, and therefore provides a database design that is focused on rapid response to users of the system. Also, the design that is built from a star schema is not as complicated as traditional database designs. Hence, the model will be more understandable for users of the system. Also, users will be able to better understand the navigation paths available to them through interpreting the star schema. This logical database design's name hails from a visual representation derived from the data model: it forms a star, as shown below:



The star schema defines the join paths for how users access the facts about their

business. In the figure above, for example, the centre of the star could represent product sales revenues that could have the following items: actual sales, budget and sales forecast. The true power of a star schema design is to model a data structure that allows filtering, or reduction in result size, of the massive measure entities during user queries and searches. A star schema also provides a usable and understandable data structure, because the points of the star, or dimension entities, provide a mechanism by which a user can filter, aggregate, drill down, and slice and dice the measurement data in the centre of the star.

Entities within a data warehouse

A star schema, like the data warehouse it models, contains three types of logical entities: measure, dimension and category detail. Each of these entities is discussed separately below.

Measure entities

Within a star schema, the centre of the star – and often the focus of the users' query activity – is the measure entity. A measure entity is represented by a rectangle and is placed in the centre of a star schema diagram.

A sample of raw measure data is shown below:

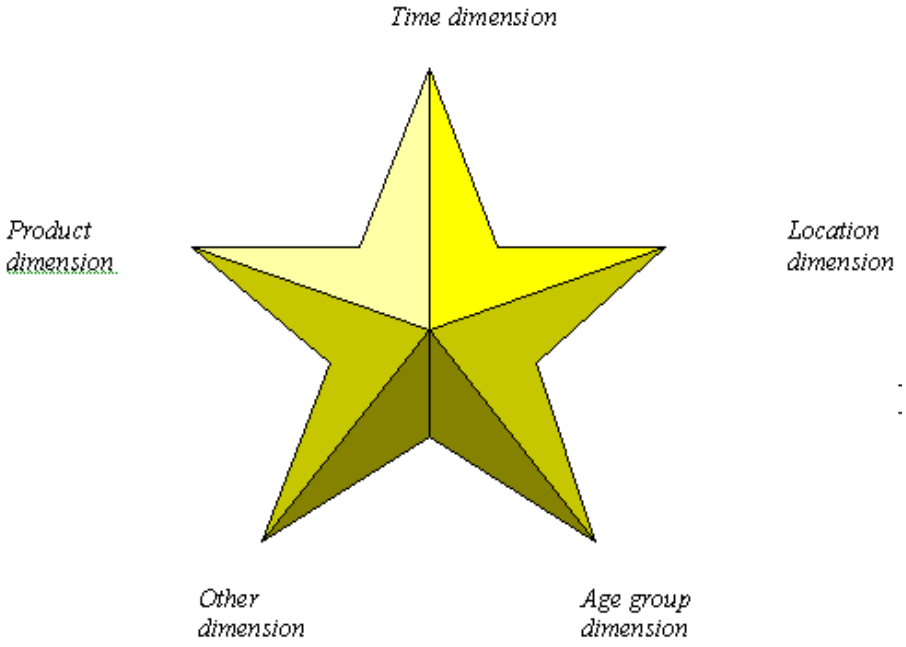
Month	Branch	Product	Sales Forecast	Sales Actual	Variance
199901	ABC	COLA	200000	1900000	-10000
199901	XYZ	COLA	150000	1550000	50000
199901	PQR	COLA	125000	1050000	-20000
.....					.....

The data contained in a measure entity is factual information from which users derive 'business intelligence'. This data is therefore often given synonymous names to measure, such as key business measures, facts, metrics, performance measures and indicators. The measurement data provides users with quantitative data about a business. This data is numerical information that the users desire to monitor, such as dollars, pounds, degrees, counts and quantities. All of these categories allow users to look into the corporate knowledge base and understand the good, bad and ugly of the business process being measured.

The data contained within measure entities grows large over time, and therefore is typically of greatest concern to the technical support personnel, database administrators and system administrators.

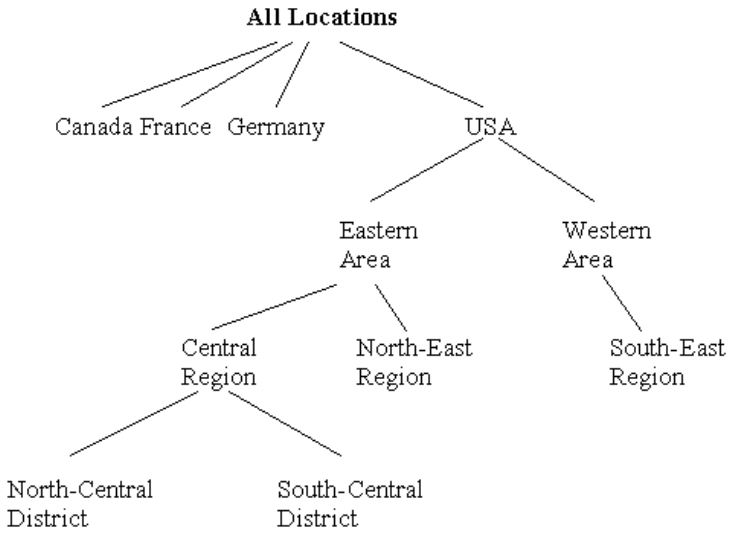
Dimension entities

Dimension entities are graphically represented by diamond-shaped squares, and placed at the points of the star. Dimension entities are much smaller entities compared with measure entities. The dimensions and their associated data allow users of a data warehouse to browse measurement data with ease of use and familiarity. These entities assist users in minimising the rows of data within a measure entity and in aggregating key measurement data. In this sense, these entities filter data or force the server to aggregate data so that fewer rows are returned from the measure entities. With a star schema model, the dimension entities are represented as the points of the star, as demonstrated in the diagram below, by the time, location, age group, product and other dimensions:



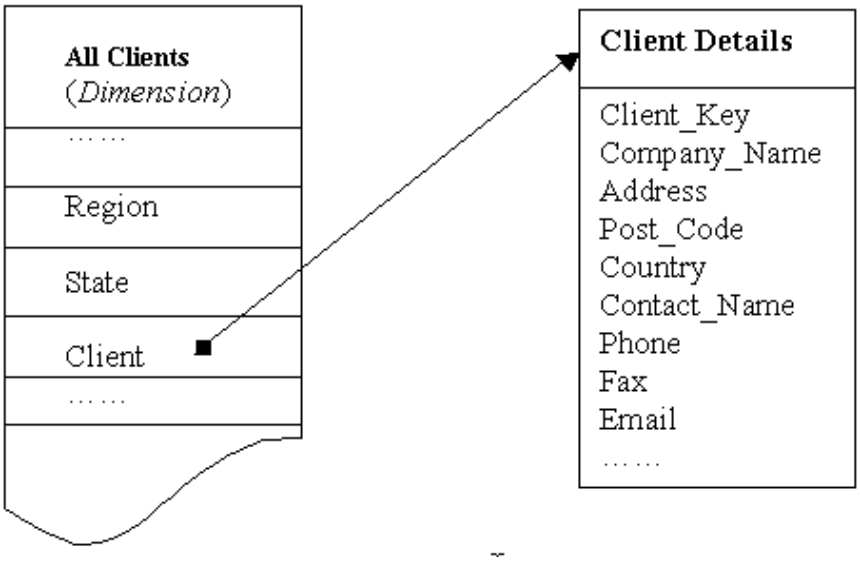
The diagram below illustrates an example of dimension data and a hierarchy representing the contents of a dimension entity:

Country Key	Area Key	Region Key	District Key	Country	Area	Region	District
USA	EAST	CEN	NC	USA	Eastern	Central	North-Central
USA	EAST	CEN	SC	USA	Eastern	Central	South-Central
USA	EAST	NE					
USA	WEST	SE					
CANADA							
FRANCE							
GERMANY							



Category detail entities

Each cell in a dimension is a category and represents an isolated level within a dimension that might require more detailed information to fulfil a user's requirement. These categories that require more detailed data are managed within category detail entities. These entities have textual information that supports the measurement data and provides more detailed or qualitative information to assist in the decision-making process. The diagram below illustrates the need for a client category detail entity within the All Clients dimension:



The stop sign symbol is usually used to graphically depict category entities, because users normally flow through the dimension entities to get the measure entity data, then stop their investigation with supporting category detail data.

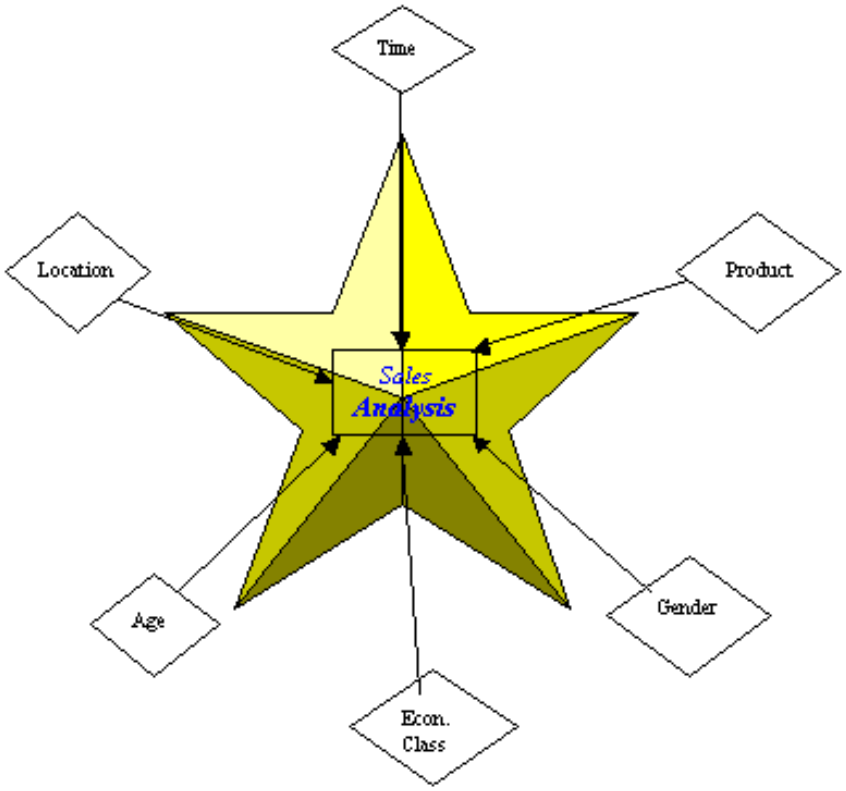
Translating information into a star schema

During the data gathering process, an information package can be constructed, based on which star schema is formed. The table below shows an information package diagram ready for translation into a star schema. As can be seen from the table, there are six dimensions, and within each there are different numbers of categories. For example, the All Locations dimension has five categories while All Genders has one. The number within each category denotes the number of instances the category may have. For example, the All Time Periods will cover five different years with 20 quarters and 60 months. Gender will include male, female and unknown.

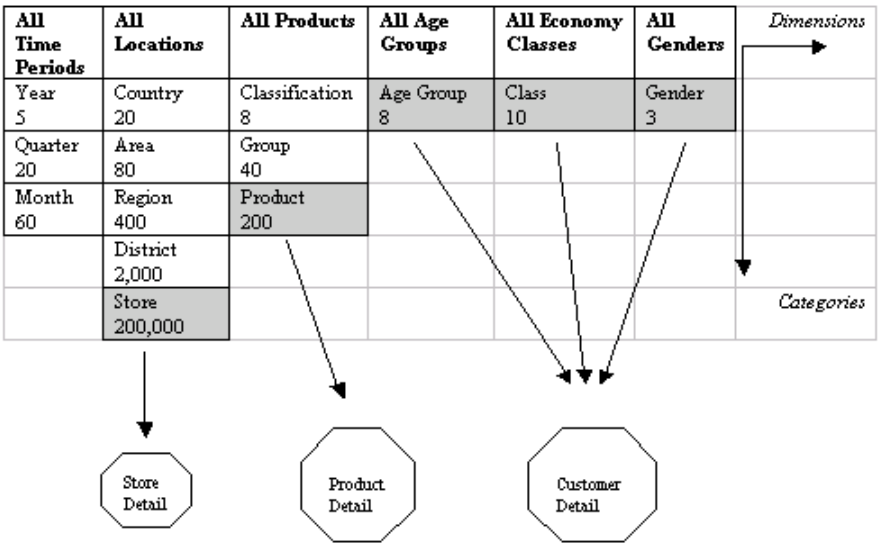
To define the logical measure entity, take the lowest category, or cell, within each dimension along with each of the measures and take them as the measure entity. For example, the measure entity translated from the table below would be Month, Store, Product, Age Group, Class and Gender with the measures Forecast Sales, Budget Sales, Actual Sales and Forecast Variance (calculated), and Budget Variance (calculated). They could be given a name Sales Analysis and put in the centre of the star schema in a rectangle.

All Time Periods	All Locations	All Products	All Age Groups	All Economy Classes	All Genders	*Dimensions* →
Year 5	Country 20	Classification 8	Age Group 8	Class 10	Gender 3	
Quarter 20	Area 80	Group 40				
Month 60	Region 400	Product 200				
	District 2,000					
	Store 200,000					*Categories* ↓
Measures/Facts: Forecast Sales, Budget Sales, Actual Sales, Forecast Variance (*calc.*), Budget Variance (*calc.*)						

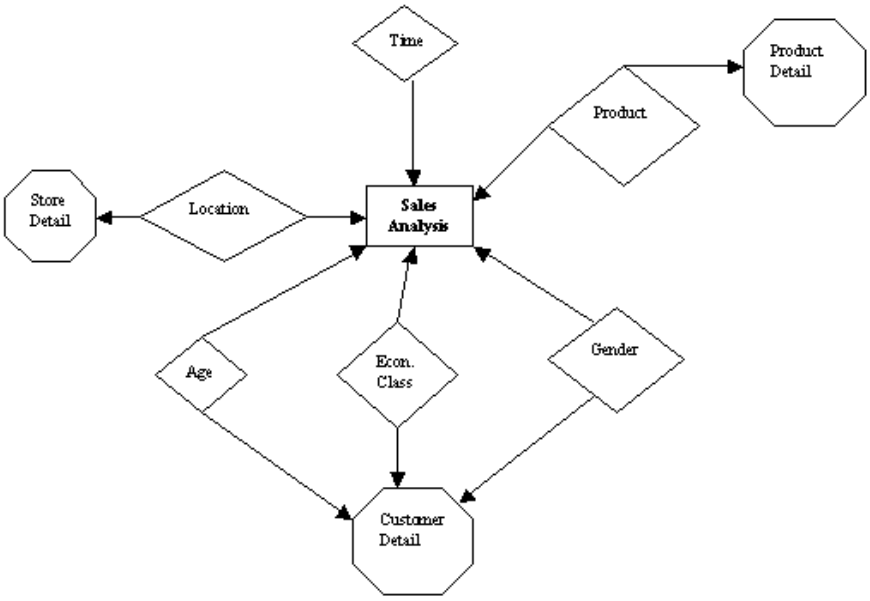
Each column of an information package in the table above defines a dimension entity and is placed on the periphery of the star of a star schema, symbolising the points of the star. Following the placement of the dimension entities, you want to define the relationships that they have with the measure entity. Because dimension entities always require representation within the measure entity, there always is a relationship. The relationship is defined over the lowest-level detail category for the logical model; that is, the last cell in each dimension. These relationships possess typically one-to-many cardinality; in other words, one dimension entity exists for many within the measures. For example, you may hope to make many product sales (Sales Analysis) to females (Gender) within the star model illustrated in the diagram below. In general, these relationships can be given an intuitive explanation such as: "Measures based on the dimension". In the diagram below, for example, the relationship between Location (the dimension entity) and Sales Analysis (the measure entity) means "Sales Analysis based on Location".



The final step in forming a star schema is to define the category detail entity. Each individual cell in an information package diagram must be evaluated and researched to determine if it qualifies as a category detail entity. If the user has a requirement for additional information about a category, this formulates the requirement for a category detail entity. These detail entities become extensions of dimension entities, as illustrated below:



We need to know more detailed information about data such as Store, Product and customer categories (i.e. Age, Class and Gender). These detail entities (Store Detail, Product Detail and Customer Detail), having been added to the current star schema, now appear as shown below:



Review question 4

What are the three types of entities in a star schema and how are they used to model a data warehouse?

Exercise 1

An information package of a promotional analysis is shown below. To evaluate the effectiveness of various promotions, brand managers are interested in analysing data for the products represented, the promotional offers, and the locations where the promotions ran. Construct a star schema based on the information package diagram, and discuss how the brand manager or other analysts can use the model to evaluate the promotions.

All Time Periods	All Products	All Locations	All Promotions
Years	Category	Region	Type
Quarters	Sub-Category	District	Sub-type
Months	Brand	Store	Name
	Package Size		
	Measures/Facts: Units, Revenue, Cost, Margin (calculated)		

Data extraction and cleansing

The construction of a data warehouse begins with careful considerations on architecture and data model issues, and with their sizing components. It is essential that a correct architecture is firmly in place, supporting the activities of a data warehouse. Having solved the architecture issue and built the data model, the developers of the data warehouse can decide what data they want to access, in which form, and how it will flow through an organisation. This phase of a data warehouse project will actually fill the warehouse with goods (data). This is where data is extracted from its current environment and transformed into the user-friendly data model managed by the data warehouse. Remember, this is a phase that is all about quality. A data warehouse is only as good as the data it manages.

Extraction specifications

The data extraction part of a data warehouse is a traditional design process. There is an obvious data flow, with inputs being operational systems and output being the data warehouse. However, the key to the extraction process is how to cleanse the data and transform it into usable information that the user can access and make into business intelligence.

Thus, techniques such as data flow diagrams may be beneficial for defining extraction specifications for the development. An important input for such a specification may be the useful reports that you collected during user interviews. In these kinds of reports, intended users often tell you what they want and what they do not, and then you can act accordingly.

Loading data

Data needs to be processed for extraction and loading. An SQL select statement, shown below, is normally used in the process:

Select Target Column List

from Source Table List

where Join & Filter List

group by

or order by Sort & Aggregate List

Multiple passes of data

Some complex extractions need to pull data from multiple systems and merge the resultant data while performing calculations and transformations for placement into a data warehouse. For example, the sales analysis example mentioned in the star schema modelling section might be such a process. We may obtain budget sales information from a budgetary system, which is different from the order entry system from which we get actual sales data, which in turn is different from the forecast management system from which we get forecast sales data. In this scenario, we would need to access three separate systems to fill one row within the Sales Analysis measure table.

Staging area

Creating and defining a staging area can help the cleansing process. This is a simple concept that allows the developer to maximise up-time of a data warehouse while extracting and cleansing the data.

A staging area, which is simply a temporary work area, can be used to manage transactions that will be further processed to develop data warehouse transactions.

Checkpoint restart logic

The concept of checkpoint restart has been around for many years. It originated in batch processing on mainframe computers. This type of logic states that if a long running process fails prior to completion, then restart the process at the point of failure rather than from the beginning. Similar logic should be implemented in the extraction and cleansing process. Within the staging area, define the necessary structures to monitor the activities of transformation procedures. Each of these programming units has an input variable that determines where

in the process it should begin. Thus, if a failure occurs within the seventh procedure of an extraction process that has 10 steps, assuming the right rollback logic is in place, it would only require that the last four steps (7 through to 10) be conducted.

Data loading

After data has been extracted, it is ready to be loaded into a data warehouse. In the data loading process, cleansed and transformed data that now complies with the warehouse standards is moved into the appropriate data warehouse entities. Data may be summarised and reformatted as part of this process, depending on the extraction and cleansing specifications and the performance requirements of the data warehouse. After the data has been loaded, data inventory information is updated within the metadata repository to reflect the activity that has just been completed.

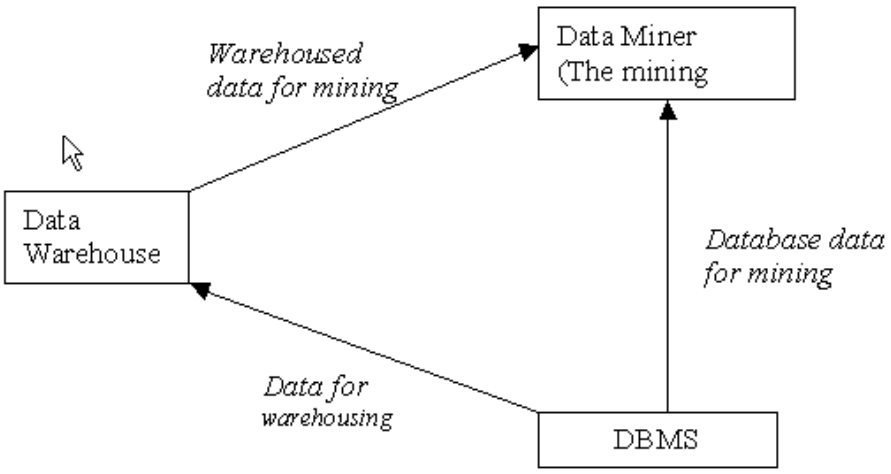
Review question 5

- How can a staging area help the cleansing process in developing a data warehousing system?
- Why is checkpoint restart logic useful? How can it be implemented for the data extraction and cleansing process?

Data warehousing and data mining

Data warehousing has been the subject of discussion so far. A data warehouse assembles data from heterogeneous databases so that users need only query a single system. The response to a user's query depends on the contents of the data warehouse. In general, the warehouse system will answer the query as it is and will not attempt to extract further/implicit information from the data.

While a data warehousing system formats data and organises data to support management functions, data mining attempts to extract useful information as well as predicting trends and patterns from the data. Note that a data warehouse is not exclusive for data mining; data mining can be carried out in traditional databases as well. However, because a data warehouse contains quality data, it is highly desirable to have data mining functions incorporated in the data warehouse system. The relationship between warehousing, mining and database is illustrated below:



In general, a data warehouse comes up with query optimisation and access techniques to retrieve an answer to a query – the answer is explicitly in the warehouse. Some data warehouse systems have built-in decision-support capabilities. They do carry out some of the data mining functions, like predictions. For example, consider a query like "How many BMWs were sold in London in 2010". The answer can clearly be in the data warehouse. However, for a question like "How many BMWs do you think will be sold in London in 2020", the answer may not explicitly be in the data warehouse. Using certain data mining techniques, the selling patterns of BMWs in London can be discovered, and then the question can be answered.

Essentially, a data warehouse organises data effectively so that the data can be mined. As shown in in the diagram above, however, a good DBMS that manages data effectively could also be used as a mining source. Furthermore, data may not be current in a warehouse (it is mainly historical). If one needs up-to-date information, then one could mine the database, which also has transaction processing features. Mining data that keeps changing is often a challenge.

General introduction to data mining

Data mining concepts

Data mining is a process of extracting previously unknown, valid and actionable information from large sets of data and then using the information to make crucial business decisions.

The key words in the above definition are unknown, valid and actionable. They

help to explain the fundamental differences between data mining and the traditional approaches to data analysis, such as query and reporting and online analytical processing (OLAP). In essence, data mining is distinguished by the fact that it is aimed at discovery of information, without a previously formulated hypothesis.

First, the information discovered must have been previously unknown. Although this sounds obvious, the real issue here is that it must be unlikely that the information could have been hypothesised in advance; that is, the data miner is looking for something that is not intuitive or, perhaps, even counterintuitive. The further away the information is from being obvious, potentially the more value it has. A classic example here is the anecdotal story of the beer and nappies. Apparently a large chain of retail stores used data mining to analyse customer purchasing patterns and discovered that there was a strong association between the sales of nappies and beer, particularly on Friday evenings. It appeared that male shoppers who were out stocking up on baby requisites for the weekend decided to include some of their own requisites at the same time. If true, this shopping pattern is so counterintuitive that the chain's competitors probably do not know about it, and the management could profitably explore it.

Second, the new information must be valid. This element of the definition relates to the problem of over optimism in data mining; that is, if data miners look hard enough in a large collection of data, they are bound to find something of interest sooner or later. For example, the potential number of associations between items in customers' shopping baskets rises exponentially with the number of items. Some supermarkets have in stock up to 300,000 items at all times, so the chances of getting spurious associations are quite high. The possibility of spurious results applies to all data mining and highlights the constant need for post-mining validation and sanity checking.

Third, and most critically, the new information must be actionable. That is, it must be possible to translate it into some business advantage. In the case of the retail store manager, clearly he could leverage the results of the analysis by placing the beer and nappies closer together in the store or by ensuring that two items were not discounted at the same time. In many cases, however, the actionable criterion is not so simple. For example, mining of historical data may indicate a potential opportunity that a competitor has already seized. Equally, exploiting the apparent opportunity may require use of data that is not available or not legally usable.

Benefits of data mining

Various applications may need data mining, but many of the problems have existed for years. Furthermore, data has been around for centuries. Why is it that we are talking about data mining now?

The answer to this is that we are using new tools and techniques to solve problems in a new way. We have large quantities of data computerised. The data could be in files, Relational databases, multimedia databases, and even on the World Wide Web. We have very sophisticated statistical analysis packages. Tools have been developed for machine learning. Parallel computing technology is maturing for improving performance. Visualisation techniques improve the understanding of the data. Decision support tools are also getting mature. Here are a few areas in which data mining is being used for strategic benefits:

- **Direct marketing:** The ability to predict who is most likely to be interested in what products can save companies immense amounts in marketing expenditures. Direct mail marketers employ various data mining techniques to reduce expenditures; reaching fewer, better qualified potential customers can be much more cost effective than mailing to your entire mailing list.
- **Trend analysis:** Understanding trends in the marketplace is a strategic advantage, because it helps reduce costs and timeliness to market. Financial institutions desire a quick way to recognise changes in customer deposit and withdraw patterns. Retailers want to know what product people are likely to buy with others (market basket analysis). Pharmaceuticals ask why someone buys their product over another. Researchers want to understand patterns in natural processes.
- **Fraud detection:** Data mining techniques can help discover which insurance claims, cellular phone calls or credit card purchases are likely to be fraudulent. Most credit card issuers use data mining software to model credit fraud. Citibank, the IRS, MasterCard and Visa are a few of the companies who have been mentioned as users of such data mining technology. Banks are among the earliest adopters of data mining. Major telecommunications companies have an effort underway to model and understand cellular fraud.
- **Forecasting in financial markets:** Data mining techniques are extensively used to help model financial markets. The idea is simple: if some trends can be discovered from historical financial data, then it is possible to predict what may happen in similar circumstances in the future. Enormous financial gains may be generated this way.
- **Mining online:** Web sites today find themselves competing for customer loyalty. It costs little for customer to switch to competitors. The electronic commerce landscape is evolving into a fast, competitive marketplace where millions of online transactions are being generated from log files and registration forms every hour of every day, and online shoppers browse by electronic retailing sites with their finger poised on their mouse, ready to buy or click on should they not find what they are looking for - that is, should the content, wording, incentive, promotion, product or service of a Web site not meet their preferences. In such a hyper-competitive market-

place, the strategic use of customer information is critical to survival. As such, data mining has become a mainstay in doing business in fast-moving crowd markets. For example, Amazon, an electronics retailer, is beginning to want to know how to position the right products online and manage its inventory in the back-end more effectively.

Comparing data mining with other techniques

Query tools vs. data mining tools

End-users are often confused about the differences between query tools, which allow end-users to ask questions of a database management system, and data mining tools. Query tools do allow users to find out new and interesting facts from the data they have stored in a database. Perhaps the best way to differentiate these tools is to use an example.

With a query tool, a user can ask a question like: What is the number of white shirts sold in the north versus the south? This type of question, or query, is aimed at comparing the sales volumes of white shirts in the north and south. By asking this question, the user probably knows that sales volumes are affected by regional market dynamics. In other words, the end-user is making an assumption.

A data mining process tackles the broader, underlying goal of a user. Instead of assuming the link between regional locations and sales volumes, the data mining process might try to determine the most significant factors involved in high, medium and low sales volumes. In this type of study, the most important influences of high, medium and low sales volumes are not known. A user is asking a data mining tool to discover the most influential factors that affect sales volumes for them. A data mining tool does not require any assumptions; it tries to discover relationships and hidden patterns that may not always be obvious.

Many query vendors are now offering data mining components with their software. In future, data mining will likely be an option for all query tools. Data mining discovers patterns that direct end-users toward the right questions to ask with traditional queries.

OLAP tools vs. data mining tools

Let's review the concept of online analytical processing (OLAP) first. OLAP is a descendant of query generation packages, which are in turn descendants of mainframe batch report programs. They, like their ancestors, are designed to answer top-down queries from the data or draw what-if scenarios for business analysts. During the last decade, OLAP tools have grown popular as the primary

methods of accessing database, data marts and data warehouses. OLAP tools are designed to get data analysts out of the custom report-writing business and into the 'cube construction' business. OLAP tools provide multidimensional data analysis – that is, they allow data to be broken down and summarised by product line and marketing region, for example.

OLAP deals with the facts or dimensions typically containing transaction data relating to a firm's products, locations and times. Each dimension can also contain some hierarchy. For example, the time dimension may drill down from year, to quarter, to month, and even to weeks and days. A geographical dimension may drill up from city, to state, to region, to country and so on. The data in these dimensions, called measures, is generally aggregated (for example, total or average sales in pounds or units).

The methodology of data mining involves the extraction of hidden predictive information from large databases. However, with such a broad definition as this, an OLAP product could be said to qualify as a data mining tool. That is where the technology comes in, because for true knowledge discovery to take place, a data mining tool should arrive at this hidden information automatically.

Still another difference between OLAP and data mining is how the two operate on the data. Similar to the direction of statistics, OLAP is a top-down approach to data analysis. OLAP tools are powerful and fast tools for reporting on data, in contrast to data mining tools that focus on finding patterns in data. For example, OLAP involves the summation of multiple databases into highly complex tables; OLAP tools deal with aggregates and are basically concerned with addition and summation of numeric values, such as total sales in pounds. Manual OLAP may be based on need-to-know facts, such as regional sales reports stratified by type of businesses, while automatic data mining is based on the need to discover what factors are influencing these sales.

OLAP tools are not data mining tools since the query originates with the user. They have tremendous capabilities for performing sophisticated user-driven queries, but they are limited in their capability to discover hidden trends and patterns in database. Statistical tools can provide excellent features for describing and visualising large chunks of data, as well as performing verification-driven data analysis. Autonomous data mining tools, however, based on Artificial Intelligence (AI) technologies, are the only tools designed to automate the process of knowledge discovery.

Data mining is data-driven or discovery-driven analysis and requires no assumptions. Rather, it identifies facts or conclusions based on patterns discovered. OLAP and statistics provide query-driven, user-driven or verification-driven analysis. For example, OLAP may tell a bookseller about the total number of books it sold in a region during a quarter. Statistics can provide another dimension about these sales. Data mining, on the other hand, can tell you the patterns of these sales, i.e. factors influencing the sales.

Website analysis tools vs. data mining tools

Every time you visit a Web site, the Web server enters a valuable record of that transaction in a log file. Every time you visit an electronic commerce site, a cookie is issued to you for tracking what your interests are and what products or services you are purchasing. Every time you complete a form on a site, that information is written to a file. Although these server log files and form-generated databases are rich in information, the data is itself usually abbreviated and cryptic in plain text format with comma delimiters, making it difficult and time-consuming to mine. The volume of information is also overwhelming: a one-megabyte log file typically contains 4,000 to 5,000 page requests. Web site analysis tools typically import the log file data into a built-in database, which in turn transforms the data into aggregate reports or graphs.

This information can be fine-tuned to meet the needs of different individuals. For example, a Web administrator may want to know about the clicks leading to documents and images, files, scripts and applets. A designer will want to know how visitors navigate the site and whether there are paths or points from which many visitors jump to another site. The marketing team will want to know the effectiveness of certain promotions. Advertisers and partners may be interested in the number of click-throughs your site has generated to their sites. Most Web site analysis tools provide answers to such questions as:

- What are the most common paths to the most important pages on your site?
- What keywords bring the most traffic to your site from search engines?
- How many pages do visitors typically view on your site?
- How many visitors are you getting from different parts of the world?
- How much time do visitors spend on your site?
- How many new users visit your site every month?

However, like statistical and OLAP tools, Web analysis tools are verification-driven. They emphasise aggregate counts and spatial views of website traffic over time, and are not easily able to discover hidden patterns, which could provide you with information like, what the visitors are really looking for. The current Web site analysis tools are very good at innovative data reporting via tables, charts and graphs.

A data mining tool does not replace a Web analysis tool, but it does give the Web administrator a lot of additional opportunities for answering some of the marketing and business questions. For example, imagine trying to formulate answers to questions such as:

- What is an optional segmentation of my Web site visitors?
- Who is likely to purchase my new online products and services?

- What are the most important trends in my site visitors' behaviour?
- What are the characteristics or features of my most loyal online clients?

Theoretically, these questions could be answered with a Web analysis tool. For example, a Web administrator could try to define criteria for a customer profile and query the data to see whether they work or not. In a process of trial and error, a marketer could gradually develop enough intuitions about the distinguishing features of its predominant Web site customers, such as their gender, age, location, income levels, etc. However, in a dynamic environment such as the Web, this type of analysis is very time-consuming and subject to bias and error.

On the other hand, a data mining tool (such as a decision tree generator) that incorporates machine-learning technology could find a better answer automatically, in a much shorter time – typically within minutes. More importantly, this type of autonomous segmentation is unbiased and driven by data, not the analyst's intuition. For example, using a data mining tool, a log file can be segmented into statistically significant clusters very quickly.

Data mining tasks

The most common types of data mining tasks, classified based on the kind of knowledge they are looking for, are listed as follows:

- **Classification:** Data records are grouped into some meaningful subclasses. For example, suppose a car sales company has some information that all the people in its list who live in City X own cars worth more than 20K. They can then assume that even those who are not on their list, but live in City X, can afford to own cars costing more than 20K. This way, the company classifies the people living in City X.
- **Sequence detection:** By observing patterns in the data, sequences are determined. Here is an example: after John goes to the bank, he generally goes to the grocery store.
- **Data dependency analysis:** Potentially interesting dependencies, relationships or associations between data items are detected. For example, if people buy X, they tend to buy Y as well. We say there is an association between X and Y.
- **Deviation analysis:** For example, John went to the bank on Saturday, but he did not go to the grocery store after that. Instead, he went to a football game. With this task, anomalous instances and discrepancies are found.

Techniques for data mining

Data mining is an integration of multiple technologies. These include data management such as database management, data warehousing, statistics, machine learning and decision support, and other technologies such as visualisation and parallel computing. Many of these technologies have existed for many decades. The ability to manage and organise data effectively has played a major role in making data mining a reality.

Database management researchers are taking advantages of work on deductive and intelligent query processing for data mining. One of the areas of interest is to extend query processing techniques to facilitate data mining. Data warehousing is another key data management technology for integrating the various data sources and organising the data so that it can be effectively mined.

Researchers in statistical analysis are integrating their techniques with those of machine learning to develop more sophisticated statistical techniques for data mining. Various statistical analysis packages are now being marketed as data mining tools. There is some dispute over this. Nevertheless, statistics is a major area contributing to data mining.

Machine learning has been around for a while. The idea here is for the machine to learn various rules from the patterns observed and then apply these rules to solve new problems. While the principles used in machine learning and data mining are similar, data mining usually considers large quantities of data to mine. Therefore, integration of database management and machine learning techniques are needed for data mining.

Researchers from the computing visualisation field are approaching the area from another perspective. One of their focuses is to use visualisation techniques to aid the mining process. In other words, interactive data mining is a goal of the visualisation community.

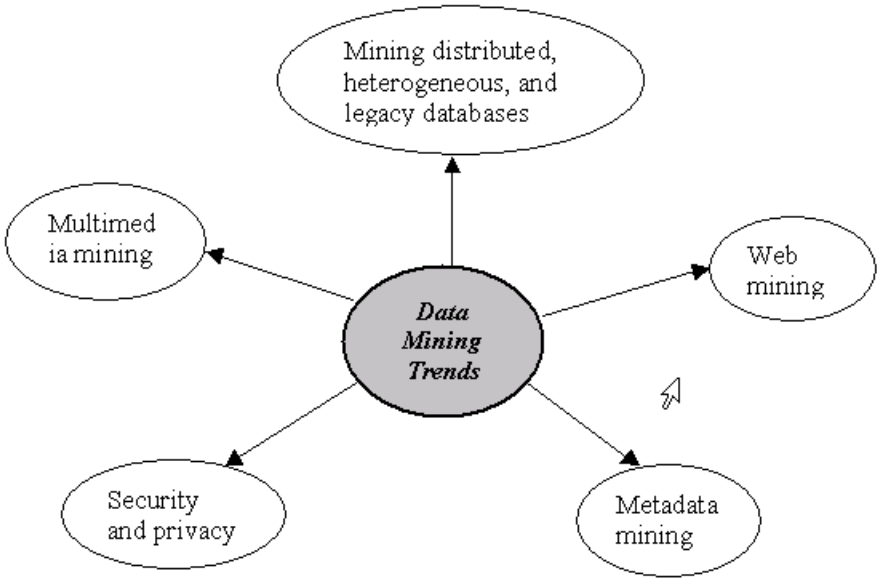
Decision support systems are a collection of tools and processes to help managers make decisions and guide them in management - for example, tools for scheduling meetings and organising events.

Finally, researchers in high-performance computing are also working on developing appropriate algorithms in order to make large-scale data mining more efficient and feasible. There is also interaction with the hardware community so that appropriate architectures can be developed for high-performance data mining.

Data mining directions and trends

While significant progresses have been made, there are still many challenges. For example, due to the large volumes of data, how can the algorithms determine which technique to select and what type of data mining to do? Furthermore, the

data may be incomplete and/or inaccurate. At times, there may be redundant information, and at times there may not be sufficient information. It is also desirable to have data mining tools that can switch to multiple techniques and support multiple outcomes. Some of the current trends in data mining are illustrated below:



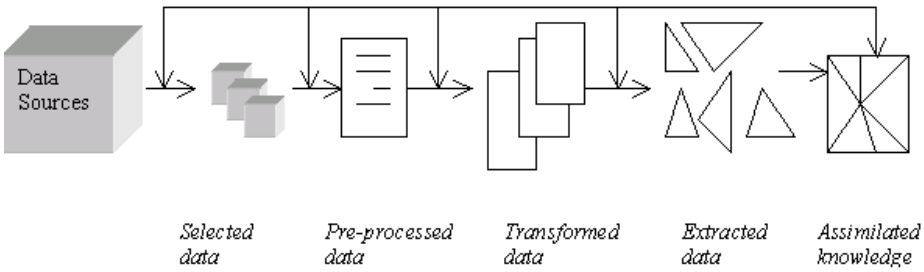
Review question 6

What is data mining? How is it used in the business world?

Data mining process

The process overview

In general, when people talk about data mining, they focus primarily on the actual mining and discovery aspects. The idea sounds intuitive and attractive. However, mining data is only one step in the overall process. The diagram below illustrates the process as a multistep, iterative process:



The business objectives drive the entire data mining process. They are the basis on which the initial project is established and the measuring stick by which the final results will be judged, and they should constantly guide the team throughout the process. Also, the process is highly iterative, with possibly many loop-backs over one or more steps. In addition, the process is far from autonomous. In spite of recent advances in technology, the whole data mining process remains very much a labour-intensive exercise.

However, not all steps are of equal weight in terms of typical time and effort spent. 60% of the time goes into preparing the data for mining, highlighting the critical dependency on clean, relevant data. The actual mining step typically constitutes about 10% of the overall effort.

The process in detail

Business objectives determination

This step in the data mining process has a lot in common with the initial step of any significant project undertaking. The minimum requirements are a perceived business problem or opportunity and some level of executive sponsorship. The first requirement ensures that there is a real, critical business issue that is worth solving, and the second guarantees that there is the political will to do something about it when the project delivers a proposed solution.

Frequently, you hear people saying: “Here is the data, please mine it.” But how do you know whether a data mining solution is really needed? The only way to find out is to properly define the business objectives. Ill-defined projects are not likely to succeed or result in added value. Developing an understanding and careful definition of the business needs is not a straightforward task in general. It requires the collaboration of the business analyst with domain knowledge and the data analyst, who can begin to translate the objectives into a data mining application.

This step in the process is also the time at which to start setting expectations. Nothing kills an otherwise successful project as quickly as overstated expectations of what could be delivered. Managing expectations will help to avoid any misunderstanding that may arise as the process evolves, and especially as the final results begin to emerge.

Data preparation

This is the most resource-consuming step in the process, typically requiring up to 60% of the effort of the entire project. The step comprises three phases:

- **Data selection:** Identification and extraction of data.
- **Data pre-processing:** Data sampling and quality testing.
- **Data transformation:** Data conversion into an analytical model.

Data selection

The goal of data selection is to identify the available data sources and extract the data that is needed for preliminary analysis in preparation for further mining. For example, if you want to find out who will respond to a direct marketing campaign, you need data (information) about customers who have previously responded to mailers. If you have their name and address, you should realise that this type of data is unique to a customer, and therefore, not the best data to be selected for mining. Information like city and area provides descriptive information, but demographic information is more valuable: items like a customer's age, general income level, types of interests and household type.

Along with each of the selected variables, associated semantic information (metadata) is needed to understand what each of the variables means. The metadata must include not only solid business definitions of the data but also clear descriptions of data types, potential values, original source system, data formats and other characteristics. There are two major types of variables:

- Categorical: The possible values are finite and differ in kind. For example, marital status (single, married, divorced, unknown), gender (male, female), customer credit rating (good, regular, poor).
- Quantitative: There is measurable difference here between the possible values. There are two subtypes: continuous (values are real numbers) and discrete (values are integrates). Examples of continuous variables are income, average number of purchases and revenue. Examples of discrete variables are number of employees and time of year (month, season, quarter).

The variables selected for data mining are called active variables, in the sense that they are actively used to distinguish segments, make predictions or perform some other data mining operations.

When selecting data, another important consideration is the expected period of validity of the data. That is, the extent to which ongoing changes in external circumstances may limit the effectiveness of the mining. For example, because a percentage of customers will change their jobs every year, any analysis where job type is a factor has to be re-examined periodically.

At this stage, the data analyst has already begun to focus on the data mining algorithms that will best match the business application. This is an important aspect to keep in mind as the other phases of the data preparation step evolve, because it will guide the development of the analytical model and the fine-tuning of the data input.

Data pre-processing

The aim of data pre-processing is to ensure the quality of the selected data. Clean and well-understood data is a clear prerequisite for successful data mining, just as it is with other quantitative analysis. In addition, by getting better acquainted with the data at hand, you are more likely to know where to look for the real knowledge during the mining stage.

Without a doubt, data pre-processing is the most problematic phase in the data preparation step, principally because most operational data is never meant to be for data mining purposes. Poor data quality and poor data integrity are major issues in almost all data mining projects.

Normally, the data pre-processing phase begins with a general review of the structure of the data and some measuring of its quality. Such an approach usually involves a combination of statistical methods and data visualisation techniques. Representative sampling of the selected data is a useful technique, as large data volumes would otherwise make the review process very time-consuming.

For categorical variables, frequency distributions of the values are a useful way of better understanding the data content. Simple graphical tools such as histograms and pie charts can quickly plot the contribution made by each value for the categorical variable, and therefore help to identify distribution skews and invalid or missing values. One thing that must be noted is that the frequency distribution of any data should be considered based on a large enough representation sample. For example, if a set has 1 million males and 1 female, then it is not a valid study for females.

When dealing with quantitative variables, the data analyst is interested in such measures as maxim and minima, mean, mode (most frequently occurring value), median (midpoint value) and several statistical measures of central tendency; that is, the tendency for values to cluster around the mean. When combined, these measures offer a powerful way of determining the presence of invalid and skewed data. For example, maxim and minima quickly show up spurious data

values, and the various statistical distribution parameters give useful clues about the level of noise in data.

During data pre-processing, two of the most common issues are noisy data and missing values.

Noisy data

With noisy data, one or more variables have values that are significantly out of line with what is expected for those variables. The observations in which these noisy values occur are called outliers. Outliers can indicate good news or bad – good news in the sense that they represent precisely the opportunities that we are looking for; bad news in that they may well be no more than invalid data.

Different kinds of outliers must be treated in different ways. One kind of outlier may be the result of a human error. For example, a person's age is recorded as 650, or an income is negative. Clearly, these values have to be either corrected (if a valid value or reasonable substitution can be found) or dropped from the analysis. Another kind of outlier is created when changes in operational systems have not yet been reflected in the data mining environment. For example, new product codes introduced in operational systems show up initially as outliers. Clearly in this case, the only action required is to update the metadata.

Skewed distribution often indicates outliers. For example, a histogram may show that most of the people in the target group have low incomes and only a few are high earners. It may be that these outliers are good, in that they represent genuine high earners in this homogeneous group, or it may be that they result from poor data collection. For example, the group may consist mainly of retired people but, inadvertently, include a few working professionals.

In summary, what you do with outliers depends on their nature. You have to distinguish the good outlier from the bad and react appropriately.

Missing values

Missing values include values that are simply not present in the selected data, and/or those invalid values that we may have deleted during noise detection. Values may be missing because of human error; because the information was not available at the time of input; or because the data was selected across heterogeneous sources, thus creating mismatches. To deal with missing values, data analysts use different techniques, none of which is ideal.

One technique is simply to eliminate the observations that have missing values. This is easily done, but it has the obvious drawback of losing valuable information. Although this data loss may be less of a problem in situations where data volumes are large, it certainly will affect results in mining smaller volumes or where fraud or quality control is the objective. In these circumstances, we may well be throwing away the very observations for which we are looking. Indeed, the fact that the value is missing may be a clue to the source of the fraud or quality problem. If there is a large number of observations with missing values

for the same variable, it may be an option to drop the variable from the analysis. This again has serious consequences because, unknown to the analyst, the variable may have been a key contributor to the solution.

The decision to eliminate data is never an easy one, nor can the consequences be easily foreseen. Luckily, there are several ways around the problem of missing values. One approach is to replace the missing value with its most likely value. For quantitative variables, this most likely value could be the mean or mode. For categorical variables, this could be the mode or a newly created value for the variable, called UNKNOWN, for example. A more sophisticated approach for both quantitative and categorical variables is to use a predictive model to predict the most likely value for a variable, on the basis of the values of other variables in observation.

Despite this stockpile of weapons to combat the problem of missing data, you must remember that all this averaging and predicting comes at a price. The more guessing you have to do, the further away from the real data the database moves. Thus, in turn, it can quickly begin to affect the accuracy and validation of the mining results.

Data transformation

During data transformation, the pre-processed data is transformed to produce the analytical data model. The analytical data model is an informational data model, and it represents a consolidated, integrated and time-dependent restructuring of the data selected and pre-processed from various operational and external sources. This is a crucial phase, as the accuracy and validity of the final results depend vitally on how the data analyst decides to structure and present the input. For example, if a department store wants to analyse customer spending patterns, the analyst must decide whether the analysis is to be done at some overall level, at the department level, or at the level of individual purchased articles. Clearly, the shape of the analytical data model is critical to the types of problems that the subsequent data mining can solve.

After the model is built, the data is typically further refined to suit the input format requirements of the particular data mining algorithm to be used. The fine-tuning typically involves data recording and data format conversion and can be quite time-consuming. The techniques used can range from simple data format conversion to complex statistical data reduction tools. Simple data conversions can perform calculations such as a customer's age based on the variable of the date of birth in the operational database. It is quite common to derive new variables from original input data. For example, a data mining run to determine the suitability of existing customers for a new loan product might require to input the average account balance for the last 3-, 6- and 12-month periods.

Another popular type of transformation is data reduction. Although it is a

general term that involves many different approaches, the basic objective is to reduce the total number of variables for processing by combining several existing variables into one new variable. For example, if a marketing department wants to gauge how attractive prospects can be for a new, premium-level product, it can combine several variables that are correlated, such as income, level of education and home address, to derive a single variable that represents the attractiveness of the prospect. Reducing the number of input variables produces a smaller and more manageable set for further analysis. However, the approach has several drawbacks. It is not always easy to determine which variables can be combined, and combining variables may cause some loss of information.

Clearly, data remodelling and refining are not trivial tasks in many cases, which explains the amount of time and effort that is typically spent in the data transformation phase of the data preparation step.

Another technique, called discretisation, involves converting quantitative variables into categorical variables, by dividing the values of the input variables into buckets. For example, a continuous variable such as income could be discretised into a categorical variable such as income range. Incomes in the range of £0 to £15,000 could be assigned a value Low; those in the range of £15,001 to £30,000 could be assigned a value Medium, and so on.

Last, One-of-N is also a common transformation technique that is useful when the analyst needs to convert a categorical variable to a numeric representation, typically for input to a neural network. For example, a categorical variable - type of car - could be transformed into a quantitative variable with a length equal to the number of different possible values for the original variable, and having an agreed coding system.

Data mining

At last we come to the step in which the actual data mining takes place. The objective is clearly to apply the selected data mining algorithm(s) to the preprocessed data.

In reality, this step is almost inseparable from the next step (analysis of results) in this process. The two are closely inter-linked, and the analyst typically iterates around the two for some time during the mining process. In fact, this iteration often requires a step back in the process to the data preparation step. Two steps forward, one step back often describes the reality of this part of the data mining process.

What happens during the data mining step is dependent on the type of application that is under development. For example, in the case of a database segmentation, one or two runs of the algorithm may be sufficient to clear this step and move into analysis of results. However, if the analyst is developing a predictive model, there will be a cyclical process where the models are repeatedly trained and retrained on sample data before being tested against the

real database. Data mining developments typically involve the use of several algorithms, which will be discussed in a later part of the chapter.

Analysis of results

Needless to say, analysing the results of the mining run is one of the most important steps of the whole process. In addition, in spite of improvements in graphical visualisation aids, this step can only be done properly by a skilled data analyst working with a business analyst. The analysis of results is inseparable from the data mining step in that the two are linked in an interactive process.

The specific activities in this step depend very much on the kind of application that is being developed. For example, when performing a customer database segmentation, the data analyst and business analyst attempt to label each of the segments, to put some business interpretation on them. Each segment should be homogeneous enough to allow for this. However, if there are only a few segments with large concentrations of customer records, the segment cannot be sufficiently differentiated. In this case, changing the variables on which the segmentation is based improves the result. For example, removing the most common variables from the large segments gives a more granular segmentation on a rerun.

When predictive models are being developed, a key objective is to test their accuracy. This involves comparing the prediction measures against known actual results, and input sensitivity analyses (the relative importance attributed to each of the input variables). Failure to perform satisfactorily usually guides the team toward the unduly influential input, or sends it in search of new input variables. One common source of error when building a predictive model is the selection of overly predictive variables. In the worst case, the analyst may inadvertently select a variable that is recorded only when the event that he or she is trying to predict occurs. Take, for example, policy cancellation data as input to a predictive model for customer attrition. The model will perform with 100% accuracy, which should be a signal to the team to recheck the input.

Another difficulty in predictive modelling is that of over-training, where the model predicts well on the training data but performs poorly on the unseen test data. The problem is caused by over-exposure to the training data – the model learns the detailed patterns of that data but cannot generalise well when confronted with new observations from the test data set.

Developing association rules also poses special considerations. For example, many association rules discovered may be inactionable or will reflect no more than one-off instances. In some other cases, only the major rules, which are already well known and therefore not actionable, are discovered. Clearly, this is one area where careful tuning and iteration are needed to derive useful information.

Assimilation of knowledge

This step closes the loop, which was opened when we set the business objectives at the beginning of the process. The objective now is to put into action the commitments made in that opening step, according to the new, valid and actionable information from the previous process steps. There are two main challenges in this step: to present the new findings in a convincing, business-oriented way, and to formulate ways in which the new information can be best exploited.

Several technical issues need to be considered. At a minimum, the new information may manifest itself as new data mining applications or modifications to be integrated into existing technical infrastructure. Integration could involve the inclusion of new predictive models and association rules in existing application code, expert system shells or database procedures. In addition, operational and informational system databases may be enhanced with new data structures. In any event, the experiences during the data preparation step will doubtless put a focus on data integrity in upstream operational systems. This focus will create a demand for improved data quality and documentation in these systems, and improved manual procedures to prevent error or fraud.

Data mining algorithms

From application to algorithm

There exist a large number of different approaches to data mining, and they can be confusing initially. One reason for such confusions might be that inconsistent terminology is used among data mining practitioners themselves. The table below offers some examples of data mining applications, together with their supporting operations (models) and techniques (algorithms):

	Market Management	Risk Management	Fraud Management
Typical Applications	· *Target marketing* · *Customer relationship management* · *Market basket analysis* · *Cross selling* · *Market segmentation*	· *Forecasting* · *Customer retention* · *Quality control* · *Competitive analysis*	· *Fraud detection*
Models	***Predictive Modelling (Classification)***	***Segmentation (Clustering)***	***Link Analysis***	***Deviation Detection***
Techniques	· *Decision tree* · *Memory-based learning* · *Neural networks*	· *Geometric* · *Neural networks*	*Association rules discovery*	· *Visualisation* · *Statistics*

The applications listed in the table represent typical business areas where data mining is used today. Predictive modelling, database segmentation, link analysis and deviation detection are the four major operations or models for implementing any of the business applications. We deliberately do not show a fixed, one-to-one link between the applications and data mining model layers, to avoid suggestions that only certain models are appropriate for certain applications and vice versa. Nevertheless, certain well-established links between the applications and the corresponding operation models do exist. For example, target marketing strategies are always implemented by means of the database segmentation operation. In addition, the operations (models) are not mutually exclusive. For example, a common approach to customer retention is to segment the database first and then apply predictive modelling to the resultant, more homogeneous segments.

Popular data mining techniques

Decision trees

Decision trees (or a series of IF/THEN rules) as a commonly used machine learning algorithm are powerful and popular tools for classification and prediction. They normally work in supervised learning situations, where they attempt to find a test for splitting a database among the most desired categories, such as "Web site visitor will buy vs. will not buy". In both instances, these algorithms will try to identify important data clusters of features within a database. Normally, an attribute (feature/field) is tested at a node of a tree; the number of branches from that node is usually the number of possible values of that attribute (for example, for gender, it will be Male, Female or Unknown, so three branches for node gender). If the attribute is numeric, the node in a decision tree usually tests whether its value is less than a predetermined constant, giving a two-way split. Missing values in a data set are treated as an attribute value in their own right. Consideration is given to the fact that a missing value may be of some significance. An example of decision trees is shown below. It may be generated from past experience (data) and can be used to decide what to do according to weather conditions.

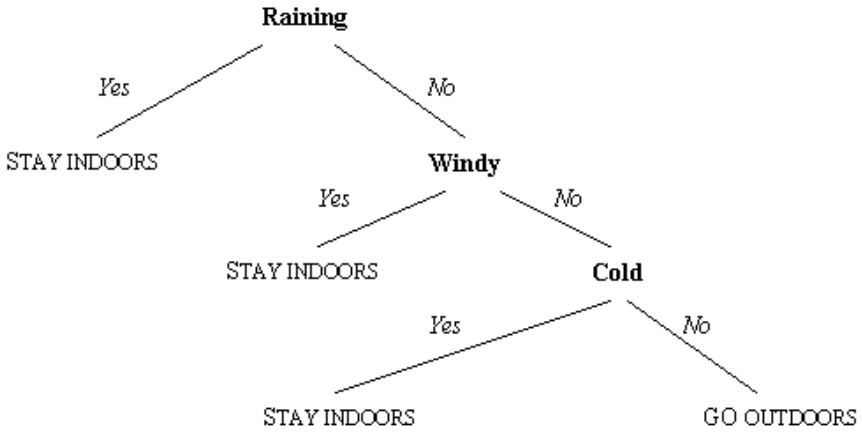
Data mining tools incorporating machine learning algorithms such as CART (classification and regression trees), CHAID (chi-squared automatic integration detection), ID3 (Interactive Dichotomizer) or C4.5 or C5.0 will segment a data set into statistically significant clusters of classes based on a desired output. Some of these tools generate 'decision trees' that provide a graphical breakdown of a data set, while others produce IF/THEN rules, which segment a data set into classes that can point out important ranges and features. Such a rule has two parts, a condition (IF) and a result (THEN), and is represented as a statement. For example:

IF customer_code is 03 AND number_of_purchases_made_this_year is 06 AND post_code is W1 THEN will purchase Product_X

Rule's probability: 88%. The rule exists in 13000 records. Significance level: Error probability $< 13\%$

A measure of information

There are two main types of decision trees: binary and multiple branches. A binary decision tree splits from a node in two directions, with each node representing a yes-or-no question like the tree below. Multiple-branched decision trees, on the other hand, can accommodate more complex questions with more than two answers. Also, a node in such a tree can represent an attribute with more than two possible values.

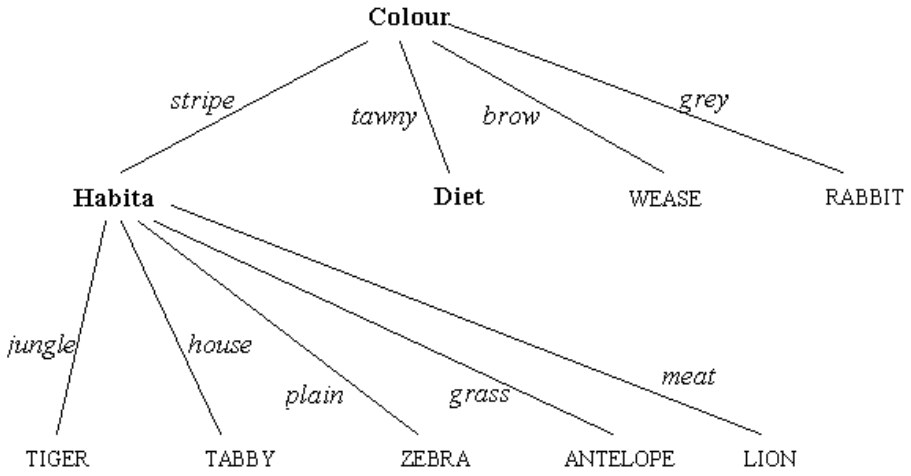


As mentioned before, there are a number of practical algorithms for building decision trees. ID3 is one of them - it can automatically build trees from given positive or negative instances. Each leaf of a decision tree asserts a positive or negative concept.

To classify a particular input, we start at the top and follow assertions down until we reach an answer. As an example, the following table lists the relationship between species of animals and their features such as diet, size, colour and habitat. Given a set of examples such as this, ID3 induces an optimal decision tree for classifying instances. As can be seen from the figure, not all of the features presented in the table are necessary for distinguishing classes. In this example, the size feature is not needed at all for classifying the animals. Similarly, once the brown or grey branches of the tree are taken, the remaining features can be ignored. It means that colour alone is sufficient to distinguish rabbits and weasels from the other animals.

Diet	Size	Colour	Habitat		Species
meat	large	striped	jungle		TIGER
meat	large	tawny	jungle		LION
meat	small	striped	house		TABBY
meat	small	brown	jungle		WEASEL
grass	large	striped	plains		ZEBRA
grass	small	grey	plains		RABBIT
grass	large	tawny	plains		ANTELOPE

The ID3 algorithm builds a decision tree in which any classification can be performed by checking the fewest features (that is why the tree is called optimal). It builds the tree by first ranking all the features in terms of their effectiveness, from an information-theoretic standpoint, in partitioning the set of target classes. It then makes this feature the root of the tree; each branch represents a partition of the set of classifications. The algorithm then recurs on each branch with the remaining features and classes. When all branches lead to single classifications, the algorithm terminates.



Neural networks

Neural networks (NN) are another popular data mining technique. An NN is a system of software programs and data structures that approximates the operation of the brain. It usually involves a large number of processors (also called elements/neurons/nodes) operating in parallel, each with its own small sphere of knowledge and access to data in its local memory. Typically, an NN is initially 'trained' or fed with large amounts of data and rules about data relationships. NNs are basically computing memories where the operations are all about association and similarity. They can learn when sets of events go together, such as when one product is sold, another is likely to sell as well, based on patterns they have observed over time.

Supervised learning

This is basically how most neural networks learn: by example, in a supervised mode (the correct output is known and provided to the network). Supervised models, such as back propagation networks, are trained with pairs of examples: positive and negative. A given input pattern is matched with a desired output pattern. Training a supervised network is a process of providing a set of inputs and outputs, one at a time. The network trains by taking in each input pattern, producing an output pattern, and then comparing its output to the desired output. If the network output is different from the desired output, the network adjusts its internal connection strengths (weights) in order to reduce the difference between the actual output and the desired output. If, however, its output matches the desired output, then the network has learned the pattern and no correction is necessary. This process continues until the network gets the patterns of input/output correct or until an acceptable error rate is attained.

However, because the network may get one set of patterns correct and another wrong, the adjustments that it makes are continuous. For this reason, training of a network is an interactive process where input/output patterns are presented over and over again until the network 'gets' the patterns correctly.

A trained network has the ability to generalise on unseen data; that is, the ability to correctly assess samples that were not in the training data set. Once you train a network, the next step is to test it. Expose the network to samples it has not seen before and observe the network's output. A common methodology is to split the available data, training on a portion of the data and testing on the rest.

Preparing data

In general, it is easier for a NN to learn a set of distinct responses (e.g. yes vs. no) than a continuous valued response (e.g. sales price). A common way to deal with this problem is to 'discrete' an attribute. Rather than having a single input for each sale amount, you might break it down to several ranges. Here

is an example. Let's say a Web site sells software products that range in price from very low to very high. Here, the 1-of-N coding conversion is adopted:

£10-19	Very low	00001	Input 1
£20-50	Low	00010	Input 2
£51-100	Average	00100	Input 3
£101-199	High	01000	Input 4
£200-£999	Very high	10000	Input 5

Most of today's data mining tools are able to shift the data into these discrete ranges. You should make sure that you include all ranges of values for all the variables that the network is subject to encounter. In the Web site example, this means including the least and most expensive items, and the lowest and highest amounts of sales, session times, units sold, etc. As a rule, you should have several examples in the training set for each value of a categorical attribute and for a range of values for ordered discrete and continuous valued features.

As a summary of supervised NNs for data mining, the main tasks in using a NN tool are listed below:

- Identify the input variables – this is very important.
- Convert the variables into usable ranges – pick a tool which will do this.
- Decide on the format of the output – continuous or categorical?
- Decide on a set of training data samples and a training schedule.
- Test the model and apply it in practice.

Unsupervised learning - self-organising map (SOM)

SOM networks are another type of popular NN algorithm that incorporate with today's data mining tool. An SOM network resembles a collection of neurons, as in the human brain, each of which is connected to its neighbour. As an SOM is exposed to samples, it begins to create self-organising clusters, like cellophane spreading itself over chunks of data. Gradually, a network will organise clusters and can therefore be used for situations where no output or dependent variable is known. SOM has been used to discover associations for such purposes as market basket analysis in retailing.

The most significant feature of an SOM is that it involves unsupervised learning (using a training sample for which no output is known), and is commonly used

to discover relations in a data set. If you do not know what you are attempting to classify, or if you feel there may be more than one way to categorise the data, you may want to start with an SOM.

Activity 2 – SOM neural networks

http://websom.hut.fi/websom/ is a site describing SOM networks for text mining. It provides good demonstration of the system. Try to see how text documents are clustered by SOM.

Review question 7

- What is a decision tree? Use an example to intuitively explain how it can be used for data mining.
- What are the differences between supervised and unsupervised learning?

Discussion topics

In this chapter, we have covered the most important aspects of data warehousing and data mining. The following topics are open to discussion:

- Data warehouse systems are so powerful – they improve data quality, allow timely access, support for organisational change, improve productivity and reduce expense. Why do we still need operational systems?
- Discuss the relationships between database, data warehouse and data mining.
- Discuss why data mining is an iterative process.

Chapter 20. Database Administration and Tuning

Table of contents

  - Tuning disk I/O
  - Tuning memory
  - Tuning contention
  - Tools to assist performance tuning
    * Software monitors
    * Hardware monitors
  - Other performance tools
  - Concluding remarks on database tuning
- Discussion topics

Objectives

At the end of this chapter you should be able to:

- Describe the principle functions of a database administrator.
- Discuss how the role of the database administrator might be partitioned among a group of people in a larger organisation.
- Discuss the issues in tuning database systems, and describe examples of typical tools used in the process of database administration.

Introduction

In parallel with this chapter, you should read Chapter 9 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

The functions of the database administrator (DBA) impact on every aspect of database usage within an organisation, and so relate closely to most of the chapters of this module. Database administrators often use the SQL language for the setting up of users, table indexes and other data structures required to support database applications. They will often work closely with analysts and programmers during the design and implementation phases of database applications, to ensure that the decisions made during these processes are appropriate in the context of the organisation-wide use of the database system. The DBA will work with management and key users in developing security policy, and will be responsible for the effective implementation of that policy in respect of the use of the databases in the organisation. In the common situation where client-server systems or a fully distributed database system are being used, the DBA is likely to be the one who determines important aspects relating to how the processing and/or data are distributed within the system, including, for example, the frequency with which replicas are updated in a DDBMS. This activity will be undertaken in close consultation with users and developers within the organisation, so that the strategy is informed by a good understanding of the priorities of the organisation as a whole. Furthermore, a good understanding of

the potential of new database technology, such as Object databases, is important to the DBA role, as it enables the DBA to advise budget holders within the organisation of the potential benefits and pitfalls of new database technology.

Modern database management systems (DBMSs) are complex software systems. When considered within the context of an organisational environment, they are usually key to the success of the organisation, enabling personnel at all levels of the organisation to perform essential tasks. In order for the database systems within an organisation to remain well aligned to requirements and provide a high degree of availability and efficiency, the system needs to be administrated effectively. The importance of the tasks of database administration was highlighted from the early days of database systems, in the writings of James Martin and others. With the increased complexity and frequent need in present systems to enable multiple databases to combine to provide an efficient overall service, the importance of database administration cannot be over-stressed. In this chapter we shall examine the tasks involved in the administration of database systems. We shall start by examining each of the major aspects of the job of a database administrator (DBA), and go on to explore the way in which the tasks involve the DBA in working with people from all levels of an organisation. In larger organisations, the functions of the DBA are sufficiently numerous to warrant being split among a number of different individuals who will form the members of a DBA group. We shall look at the ways that the job of the DBA might be partitioned, and explore the issue of what level within an organisation the DBA functions should be placed.

One of the most important and technically demanding aspects of database administration is the performance tuning for the database. This involves ensuring that the database is running efficiently, and providing a reliable and responsive service to the users of the system. In the later part of the chapter, we will discuss the major aspects of database tuning, using examples drawn from current systems.

Functions of the DBA

Management of data activity

The term 'data activity' refers to the way in which data is accessed and used within an organisation. Within medium to large organisations, formal policies are required to define which users should be authorised to access which data, and what levels of access they should be given; for example, query, update, copy, etc. Processes also need to be defined to enable users to request access to data which they would not normally be able to use. This will involve the specification of who has responsibility for authorising users to have data access. This might typically involve the line manager of the user requesting access, who should be in a position to verify that access is genuinely required, plus the authorisation of the user identified as the owner of the data, who should be in a position to

be aware of any implications of widening the access to the data. From time to time, conflicts may arise during the processing of requests for access to data. For example, a user may request access to particularly sensitive sales or personnel data, which the data owner refuses. In these circumstances, the DBA may be involved as one of those responsible for resolving the conflict in the best interests of the organisation as a whole.

Much of the DBA's activity will be focused on supporting the production environment of the organisation, where the operational programs and data are employed in the daily business of the organisation. However, in medium to large organisations, there will be significant activity to support developers in the processing of writing and/or testing new software.

Data activity policies formulated to address these issues need to be clearly documented and disseminated throughout the organisation. The policies might include agreements about the availability of data, specific times at which databases may be taken offline for maintenance, and the schedules for migrating new programs and data into the production environment.

A further aspect of data activity policy is the determination of procedures for the backup and recovery of data. The DBA will require a detailed knowledge of the options provided by the DBMSs being used regarding the backup of data, the logging of transactions, and the recovery procedures in light of a range of different types of failure.

Management of database structure

Another fundamental area of activity for the DBA concerns the definition of the structure of the databases within the organisation. The extent to which this is required will vary greatly depending on how mature the organisation is in terms of existing database technology. If the organisation is in the process of acquiring and setting up a database or databases for the first time, many fundamental decisions need to be taken regarding how best this should be done. Most organisations have progressed well beyond these fundamental issues today; however, even in such organisations, the effectiveness of those initial decisions needs to be monitored to ensure that the computer-based information within the organisation provides a good fit for the needs of its users. Among the key decisions to be taken during the process of establishing database systems within an organisation are the following:

- Which type of database systems best suit the organisation? There is no doubt that Relational database systems dominate the current marketplace, but for some organisations, particularly those specialising in engineering or design, there is certainly a case for examining the potential of Object-based systems, either instead of or running alongside a Relational system.
- How many databases are required? In the early days of database systems,

most organisations used one single database system, and the data on that system could be considered to provide the data model for the enterprise as a whole. Today, it is much more often the case that a number of database systems are employed, sometimes from different vendors, and sometimes with different underlying data models. This usually implies the need to transfer at least some of the data between these different systems.

- Is it necessary to establish a number of different database environments to support different types of activity within the organisation? This might, for example, involve the creation of development and production environments mentioned above, or may require a different organisation of work, such as, for example in a chemical company, the setting up of one environment to support online transaction processing, and another environment to enable the research and development of models of chemical and manufacturing processes.
- The interfaces between different database systems must be defined to allow transfer of data between them. Furthermore, interfaces between each database system and any other important software components, such as groupware and/or office information systems, need to be established.

The above decisions can be seen as arising at the strategic level of database administration, and will be considered in conjunction with senior members of the company and/or department in which the DBA is based. Nearer to the operational level, there is a range of important decisions concerning the structure of individual database systems within the organisation. Some of these are described below.

Tables and tablespaces

A 'tablespace' is a unit of disk space which contains a number of database tables. Usually each tablespace is allocated to data of a particular type; for example, there may be a tablespace established to contain user data, and another one to contain temporary data, i.e. data that is only in existence for a short time and is usually required in order to enable specific processes such as data sorting to take place. A further example might be in a university database, where separate tablespaces might be established respectively to support student, teaching staff and administrative users.

As an aside, the pattern of usage of tables in the student tablespace of a university database is very different to that typically seen in a commercial system. Very generally speaking, in a commercial system, data activity in a production environment might be characterised as comprising the following:

- Access to a relatively small number of fairly large tables.
- Regular executions of a predefined set of transactions.

- Several transactions may regularly scan large volumes of data.

In contrast, data activity in a student tablespace during a database class might be characterised as comprising:

- Many different tables, each of which contains only a few rows.
- Irregular and different types of transactions.
- Very small numbers of records processed by each transaction.

Tablespaces are an extremely important unit of access in database administration. In many systems they are a unit of recovery, and, for example, it may be necessary to take the whole tablespace offline in order to carry out recovery or data reorganisation operations to data in that tablespace.

A further important consideration is the way in which tables, indexes and tablespaces are allocated to physical storage media such as disks. This will be discussed further later in this chapter, in the section on database tuning. It is important here to point out that the DBA requires a good understanding of the volume of query and update transactions to be made on the tables in a tablespace, so that the allocation of physical storage space can be made in such a way that no physical device such as a disk becomes a major bottleneck in limiting the throughput of transactions.

Typically, database systems provide a number of parameters that can be used to specify the size of tablespaces and tables. These often include the following:

- Initial extents: The amount of disk space initially made available to the structure.
- Next extents: The value by which the amount of storage space available to the structure will be increased when it runs out of space.
- Max extents: The maximum amount of space to be made available to the structure, beyond which further growth can only be enabled by further intervention by the DBA.

Note: An extent is a term referring to a contiguous area of disk space. For any specific DBMS, it will consist of a number of data blocks which will be stored together in the same extent.

Data fragmentation

Over a period of time during which a database table is being used, it is likely to experience a significant number of inserts, updates and deletions of data. Because it is not always possible to fit new data into the gaps left by previously removed data, the net effect of these changes is that the storage area used to contain the table is left rather fragmented, containing a number of small areas which are hard to insert new data into them. This phenomenon is called ‘data

fragmentation'. The effect of data fragmentation in the long run is to slow down the performance of transactions accessing data in the fragmented table, as it takes longer to locate data in the fragmented storage space. Some database systems provide utilities (small programs) that can be used to remove these pockets of unusable space, consolidating the table once more into one contiguous storage structure — this is known as defragmentation. In other DBMSs, where no defragmentation utility is available, it may be necessary to export the table to an operating system file, and re-import it into the database, so that the pockets of unusable free space are removed.

Review question 1

- Describe the difference between production and development environments.
- Make a list of the range of different types of failure that the DBA needs to plan for in determining adequate backup and recovery strategies.
- Describe typical parameters that can be used to control the growth of tables and tablespaces.
- What is gained when a table is defragmented?

Designing for the future

An important overall principle to be applied when trying to estimate the requirements for storage and performance tuning is to design for the future. It will be part of the DBA's role to collect information from those responsible for the introduction of new database applications, about the volumes of data and processing involved. Producing such estimates can be extremely difficult, but even when these are correct, it is a mistake to plan on the basis of these figures alone. It is more realistic to base estimates of the volume of data and processing required on what they might be expected to have reached in a year's time, and to provide sufficient storage and processing capacity on that basis. This avoids being caught by surprise, should the volume of activity within the application grow much faster than was initially anticipated.

Information dissemination

Communication is an essential aspect of the role of the DBA. When new releases of DBMSs are introduced within the organisation, or new applications or upgrades come into use, it is important that the DBA is sensitive to the needs of the user population as a whole, and produces usable documentation to describe the changes that are taking place, including the possible side effects on users and developers. An equally important role in communicating information relates to the development and dissemination of information about programming and testing standards that may be required within an organisation. This may include

specifications of who is able to access which data structures, and standards for the use of query and update transactions throughout the organisation.

Supporting application developers

The DBA plays an important role in assisting those involved with the development and/or acquisition of software for the organisation. As well as being the gatekeeper of database resources, and monitoring future requirements for database processing (as exemplified in activity 1 further on), the DBA can provide ongoing advice to analysts and programmers about the best use of database resources. Typical issues that this may involve are the following:

- Information on the different types and instances of databases in the organisation, including the interfaces between them, any regular times when databases are unavailable, etc.
- Advice on the details of existing tables and tablespaces in the databases of the organisation.
- Advice on the use of indexes, different index types available and indexes currently set up.
- Details of security standards and procedures.
- Details of existing standards and procedures for the use of programming languages (including query languages).
- Details of migration procedures used for moving programs and data in and out of production.

Use of the data dictionary

The data dictionary is a key resource for database administration. It contains data about the tablespaces, tables, indexes, users and their privileges, database constraints and triggers, etc. Database administrators should develop a good knowledge of the most commonly used tables in the dictionary, and reference it regularly to retrieve information about the current state of the system. The way in which dictionaries are organised varies greatly between different DBMSs, and may change significantly with different releases of the same DBMS.

The Oracle data dictionary

The Oracle data dictionary is a set of tables that the Oracle DBMS uses to record information about the structure of the database. There is a set of core system tables, which are owned by the Oracle user present on all Oracle databases, the SYS user. SYS is rarely used, even by DBAs, for maintenance or enquiry work. Instead, another Oracle user with high-level system privileges is used.

The DBA does not usually use the SYSTEM user, which is also automatically defined when the database is created. This is because product-specific tables are installed in SYSTEM, and accidental modification or deletion of these tables can interfere with the proper functioning of some of the Oracle products.

Core system tables

The core data dictionary tables have short names, such as tab$, col$, ind$. These core system tables are rarely referenced directly, for the information is available in more easily usable forms in the data dictionary views defined when the database is created. To obtain a complete list of the data dictionary views, query the DICT view, as shown in the section on data dictionary views below.

Data dictionary views

The data dictionary views are based on the X$ and V$ tables. They make information available in a readable format. The names of these views are available by selecting from the DICT data dictionary view. Selecting all the rows from this view shows a complete list of the other accessible views.

SQL*DBA is an Oracle product from which many of the database administration tasks can be performed. However, SQL*Plus provides basic column formatting, whereas the SQL*DBA utility does not. Therefore, you use SQL*Plus for running queries on these views. If you are not sure which data dictionary view contains the information that you want, write a query on the DICT_COLUMNS view.

Most views used for day-to-day access begin with USER, ALL or DBA. The USER views show information on objects owned by the Oracle user running the query. The data dictionary views beginning with ALL show information on objects owned by the current Oracle user as well as objects to which the user has been given access. If you connect to a more privileged Oracle account such as SYS or SYSTEM, you can access the DBA data dictionary views. The DBA views are typically used only by the DBA. They show information for all the users of the database. The SELECT ANY TABLE system privilege enables other users to access these views. Querying the DBA_TABLES view shows the tables owned by all the Oracle user accounts on the database.

Application

Activity 1 - Capturing requirements for application migration

Imagine you are a database administrator working within a large organisation multinational. The Information Systems department is responsible for the in-house development and purchase of software used throughout the organisation. As part of your role, you require information about the planned introduction of software that will run in the production database environment. You are required

to develop a form which must be completed by systems analysts six weeks before their application is moved into production. Using a convenient word processing package, develop a form to capture the data that you think may be required in order for the DBA team to plan for any extra capacity that might be required for the new application.

Activity 2 - Examining dictionary views available from your user account

Query the DICT_COLUMNS table to examine the list of tables including the string USER visible from your own Oracle account. Remember to put USER in upper case in the query. The output should contain quite a lot of views. To avoid the information scrolling off the top of the screen, issue the command

Set pause on

prior to issuing the query. The information should then appear 24 lines at a time. You can view the next 24 lines by pressing the Enter key on your PC.

Activity 3 - Examining the details of available tables

Log into your Oracle account. Issue the following command to view the details of tables that you own:

SELECT * FROM USER_TABLES;

Examine the columns of the results that are returned.

To explore any further tables that are available to you, issue the following query:

SELECT * FROM ALL_TABLES;

To examine any of these tables further, use the DESCRIBE command, or issue further SELECT commands on the specific tables you find.

Control of redundancy

We have seen from the chapters on database design that a good overall database design principle is to store one fact in one place. This reduces the chances of data inconsistency, and avoids wasted space. There are, however, situations in which it can be appropriate to store redundant information. One major example of this we have seen in connection with the storage of replicas or copies of data at different sites in a truly distributed database system. Even within a non-distributed system, data may sometimes be duplicated, or derived data stored, in order to improve overall performance. An example of storing derived data to improve performance, might be where a value which normally could be calculated from base values in an SQL query is actually stored explicitly, avoiding the need for the calculation. For example, we might store net salary values, rather than relying on transactions to calculate the value by subtracting various deductions (for tax, national insurance, etc) from gross pay. This reduces

the processing load on the system, at the expense of the extra storage space required to hold the derived values. The issue of calculating derived values every time from the underlying base values is generally more of a significant problem than the provision of the extra storage space required. The DBA will play an important role in advising developers about the use of redundant and derived data, and will maintain control of these issues, as part of the overall responsibility for ensuring the database runs efficiently.

Configuration control

In a modern database environment, there are usually several products which are used to provide the range of functions required. These will typically include the database server, and various additional products such as a GUI forms-based interface, report writing tool, data graphing tool, software to assist the design of new applications, utilities for migrating programs and data in and out of production, etc. All of these different software components will be upgraded from time to time, and it is likely that some of them will come from different vendors. Many of the problems that occur in developing Information Systems arise when trying to enable two or more software components to communicate effectively. For this reason, it is essential that the DBA maintains a record of which versions of which software components have been and are currently running. In general, this area of work is known as configuration control. The information needed to be maintained to carry out successful configuration control is, as a minimum, the following:

- For currently running software components, a note of the name, exact release number, date put into production, vendor, and the details of any parameter values or activities that have been necessary in order to establish interfaces with other products, including the release numbers of those products and the dates during which these changes were effective.
- For software components that have been used previously: the name, exact release number, dates introduced and withdrawn from production, vendor, and as above, any details required of parameter values or other activities required to enable this component to communicate effectively with other products, including the release numbers of those products and the dates during which these values, etc, were in effect.

Even when all of the products have been purchased from the same vendor, proper configuration control is extremely important to ensure the correct running of the environment. Having this information available gives the option that, should a problem arise involving incompatibilities of software components, it may be possible to revert back to a previously stable configuration, maintaining a service to users while the incompatibility problem is resolved.

Security

Security is a major issue in database systems, and the DBA is the foremost person with responsibility for ensuring the day-to-day security of the stored data. This process begins with the DBA working in conjunction with managers, key users and owners of the organisations data, to establish appropriate security mechanisms and procedures. This will be a process of determining how to make the most appropriate use within the organisation of the security mechanisms provided by the DBMS and other software in use, plus a clear definition and documentation of supporting policies and processes to ensure that data and programs are properly protected. Typical issues that should be addressed include:

- Procedures for the allocation of passwords. Many database systems provide considerable flexibility in the use of passwords, enabling them to be set from the database level right down to the individual attribute level. Procedures need to be defined regarding how passwords are allocated, including any rules regarding the format of passwords; e.g. whether they should exceed a certain length, and how long they can be used for before they expire and need to be reset. The requirements for passwords may vary hugely, from a development environment in which a number of standard user accounts have been established requiring no password protection, right through to highly secure production situations such as records of bank account details, in which two passwords may be required to access a particularly sensitive attribute. In general, a very important consideration to keep in mind with all security mechanisms, is the set of procedures and practices that are used within the organisation for the use and communication of security information. For example, there is no point in having a very sophisticated software protection system to prevent people from discovering one another's passwords, if it is commonplace for people to write their passwords on their PCs or in other places in their work area, where they can be easily read by anybody.
- Procedures for the administration of database privileges, such as the granting and revocation of access to tables, query and update transactions.
- The use of encryption techniques for encoding data while it is being transmitted over networks, including any intranet and the Internet.
- As discussed earlier, the establishment of procedures for transaction logging and recovery from a range of different failures.

Summary of DBA functions

Documentation

We have seen that the job of the database administrator impacts on many aspects of an organisation's work. Depending on the geographical spread of an

organisation, and its size in terms of personnel, the person or people undertaking the DBA role may be required to produce a significant amount of documentation, in order to describe various aspects of database activity to users as a whole. This documentation is likely to include the following elements:

Documentation for use throughout the organisation

- Security standards and procedures.
- Details of forthcoming changes to DBMSs being used or of DBMSs to be introduced.
- Database change procedures for developers.
- Meeting documentation to clarify agreed developments and changes with users.
- Programming and query language standards.

Documentation for use within the DBA function

- Documentation for configuration control.
- Records of changes made to the database structure and system.
- Records of test procedures and test runs after changes.

Qualities and roles of the DBA function

The DBA is clearly a key player in the success or failure of a company. The role encompasses a wide range of technical and political/social skills. In medium to large organisations, it is extremely likely that the job of database administration will be split into a number of different parts, and be performed by a group of between 3-5 people. Each of these individuals will take responsibility for specific aspects of database administration. Among the qualities that would be required collectively of this group of people, we would expect to see the following:

- Good communications. The DBA needs to communicate effectively with people throughout the entire spectrum of the organisation. Communications with high-level management is required, in order that the DBA function can be sufficiently informed about strategic directions, so that the database strategy for the organisation can be closely aligned with the business strategy. Frequent communications will be needed with many other levels of the organisation, including Information Systems personnel, end-users and their managers.
- Technical knowledge. In addition to the need to have a sound knowledge of the various utilities and database languages being used to administer user privileges and monitor database activity, a detailed understanding of

the interfaces to other database systems is often required. The knowledge used to tune the performance of a database system ranges from the ability to spot individual trouble spots, such as an inefficiently coded SQL transaction, which can be sped up by recording or the use of indexes, through to variations of system parameters that might be used to make global performance improvements to the DBMS.

- Good understanding of the organisation and its priorities; ability to liaise with management.
- Good arbitrator, in situations where it is necessary to make decisions regarding disputes over access to data or processes.
- Trustworthy - clearly a major part of the security of the organisation is in the hands of the DBA.
- Respected by application development staff and management.
- Cool under pressure. Should disasters arise, for example at the database-application or whole-DBMS level, it is likely that the DBA will be involved in trying to resolve the situation, with minimum disruption to the users and clients of the organisation.
- Flexible. It is possible that years of hard-won knowledge relating to a specific DBMS may from time to time become obsolete, either because that system is replaced by a substantial new release, or because for business reasons, the organisation decides to migrate to a totally different DBMS.

In situations where the functions of database administration are to be organised among the members of a DBA group or team, the following roles might be identified. Depending on the level of work related to each role, one person may adopt more than one role within the overall context of the DBA group.

- Database project leader.
- Documentation and standards developer/disseminator.
- User representative.
- Database systems manager.
- Performance tuning expert.
- Research and development specialist, perhaps looking into database technologies which are new, or new to the organisation, such as replication, Object databases, data mining, etc.

Review question 2

- Describe an example of a situation other than that given in the content of the chapter, in which it may be desirable to store redundant or derived data.
- Describe the arguments for and against storing derived data.

- Explain what is meant by the term configuration control.

Administration of client-server systems

The job of the DBA is to decide what database system and architecture is suitable for the company. He/she needs to be well in tune with the business strategies and objectives, and how the database architecture impacts the development and priorities of the organisation's information systems. He/she is the person to establish policies for maintaining and dealing with database systems in the organisation. He/she is also responsible for ensuring that the system operates with adequate performance to meet the demands of the organisation. Faced with such great responsibility, the DBA needs to know the various issues of client-server architecture and what impact it has on the organisation.

The advantage of client-server architecture is its potential in portability, scalability and interconnectability of clients and servers using various network configurations. In addition, when evaluating Relational DBMS on client-server systems, the DBA must consider many factors besides the architectural model - transaction control, performance, security, integrity, procedure logic and other issues also figure prominently.

SQL has become a standard data access language between client and server machines. We expect to find a mechanism for the server to accept SQL statements and return data and status codes to the client. However, many vendors have added their own extra extensions to standard SQL (i.e. proprietary data-access languages). These extensions are advanced and change as strategy and technology develops.

However, though the extra SQL features can be attractive, the portability of the application in the future may be affected.

It should be noted that no theoretical rule says that only one server may access the database, and there are many reasons for wanting more than one in operation. Once a server has been activated, it is called an instance. A database without an attached instance is a lifeless collection of data and status information; likewise, an instance that is not attached to a database is useless.

Tools used in DBA administration

Tools that are of value to the DBA in supporting client-server systems are as follows:

Data dictionary

The data dictionary is, as we have seen, itself a set of tables and views. When responding to a client request, the server can find any data required about the

data it needs in the data dictionary. It can use the same mechanisms it employs on behalf of clients to read its own data. These are commonly known as recursive requests because the server generates them automatically.

Stored procedures

(These are used by the DBA and programmers, briefly encountered in the chapters on distributed database systems.) Stored procedures are groups of SQL statements which are stored in the server. They can be run like procedures written in standard programming languages, and allow portions of application code (normally commonly executed tasks or transactions) to move from client to server. The server checks the syntax of these stored procedures, The existence, accessibility and data type of each object mentioned in each statement must then be verified. The database engine's SQL optimiser is usually invoked at this point to choose the best access path to the referenced data. Advantages of stored procedures are that they reduce network traffic (fewer SQL statements are sent from client to server), and they improve server performance as they are compiled prior to execution on the server.

Data buffering

During execution of statements that query or update the database, certain data (sent from or requested by the client) is placed in memory. The server tries to keep the data in memory to save disk input/output (I/O) should the next request require data that's still in the buffer. The buffer size should be configured so the DBA can optimise the memory-versus-speed trade-off.

Most servers provide shared buffering, in which data brought into the buffer for one client will be later used by all others. When data is regularly shared by many users, server buffering is a necessity.

Asynchronous data writing

This feature is used to try to smooth out peaks in I/O that may arise during database processing. If I/O slows down, the server starts writing data blocks from the buffers to disk. Since this write activity is scheduled during periods of otherwise low I/O, there is no degradation in performance, even if the data in the buffer is changed later. If an urgent need for buffer space develops, the disk write is already done; the new request can be serviced without a time-consuming disk wait.

Data integrity enforcement on a server

With the client-server architecture, all database processing can be consolidated on a server machine. Such consolidation provides an opportunity to achieve

a high degree of data integrity. Since every database request is processed by a server, if database constraints are defined at the server level, they can be consistently applied. Server-based enforcement of data integrity guarantees data correctness and integrity by having the server enforce constraints on any changes or updates to the database. As such constraints are held centrally, they cannot be bypassed as the data can only be accessed through the server software.

Concurrency control features

One of the challenges in the development of client-server applications is to gain the maximum degree of parallelism on the client computers while providing protection against problems such as lost updates and inconsistent reads. Concurrency control allows multiple clients to access the server and still preserve the integrity of shared data. Updates by users are controlled and isolated to prevent one's changes disrupting or overwriting another's. This is usually enforced by using various automatic locking mechanisms, multiversioning techniques or rollback of partially completed transactions.

Some server implementations provide consistency by blocking writers from accessing the data being read. Others offer snapshots showing the state of data when the read began. Changes are ignored until the next read statement begins.

Communications and connectivity

A characteristic of client-server architecture is that a client application and server software are on different computers. The protocol used to pass messages, SQL and data between them is therefore of crucial importance. As there is no single standard protocol for computer networks, the server has to offer tools to handle the complexities of multivendor networks (i.e. to enable any application to be able to run on any network supported by the server without the application developer needing to be concerned with handling the hardware and network-specific communications issues). However, the connection between PCs and minicomputers is much more complex to implement than a conventional terminal-to-host implementation.

Client-server security

Server security

The built-in security mechanism of the database server provides central data access control, thereby reducing the need for security measures in the client applications. The server normally provides three basic levels of security:

1. User enrolment. This involves granting a user access to the database server itself.

2. Access privileges. This grants users access and privileges to individual database objects.
3. Resource allocation. This controls the amount of disk space allocated on the server to each user's database objects.

Client security

As the general administration of security in a client-server system is handled by the DBA at the server end, the client needs only to be concerned with errors returned from the server when an unauthorised operation is detected.

Network security

The distribution of a system, be it as a client-server or truly distributed database system, requires the additional issues associated with the protection of the data as it is transmitted across the network to be handled. This is most often dealt with by encryption algorithms, which encode the data, rendering it useless if it is intercepted during transmission. Following reception of encrypted data, the receiver of the data will run the decryption algorithm to re-establish the original data values.

Checking of security violations

Journal logging and other facilities are used to locate security breaches or violations in the server. The reason for a user failing action should also be logged so that the DBA can distinguish between a simple human error and attempted security violations.

Review question 3

Within the context of client-server systems, explain the meaning of the following terms:

1. Shared data buffering
2. Asynchronous data writing

Database tuning

Providing an efficient service to users of the database system is an ongoing responsibility of the DBA. In the following sections we shall examine some of the major considerations involved in the tuning of a database system. Most of the examples we shall give are somewhat specific to the Oracle DBMS, but certainly have analogies in any of the other major database systems of today.

Tuning SQL

The optimisation of SQL transactions is a major topic in its own right. There are a number of guidelines which can be followed, which in general will lead to more efficient SQL. In general, the DBA activity in tuning specific transactions should be focused on those which:

- are run sufficiently often to have a noticeable impact on performance;
- access sufficient numbers of records (including intermediate results obtained during the evaluation of the query) to provide scope for transaction tuning.

Guidelines

The following guidelines can be applied when tuning SQL transactions:

- Use indexes on primary and foreign keys, and consider their use on other attributes that are frequently referenced in the WHERE clause of queries.
- Corollary to the above, indexes improve performance for queries that return fewer than approximately 20% of the rows in a table; otherwise it is faster to use a full table scan. For update transactions, indexes can actually make things slower, because of the need to update the index structure.
- Unique indexes are faster than non-unique indexes.
- Several SQL constructs, such as use of the keywords like '%string', distinct, group by, order by, max, etc, lead a query not to use an index. The reason for this is that either the data to use the index is not available (as with '%string'), or the operation implies a sort of the data, in which case all of the rows need to be accessed.
- Compressed indexes save space, but do not provide as substantial an improvement in response time.
- In general, joins are executed more efficiently than nested queries. This may provide scope for recasting an existing nested query in the form of a join.
- Use of short table aliases in queries can improve performance.

Activity 4 - Examining the time response of queries

To examine the times associated with the execution of an SQL command in Oracle, we can use the sql*plus command

Set timing on

Log in to the Oracle system and issue the above command. Experiment then by running several queries to examine the time values returned. For example, experiment for queries that return a single row, that perform a full table scan,

and for JOIN queries. The value returned for each query appears after the rows displayed in the result of the query.

Tuning disk I/O

No matter how much money you spend on high-speed disk technology, there is still no getting away from the fact that disks are slow when compared with solid-state devices. For this reason, most data structures and design options are geared around minimising disk input and output. As long as we are limited to disks as the main medium for storing our data, then researchers will continue to search for methods to improve the efficiency of storing and retrieving data from these devices. The following guidelines are useful when trying to minimise the impact of relatively slow disk I/O processes:

- Reduce disk contention. Contention occurs when several users try to access the same disk, at the same time. If contention is noticeable on a particular disk and queues are visible, then distribute the I/O by moving heavily accessed files onto a less active disk. Distribute tables and indexes on different disks if resources are available.
- Allocate free space in data blocks (i.e. space in a block is used by INSERTS and also UPDATES which increase the size of a row).
- Allow for block chaining by using PCTFREE (Oracle specific, i.e. the percentage of blocks reserved for row expansion) parameter to control/limit chaining. Chaining can be examined using the ANALYZE command. (Chaining occurs when data in a block grows beyond the size that can be contained within the block, and so some of the data must be stored in a further block, to which the original block must have a pointer. We describe this dynamic expansion of data into additional blocks as chaining. Its overall effect is to reduce performance, as the system must follow the series of pointers to retrieve requested data.)
- Seek to minimise dynamic expansion. For example, with Oracle, set up storage parameters in the CREATE table and CREATE tablespace statements so that Oracle will allocate enough space for the maximum size of the object. Hence space will not need to be extended later.
- Tune the database writer DBWR (an Oracle-specific process which writes out data from the buffer cache to the database files).

Note: If the hardware is changed in the future, then it is quite likely that the system will need re-tuning, because many of the settings will be hardware specific.

Tuning memory

As previously mentioned, accessing disk is very expensive in terms of performance, whereas access to memory is much faster. Hence, we want to make the majority of accesses to be to memory rather than to disk. In an ideal world(!), it would be nice if we could load the whole of the database into memory so that all accesses are then to memory, rather than disk. Obviously this is not possible in the real world, so instead the system needs to be tuned to make the best use of the limited amount of memory available.

Oracle's memory can be broken down as follows:

- **System Global Area (SGA):** This block of memory is used for storing data for use by Oracle processes. It is a shared portion of memory for all Oracle processes.
- **Caches:** Blocks of memory used for keeping copies of data that is also on disk, but which can be accessed much quicker here. Hence it makes sense to keep frequently accessed data in cache. The two main caches of concern are:
- **Data dictionary cache:** Requires only a small amount of memory in comparison to the buffer cache, but can have a dramatic effect on performance. The actual size of this will depend on the different types of objects used by applications.
- **Buffer cache:** It is here where tables and indexes can be stored. The most frequently accessed tables and indexes should be stored here to minimise disk I/O as much as possible.
- **Program global area (PGA):** A PGA is a non-shared memory region that contains data and control information exclusively for use by an Oracle process. The PGA is created by Oracle Database when an Oracle process is started.
- **User Global Area (UGA):** The UGA is memory associated with a user session.
- **Software code areas:** Software code areas are portions of memory used to store code that is being run or can be run. Oracle Database code is stored in a software area that is typically at a different location from user programs — a more exclusive or protected location.

Note

Memory is also required for operating system use, hence other factors need to be taken into account, such as memory allocation for paging and virtual memory. For example, if the system is multi-user, then an increase in the number of users currently online could alter the performance on the machine quite dramatically.

Tuning contention

The term 'contention' refers to a problem which can arise in most areas of computing. It occurs when several processes make an attempt to gain access to the same resource at the same time. This will obviously result in a performance degradation, as one or more processes will need to wait until the resource is available. There are three main areas concerning memory contention in Oracle:

- Data blocks. Usually occurs when users attempt to update the same block.
- Rollback segments. All transactions use the rollback segments, so if there is only a small number of segments, contention is quite likely. A guideline given by Oracle is to use one rollback segment per five active users.
- Redo log buffers. Any block modification will write data to this buffer. The 'redo space waits' statistic can be used to provide information on contention for this buffer.

Tools to assist performance tuning

Having looked at the various factors that can affect performance in a system, what tools are available to aid the tuning process? Depending on the type of database, there will be a selection of tools available to monitor the system, allowing the DBA (or similar) to see the effects of tweaking the system.

Monitors can be broken down into two types:

- Software monitors
- Hardware monitors

Software monitors

These are programs which can be called up when necessary to provide statistics on the state of the system. These tools are flexible and may even be specially written by the DBA, although most vendors now supply these. Unfortunately, as these tools actually run on the system, they themselves apply an additional performance overhead, requiring CPU time, etc, in order to execute.

Hardware monitors

Hardware monitors consist of electronic devices which record data collected by probes, where each probe is connected to circuitry in the machine and/or peripherals. A major advantage of these, is that they do not interfere with the operation of the system.

As well as monitors, tools are available which aid database set up, loading, checking, back-up, and recovery and general database maintenance.

Other performance tools

An overview of some of the performance tools which are commercially available are as follows:

- Explain facility. This excellent facility allows the user to obtain information regarding the optimiser's choice of access strategy for a particular SQL statement. It is available for both Oracle and DB2. It works by examining the access paths that would have been chosen for the execution of a particular query. Note that it doesn't actually execute the query, which is a major bonus if trying to analyse the processing of a lengthy transaction.
- SQL*DBA monitor facilities. This Oracle utility allows the DBA to monitor database activity, which is grouped into areas such as Locking, File I/O, Statistics and Tables. This utility allows most bottlenecks which are causing performance degradation to be discovered.
- RUNSTATS utility. A DB2 utility which calculates statistics based on data stored. It is usually run after data is loaded or after a significant amount of updating has taken place.

Concluding remarks on database tuning

The subject of database performance tuning continues to receive a great deal of attention in both academic and industrial establishments. As databases become more complex, they place an even greater load on available resources, hence improved techniques for getting the best performance out of the system will always be in demand.

Discussion topics

1. Evaluating database systems

   Database administrators require from time to time to evaluate database systems in order to assess whether they will meet the needs of their organisation. This process will involve discussing requirements and features with the vendors of the database systems being considered, viewing of demonstrations, and finding out the opinions of other users of the systems. Some of the requirements for the system may be very specific to the organisation, but there are many characteristics which are generic to the evaluation of database systems. In discussion, identify the factors that you consider to be important in evaluating a DBMS. You may wish to start by making a personalised list of factors, and proceed by exchanging and discussing the lists prepared by other people on the course. In particular, given the fact that no DBMS is likely to fulfil all the requirements stated, it may

be interesting to seek agreement on a prioritised list of your top 10 or so requirements. There may be a few cases where it is not possible to place a higher priority on one factor than another, but these situation should be kept to a minimum, to sharpen the selectivity of your prioritised list.

2. Distribution of the DBA function

   In an organisation using a truly distributed database system, it would be possible to allocate aspects of the database administration function to individuals based at sites where significant amounts of data are to be stored. Consider in discussion what may be the possible benefits of distributing the database administration function in this way, and whether anything might be lost in distributing the function, when compared with a more centralised approach.

Answers to the chapter review questions

Chapter 1

Review Question 1

Review Question 2

Review Question 3

Review Question 4

Review Question 5

Review Question 6

Review Question 7

Review Question 8

Chapter 2

Review Question 1

Review Question 2

Review Question 3

Review Question 4

Review Question 5

Review Question 6

Review Question 7

Review Question 8

Review Question 9

Chapter 3

Review Question 1

Review Question 2

Review Question 3

Review Question 4

Review Question 5

Review Question 6