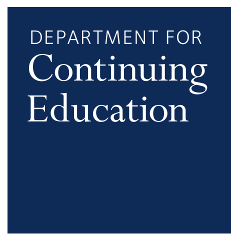


University of Oxford Department for Continuing Education

ADITSAD unit 3: Systems analysis and design – data organisation

Contents

Unit 1. Data and the database environment 2
Unit 2. Relational theory 23
Unit 3. Database design 54
Unit 4. Database creation and manipulation 76
Unit 5. Implementing database solutions 113
Unit 6. Post-relational databases 138

This document is a print-friendly version of the online course *ADITSAD unit 3: Systems analysis and design – data organisation*.

Naturally, some website features will not be available in this print version.

© Copyright the University of Oxford and contributors.

Unit 1. Data and the database environment

1.1. Introduction and objectives

Topic 1 looks at the problems of storing and manipulating data using traditional process-centred software, and the development of the database approach as a solution.

By the end of this topic you should be able to:

- describe the nature and problems of process-centred computing
- describe and evaluate the database approach
- describe the database environment
- describe and evaluate data models
- describe the management and technical aspects of data security
- evaluate data security policies

1.2. Data and information

You may have come across this question before: what is the difference between data and information? We would like you to consider it again at the start of this unit.

The notion that people can communicate with 'thinking' machines is based on the facts (recognised by many, but notably by Claude Shannon[1]) that a machine can recognise the states 'on' and 'off' and that these states can be represented as 1 and 0. If something can be represented as a stream of binary digits (and most things can) they can be communicated to and manipulated by a machine.

Data is ultimately a stream of binary digits that needs to be mapped by some algorithm to a data model before it is capable of becoming information. If the SETI[2] project eventually identifies a repeated pattern of 0 0 1 0 1 0 1 0, it would certainly be significant. Would it be information? In the process of analysis we might map the digits to the ASCII algorithm and conclude that the message is the value 42. We might then, using the 'Douglas Adams algorithm'[3], conclude that what we have received is the answer to 'Life the Universe and Everything'. Would we then have information?

I have a file that contains a bitstream which begins 10011101101111110100 and continues for several thousand bits. Mapping it through the ASCII algorithm and the English Language algorithm I can convert it into part of an insurance policy document:

© Copyright the University of Oxford and contributors.

```
Notwithstanding anything herein contained to
 the contrary the value of the policy at a
 retirement date earlier than the maturity date
 shall be ascertained as the sum (hereinafter
 referred to in this endorsement as the 'reduced
 sum') secured by the premiums paid in respect
 of this increase prior to that date, in
 accordance with the society's rates in force at
 the date hereto and adding to the reduced sum
 the reversionary bonuses attaching to the
 policy in respect of this increase reduced in
 the proportion which the reduced sum bears to
 the undermentioned increase in the sum assured.
```

When I map this bitstream to characters and then to words, have I converted the data into information?

What are your definitions of data and information? Let us have your insights into the mapping between data and information by posting your views and experiences in the unit 3 forum.

[1] many web sources, for example http://www.thocp.net/biographies/shannon_claude.htm

[2] Extra Terrestrial Intelligence project - http://setiathome.ssl.berkeley.edu/

[3] Adams, Douglas. The Hitchhiker's Guide to the Galaxy ISBN 0345391802. For a quick reference, do an internet search for "deep thought" and 42.

1.3. The problem of data

We concluded Section 2 with a discussion of mapping. An example of mapping with which you are probably familiar is the mapping of a two-dimensional array to a series of bytes using a row major or column major algorithm. Here the logical structure is a matrix or table that is mapped by the chosen algorithm to a simple stream of bits.

To achieve this mapping you need to be able to define the logical structure precisely and completely.

Logical data models are often discussed in terms of their structure, but they can only be fully defined in terms of the operations that can be carried out on them. For example a Stack is often described as a pile of coins, or one of those push-down coin holders sometimes used by bus drivers/conductors. But to define it fully you need to talk about the operations it supports. If you study the formal definition of abstract data types your definition of the Stack type is likely to include something like:

```
Operations :
   1. Create(S’) --> S’’ = (  )
   2. Empty(S’) --> true or false, S’’ = S’
   3. Push(x;S’) --> S’’ (where S’ = (d1,d2....dn)    and    S’’ = (x,d1,d2.....dn)
   4. Pop(S’) --> d1 (where S’ = (d1,d2....dn)    and S’’    = (d2......dn)
   5. Top(S’) --> d1, S’ = S’’
```

You do not need to know any detail of the above for this unit. It is included merely to illustrate and underline the principle that data types are defined in terms of their operations. It also illustrates the degree of precision required to produce an unambiguous definition of the structure. A database, though much more complex, is

© Copyright the University of Oxford and contributors.

based on abstract models defined in terms of the operations that must, may or cannot be carried out on the data. These concepts are central to the definition of database objects, and we will return to them when we study the formal specification of a database.

You may also be familiar with the complex data types used in high-level programming languages. If you have studied programming recently you will be most familiar with the classes and attributes of the object-oriented paradigm. In this topic, however, we are looking at the history of data processing, so the illustrations used are taken from more traditional high-level languages.

An application to automate the payroll of a company would need to handle a lot of data, many of which would be related to employees. A program, written in Pascal, would probably declare a new data type called Employee as in this (somewhat oversimplified) example:

```
type employee = record
    name: String[200];
    address:String[255];
    worksNo:String[6];
    rateOfPay:real;
    PAYEcode:String[6]
end;'
```

The payroll application would declare individual instances of the employee type using:

```
thisEmployee:employee;'
```

or more realistically, an array of employee types:

```
theseEmployees:array[500] of employee'
```

and it would load and re-save the data structures as required from the backing store. The application would be responsible for mapping the logical data to the backing store. Each programming environment has its own algorithms for mapping, so in this example the mapping would be provided by Pascal-specific code built by the Pascal compiler.

In early applications data were often held on tape in indexed sequential files, but more sophisticated ways of storing and retrieving data emerged which allowed more flexible random access. In applications of this sort the main data type is usually called a **record** and the individual data items in each record are usually called **fields**. We shall later reflect on the correspondence between these structures and those used in defining a logical database.

As data processing became more powerful, the same firm might have set up a Sales system (this time, for example, written in Cobol). It would be necessary to keep some details of the employees who handle the sales so again there would be an Employee record, though its declaration and format would be different. For example, it might be coded:

© Copyright the University of Oxford and contributors.

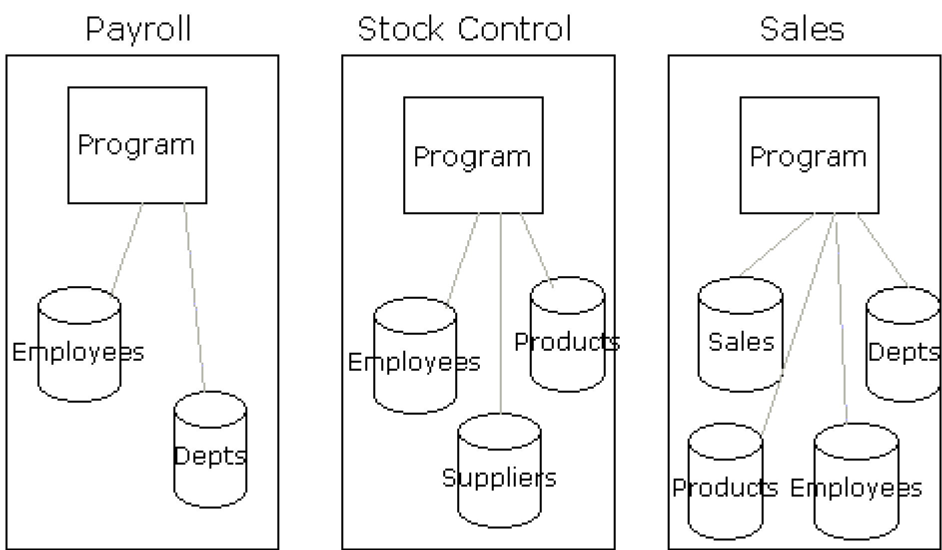
```
DATA DIVISION.
  FILE SECTION.
     FD EmployeeFile.
        01 WorksNumber    Pic 9(3)
        01 FullName       Pic X(255)
           02 LastName    Pic X(100)
           02 FirstName   Pic X(155)
        01 Department     Pic X(3)
        01 Grade          Pic 9(1)
```

Again the application would create and handle many instances of this data structure. These would be committed to the backing store using an algorithm decided and written by the application programmer using the conventions of the Cobol development environment.

Meanwhile the empire-building Human Resources department might have transferred its traditional card index of employee details to a bought-in software card-indexing system.

In each case there would be a mapping of the logical structure of data in the program with the way it is physically represented on the backing store. In each case, however, the mapping would be a product of the design decisions of the programmer, the programming paradigm and the application used to create the particular application.

The ‘Silo’ concept is one often used when discussing fragmentation and poor communications in business management. It is an apt metaphor here and early business systems can be envisaged as running in silos.



Process-centred Data Storage

Forum activity

The ‘problem of data’ as it came to be known was a product of this piecemeal approach to storing data.

Pause at this point, make a list of the problems you can see arising in the scenario outlined above, and post one of the problems to the unit 3 forum. When several people have posted, discuss the views expressed and suggest a concise summary of the historic ‘problem of data’.

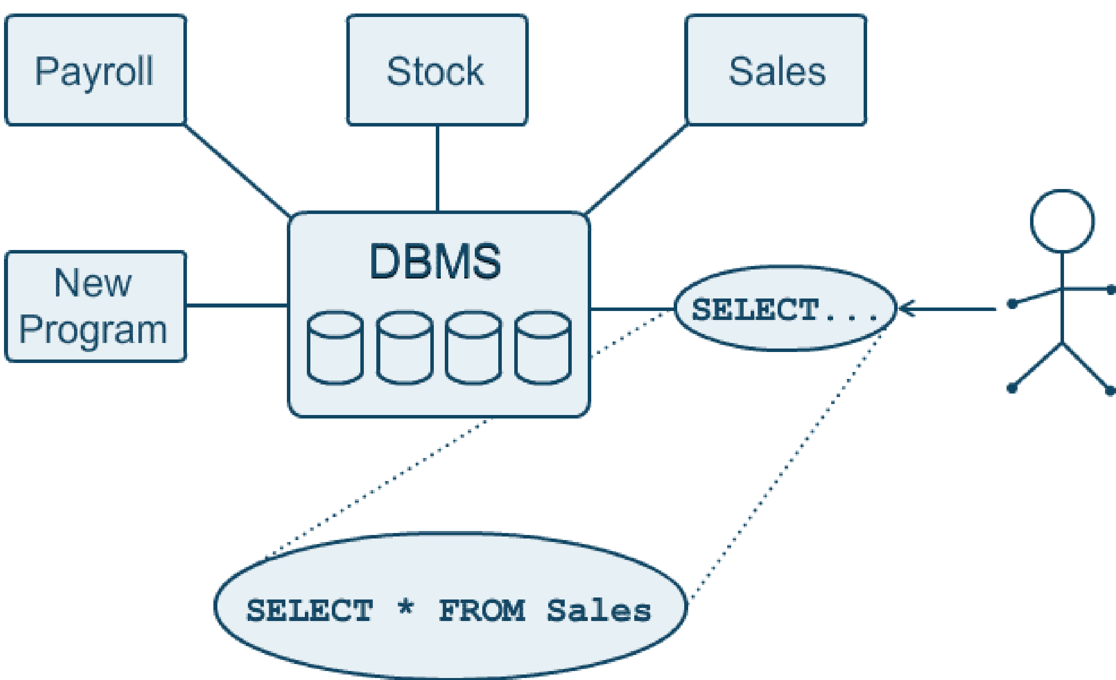
© Copyright the University of Oxford and contributors.

1.4. The database approach

The limitations of the traditional file-based approach, which we considered at the end of the last section, arise ultimately because:

- the definition of data is embedded in the applications that use it
- there is no independent way of accessing, controlling or validating the data

The need to overcome these limitations was the motivation behind the revolution in data processing that produced the database approach. It is sometimes called the 'Copernican Revolution in Data Management' because it changed the focus from *process-centred* systems to *data-centred* systems. Instead of data being a seen as part of an application, it acquired an independent existence. No longer a satellite of an application, it became the centre around which systems were built. Instead of the silo configuration we saw in Section 1.3, the new approach envisaged an integrated data management system that all applications could use:



Data-centred Computing

It would be possible not only for different applications to use the same data, but even for ad hoc queries to be made by quickly developed new programs or scripts.

At least, that was the theory. Early database models did not achieve all these objectives and even today an organisation's data often remains duplicated, in separate databases. We will return to this shortly, when we have considered the advantages and disadvantages of the database approach.

© Copyright the University of Oxford and contributors.

Some preliminary definitions

We will develop this theme as we progress through the unit, but for the moment here are some preliminary definitions for the purposes of this section:

Database

A database is a shared collection of logically related data that is defined and controlled independently of user applications. The data is ultimately stored in bits on (usually) magnetic storage, but it is mapped to a logical model which determines how the data is presented to users. As well as the operational data, the database contains metadata recording its logical structures and data types. Depending on the product and the nature of the data, it may be possible to build into the metadata some of the semantics of the operational data, and thus provide centralised control of the integrity of the operational data.

Database management system (DBMS)

A database management system (or DBMS) is the software used for developing and managing a database. The DBMS contains processes that allow the creation and manipulation of logical data structures. The DBMS also defines and controls access to different groupings or levels of data. Most database management systems also provide transaction processing. It is often necessary to make changes to items of data in one transaction, for example when transferring a sum of money between accounts. Transaction processing ensures that if any part of the transaction fails the state of the data is 'rolled back' to the beginning of the transaction.

Pause for thought

Identify the three most important characteristics of a database using only three words. Do the same for a database management system.

My Conclusion

The words I thought of are:

- Database: shared, controlled, metadata;
- Database management system: creation, access, transaction.

If you think you have made a better choice (and you may well have done so) post your views in the unit 3 forum and see what others think.

Advantages of the database approach

Data accessibility

The database contains its own definition of the data model and can be interrogated to find not only the operational data it holds but also the logical structures used to access the data. This means that it is possible for applications to access the data without having to use a particular programming or data modelling paradigm. A database can also cope with ad hoc queries that were not envisaged when the applications currently using it

© Copyright the University of Oxford and contributors.

were written. (For example, when the payroll application was written it might not have included the facility to make reports analysed by both gender and salary level, but it would be relatively easy to obtain such information from a database using an ad hoc query.)

Data integrity

The validity and consistency of stored data is greater because, in addition to the checks carried out by user applications, the central database carries out its own integrity checks. For example, the database can help to prevent an application from storing duplicate records, insufficiently linked records or records containing the wrong types of data. These checks are imposed using constraints that are an important part of database design and definition.

Control of data redundancy

As there is a central store of data there is no need for it to be duplicated. For example, the Payroll, Sales and HR applications can use the same employee name and address data. Note we say *control* rather than *elimination* of redundancy. There will usually be duplication of some data because this is how tables are linked, but a database reduces the amount of redundancy and, more importantly, controls it by imposing key and other constraints.

Sharing of data

The use of a shared database can eliminate 'swivel-chair' computing (reading data from the screen of one system and typing into the keyboard of another). In practice, however, many organisations still have separate databases for separate functions and rely on manual transfers of data between them.

Pause for thought

Given the obvious advantages of being able to share data, why do you think so many organisations still run separate databases and transfer data between them manually.

My Conclusions

> A major problem with the database approach is that people must be willing to share data and be ready to relinquish control of their data to the ownership of the central database. People do not want to risk compromising a system that is working well, and may be unwilling to share data. There are also issues of organisational politics which can prevent the most effective solutions being implemented. There is also a misplaced fear that confidential data might be exposed, for example that sharing data about employees could give sales staff access to tax code numbers. We shall examine later how database management systems can prevent this from happening.

More and better information

If all data is stored in the same place and according to the same data model, it may be possible to make links between data-sets that provide improved or even new kinds of information. For example, the HR system (which already uses a shared Employee table) might also make use of the Sales table to provide performance

© Copyright the University of Oxford and contributors.

management profiles. The integration of data is taken to its ultimate conclusion in data warehousing and data mining, where all the data of an organisation is collected and manipulated in the hope of finding new information. (We will return to this theme later in the unit.)

Improved security

A database may not necessarily improve security, but it often does. The management of large amounts of data and of different users means that a database management system must give security a high profile. Data will be backed up frequently and protected both physically (since the server is in a secure room) and logically (through passwords). Another feature of a database management system, not often found in application programs, is the ability to manage transactions. If data is being transferred from one table to another, the first half of the transaction will be rolled back if the second part cannot be completed.

Easier maintenance and upgrading

If the load on the system requires more capacity or a faster way of storing and accessing data, this can be done on the database without affecting the user applications. The physical storage of the data may change, as may the algorithms used to manipulate the raw data, but to user applications the logical view of the database remains the same.

Lower costs?

The question mark reflects a difference between theory and practice. If data models and data storage systems have to be designed into every application there is duplication of costs. In practice, however, applications tend to be produced one at a time, and the cost of a database is greater than the costs of internally managing data in a single application.

Pause for thought

Which three of the above advantages do you consider to be the most important?

My Conclusion

There is no right answer to this question. I think I would say:

- Accessibility
- Control of redundancy
- Improved security

If you have other ideas, why not post them on the unit 3 forum.

Disadvantages of the database approach

Against the many advantages of the database approach there are some important disadvantages. A database may not be always the right answer.

© Copyright the University of Oxford and contributors.

Concentration of risk

Though the chance of data loss or compromise is smaller in a database, if it did happen the effects could be catastrophic.

Complexity

A modern database is an extremely complex piece of software and can only be operated by highly trained personnel.

Cost

The cost of a database management system varies, but the kind used for enterprise-wide shared data systems are very expensive and represent a major investment. In some circumstances the less satisfactory stand-alone system is the right choice because it is the only affordable choice.

Conversion

An organisation starting from scratch would build applications that use a shared database, but most have legacy systems that need to be integrated into the new database. That is difficult, risky and costly. Again there is a difficult trade-off that managers have to consider. They must weigh whether the advantages of the introduction of a DBMS are sufficient to warrant facing the risks and costs of conversion.

Performance

A file-based data store holds data in (broadly) the same form as the application, so data can be loaded instantly and used with little or no transformation. A database may need to undertake a considerable amount of processing – joining tables, etc. – before it is able to provide the required data. And when the record set is received by the application it will need to be transformed once again from the common logical structure of the database to the internal data structure of the application.

If the database is a shared resource then the application will also be competing with other applications for the processing power of the database.

It is often possible to manage this potential fall in responsiveness by upgrading hardware, but it is potentially a real problem which must always be considered when adopting a database solution in place of existing stand-alone systems.

Pause for thought

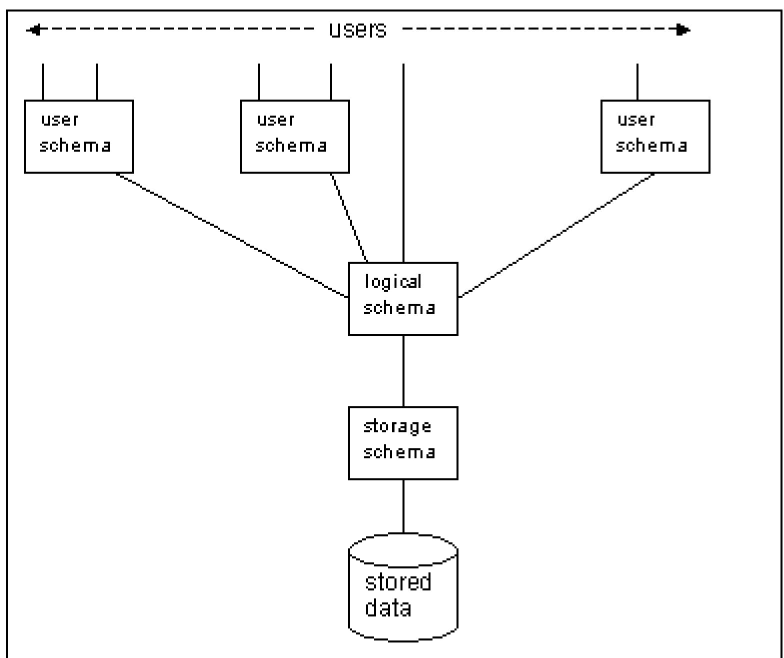
These are the disadvantages I can see, and you will find similar ones in most textbooks. But how up-to-date are these views? Have you any experience of using database systems? Do you agree or disagree with the above analysis? Please share your thoughts with others on the unit 3 forum.

1.5. The database environment

We began this topic by thinking about the way data in user processes are mapped to stored data. In a database this mapping is defined and implemented using schemas. We meet schemas in a number of places, and the word has slightly different meanings depending on the context. The schema of a database is often seen as a catalogue

© Copyright the University of Oxford and contributors.

of the structure and semantics of the data structures in the logical model. This is, however, properly termed the **logical schema**. It is this schema that we design and include in our documentation, and which we implement when we create the database. The term 'schema' is also used, however, to describe the mapping within the database. A database will have a **storage schema**, which maps the raw data to the logical model, and a **user schema**, which maps user views to the logical data model. The place of schemas in the database environment can be pictured like this:



Schemas in a Relational Database

The storage schema usually comes with the database management system and we do not consider it here except to note that, at least in theory, it is possible to change the way physical data is stored and accessed (for example, to provide more capacity or faster access) without affecting the logical schema.

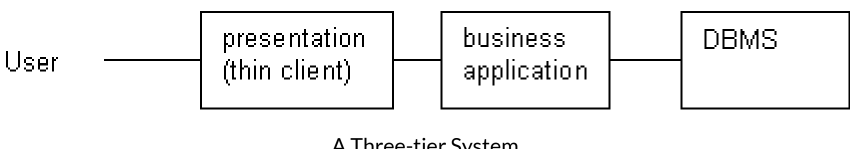
The logical schema is the main focus of our study in this unit. It defines the data model, recording both its structure and the operations that can be carried out on it. Part of the logical schema is a standard core of structures and operations relevant to the abstract data model used (for example a Relational DBMS will include the Structured Query Language). The logical schema also includes the definition of data structures and constraints reflecting the structure and semantics of data in the particular database.

The user schema is a similar mapping between the whole of the logical database and the part of it that particular users need (or are allowed to see). The users will generally be application programs, but they can be individuals using the schemas directly. The logical schema may contain an Employee table which records, as well as names and addresses etc., things like tax codes, salary scales and performance reviews. User schemas would be used to ensure that the Employee records seen by the Sales department included only the basic details, whilst Payroll would also be able to see the salary and tax data, and HR would see the performance management data. Thus user schemas have an important part to play in making the database and its data secure from unauthorised access. The user schema also has another function: it ensures that (regardless of confidentiality) the user only sees what is needed, thus reducing the scope for accidental corruption of data and simplifying the programming of user applications.

© Copyright the University of Oxford and contributors.

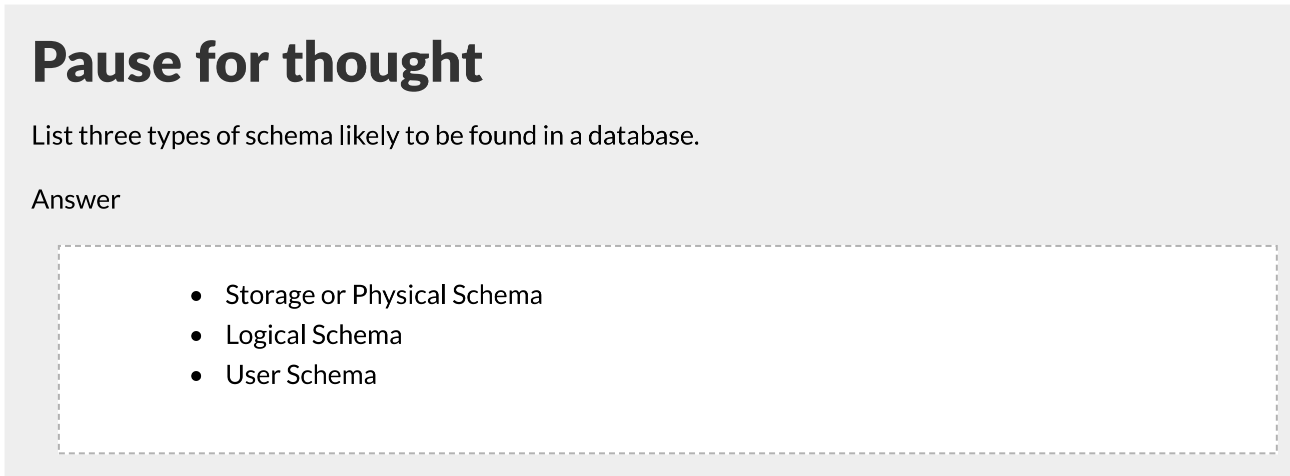
In the diagram above (Schemas in a Relational Database) users are shown using the logical schema directly. This is not entirely accurate but it is a reminder that some users may be given access to the logical database through an interface that does little more than pass requests to the logical schema. We shall be using such an interface when we come to study and practise implementing and manipulating a database.

The interface with the human user is usually through an application program. It may be very closely coupled, for example when the Query by Example facilities of Microsoft *Access* are used to interrogate an .mdb database. In enterprise applications, however, the user application is often split between the display function and the business function.



A Three-tier System

The separation of the presentation function from the business function makes it possible to provide access to a large number of geographically separated users through one instance of the business application. The business function can be extended or changed without the need to amend or update a large number of client-side programs. The thin client approach is frequently used in new database applications because it enables users to access data using very simple software (often just a browser), thus reducing the costs of both kit and training.



Pause for thought

List three types of schema likely to be found in a database.

Answer

- Storage or Physical Schema
- Logical Schema
- User Schema

The people in the database environment

The nature and complexity of the applications in the database environment depend on the people using the database and it is useful to have some terminology to refer to them. In this unit we shall refer to the following types of database users:

Casual users

Databases are increasingly accessed by people who have no knowledge of the database or the application they are using to access it. They may even be unaware they are using a database. A good example of a casual user is someone shopping on the Internet. The data displayed must be instantly understandable and any actions needed must be explained very simply and, if possible, be intuitive. In the following example most web users would have no difficulty in understanding what the data means and what they need to do in order to get more details or make a purchase.

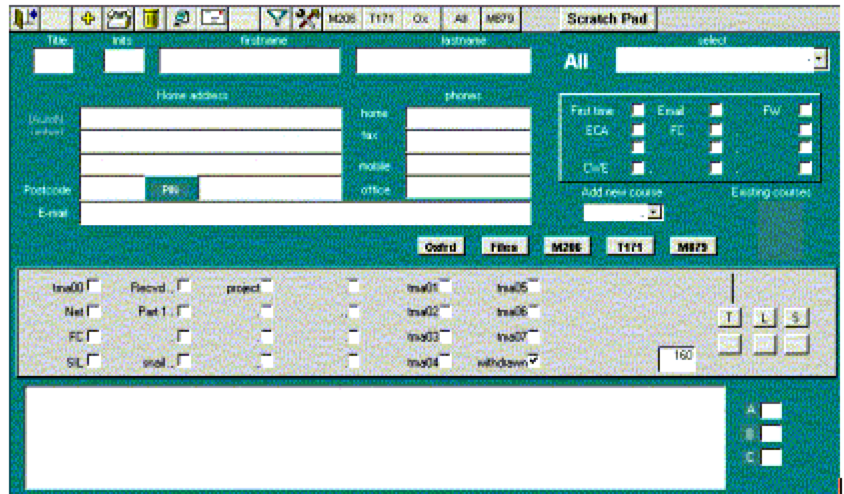
© Copyright the University of Oxford and contributors.

Product	Price	in stock	details	basket
Pink Widget with flanges	£235	23		
Ortho-Widget assembly	£300	10		
J S Widget	£1	100		
Mark II Super Widget	£300	30		
Blue widget	£0	0		
Black Widget	£0	0		

Interface for a Casual User

Parametric users

The parametric user accesses the database through a sophisticated interface.



Interface for a Parametric User

Like casual users, parametric users do not have direct contact with the database and will usually have no knowledge of it. They will, however, have been trained in the use of the application and will possess skills enabling them to do things a casual user would be unable to do. Parametric users see themselves as qualified in using the 'Sales System' or the model and the operations that can be carried out on it, and they access the database using a language that provides for ad hoc retrieval and manipulation of data. In a relational database this language would be Structured Query Language (SQL). The general user is often a computer specialist but in some organisations management support staff and some senior managers are trained for this kind of access. The SQL queries may be entered through a specialised application or may be through a built-in access function of the database management system. When we come to study SQL in Topic 4 we will start by acting as general users making ad hoc queries of an online database.

Database Administrator

The Database Administrator (DBA) is responsible for managing and controlling the DBMS. The role of DBA may be filled by a team of people, or it may be just one aspect of someone's job. Whatever the size of the system, however, it is essential that there is clear responsibility for database administration. The role involves documenting and enforcing standards, policies and procedures, and implementing them within the DBMS. An

© Copyright the University of Oxford and contributors.

important feature of this role is the implementation of access controls through user schemas and management of privileges. The DBA is responsible for the physical database design though, in practice, modern database management systems provide only modest scope for changing the physical schema. The backing up and recovery of data is a universal responsibility of the DBA.

Data Administrator

This is a separate role but it is often combined with the DBA role. The Data Administrator is involved in the determination of corporate data requirements and the design of the conceptual and logical database. The DA decides the business requirements and the data models that support them, and the DBA implements the models in a DBMS.

Data Security Manager

We look in more detail at this role in section 1.7. It is often combined with the role of DA and/or DBA.

Database Programmers

Clearly database management systems involve design and coding but we are not here concerned with the production of the DBMS software. Once delivered and installed there remains scope for programming the DBMS to optimise its performance in the particular context. A common reason for programming a DBMS is to optimise the way queries are executed. Consider a query made of a relational database in which the user wants to

- join three tables together
- select from the result a subset of rows that satisfy a particular criteria
- select from the result a subset of the columns

It is possible that in a particular context the query could be much more efficient if the restriction in the rows selected were made in one of the tables before the tables were joined. The database programmer is able to change the way the DBMS executes categories of queries. Another useful product of the database programmer is the stored query. Even trained professionals can have difficulty putting together the long and syntax-critical expressions needed to query a complex database. A stored procedure allows a previously written query to be identified and executed by its name.

Database programmers may use a high-level programming language like C++ or Java, a proprietary DBMS language or, for the writing of stored procedures, the relevant data manipulation language (for example, SQL in a Relational Database Management System).

Data dictionaries and metadata

Database management systems include facilities for users to find out (subject to access permissions) the details of the database schemas. These facilities are sometimes referred to in aggregate as the System Catalogue. They include information and processes used by the Database Administrator to document and maintain the system, and facilities that can be used by ordinary general users. For example. a command ‘Show Tables’ will list the tables in a mySQL database.

© Copyright the University of Oxford and contributors.

1.6. Data models

Data models are at the heart of the database approach. Users (software applications and people who are general users) need to work with a common logical model of the data. In this section we look briefly at the hierarchical and network models, which preceded the development of the relational model. They are of interest as the context in which the relational model emerged, but you do not need to know these models in any detail. The main focus of this unit is the relational model, which we examine in detail in Topics 2–5. In Topic 6 we will look at object-oriented and other post-relational models.

A data model can be very simple. One you will have met, though possibly not noticed, is the simple attribute/ value pair used to transmit data within an HTTP request. For example, a link to a dynamic page on a website might be:

```
http://www.betaGamma.co.uk/products.php?product=”23”
```

On the other hand in some object-oriented databases the model is very complex, reflecting the richness of classes, inheritance and polymorphism.

Most models, however, are based ultimately on the idea of a **tuple**. This is a term borrowed from mathematics (where it means, broadly, an ordered list of items), but in computer science it refers to a group of related data identified by a unique field name. A model of data about a person might be based on the tuple:

```
(name, address, phone)
```

The flat (or two-dimensional) model

The use of the tuple model immediately suggests a table structure, and most models are indeed based on the logical concept of a table consisting of rows and columns. Some data-sets can be envisaged as a single table and can be successfully implemented using a sufficiently powerful spreadsheet (for example Microsoft *Excel* or *Oracle Open Office*). If you have not used a spreadsheet as a database and you have Microsoft *Excel* you may like to work through the exercise flatDatabase.zip [Link not available in this medium, please view it in the online course.],which demonstrates a simple but very effective flat database.

The following file provides a good *Excel* database tutorial:

Lawrence_Tutorial.zip [Link not available in this medium, please view it in the online course.]

These examples of flat databases are tied to a particular application, but they can also be accessed by other applications either directly (for example using Microsoft Common Object Model) or by exporting the data in comma-delimited format.

A flat database can be very useful and powerful, but it is unsuitable for all but the simplest of data-sets. The following two-dimensional database is not capable of providing a common model because it seeks to model data which in fact have more than two dimensions.

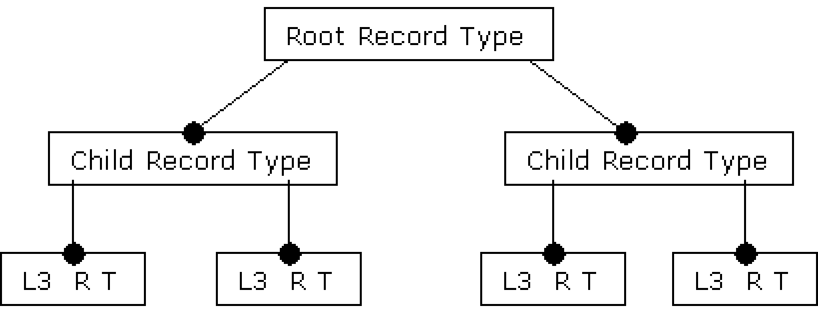
Name	Project Code	Project	% of time
John	TX123 PD345	Amend Network New Program	50 50

© Copyright the University of Oxford and contributors.

Mary	PD345 AB123	New Program Relocation	75 25

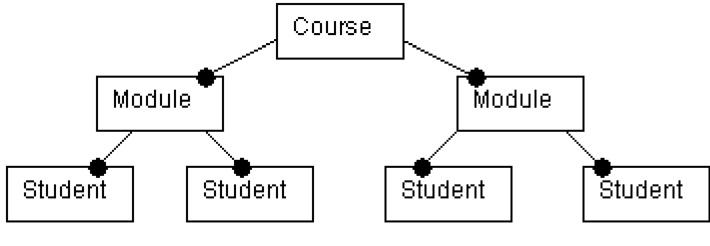
The need to cope with further dimensions led first to the development of navigational databases, which include the hierarchical (or tree) model and the network model. These models are characterised by the use of paths or pointers to navigate to the required record. The distinction between the navigational and relational models can be illustrated by thinking of ways of directing someone to a house. Using a navigational model I might say, 'Start from the town centre, then go to the fourth street, then to the fifth house.' In the relational model I would say, 'Go to the house where the street name is "Garden Street" and the house name is "Dunroamin."'

The hierarchical model



The Hierarchical Model

This model addresses some of the problems of three-dimensional data-sets but it leaves others unresolved. In particular it cannot cope with many-to-many relationships. Consider for example the following model:



Data Redundancy

It is, of course, unlikely that the database would be structured in exactly this way, but similar problems would arise whichever record type was chosen as the root. There is redundancy and data entry is restricted. (For example, it is not possible to keep a record of a student prior to enrolment on a module.) Various fixes were used to get round these limitations but at best they worked only because the database was coupled very closely to the application using it.

One advantage of the hierarchical model, and a reason why it is still in use for some applications today, is that the records within each record type are held in the form of a linked list enabling the data to be easily traversed.

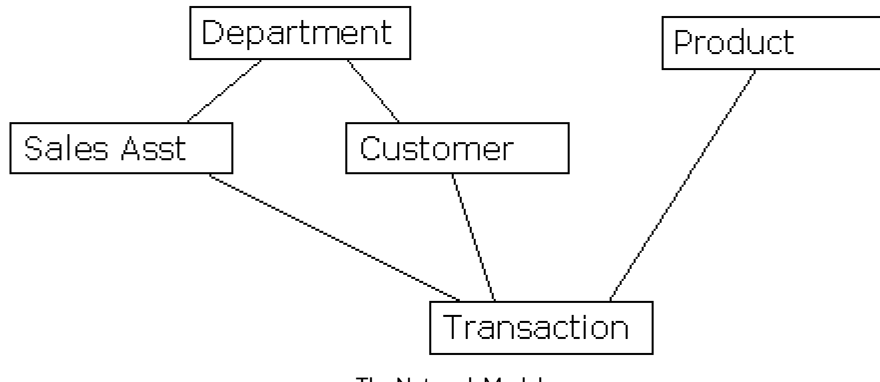
© Copyright the University of Oxford and contributors.

Network or CODASYL models

The **Network Data Model** (or **NDM**) was proposed by a task group of the 'Conference on Data Systems Language' (CODASYL), which was seeking to improve standards and performance in business computing, and systems that use this model are often called CODASYL databases. The NDM was developed, among other reasons, to overcome the inflexibility imposed by the hierarchical model and to reduce the degree of data redundancy.

The network model has two main object types: a **record type** and a **set type**. A record type is a template that defines which data are contained in records of the named type. Thus a record type for an employee would include name, address, TaxCode, etc. Within the database there would be many Employee records each conforming to the Employee record type. Records in a CODASYL database belong to sets. The set type defines the **owner** and **members** of the set. You might have, for example, a set type that defined the owner of the set as a particular Department record, with a number of Employee records defined as members of the set. Individual records can be members of more than one set. For example, the above Employee records could also be members of the set owned by a Region set type.

Another example would be orders or transactions that would typically be included in sets owned by Customer, Product, SalesPerson and Sales record types. Since the particular transaction is included in only one record, redundancy is reduced, while there is also flexibility in the way the records are grouped for access and updating. In the network model a record can effectively have multiple parent records as well as multiple child records:



The Network Model

1.7. Data security

An organisation's data assets form a valuable and often sensitive resource, which can have strategic importance for the organisation. Where they are managed by a database management system there is a concentration of risk, and that risk must be controlled and managed.

Security is always an important issue for the systems engineer. Even where the topic is not explicitly included in the negotiated statement of requirements, security issues must be considered in order to comply with the professional standards laid down in the British Computer Society Code of Conduct.

The database environment is inherently risky. A database is a shared resource, often used by a large number of people, and there are many opportunities for the data to be compromised by deliberate action or by accident. Data can be compromised by:

- unauthorised use or disclosure of data

- corruption of data
- loss of data

Data can be compromised at many stages in its daily existence. It can be compromised through unauthorised access at a remote terminal, through interception of the messages between the terminal and the DBMS, and even by physical access (perhaps with a powerful magnet) to the physical storage medium. The broader issues of data security management are outside the scope of this unit, but it is important to have some insight into the kind of security management system the client will (or should) have implemented. If you are not already familiar with the broad concepts of data security management you should read this background paper information_security_management.pdf [Link not available in this medium, please view it in the online course.] which outlines the relevant ISO and BSI standards.

The systems engineer must also be familiar with data protection legislation affecting the client company and its area of operation. Data protection legislation varies between countries but in most of Europe, and increasingly in other places, there are rules about how personal data is kept and used. The detailed study of data protection laws is outside the scope of this unit but again it is important to be familiar with the principles which characterise most regimes. The eight principles on which the UK legislation is based are that data must be:

- fairly and lawfully processed
- processed for limited purposes
- adequate, relevant and not excessive
- accurate
- not kept for longer than is necessary
- processed in line with the user's rights
- secure
- not transferred to other countries without adequate protection

As the need arises the application of these principles to particular situations can be checked on the Information Commissioner's website where there is clear and helpful guidance (though sometimes it takes a bit of finding).

It is the client's responsibility to comply with data protection laws, but IT professionals should not be party to systems designed to operate in contravention of them.

Security in database systems

Security in database systems is based on:

- authentication
- privileges
- audit
- encryption
- backup and recovery

© Copyright the University of Oxford and contributors.

Authentication of users is usually a function of the background system on which the database system runs, but it is important to ensure that it is satisfactory and that the authentication details feed through to the database management system. It is not enough that the database is protected by a password: it must be possible to identify the user so that the correct privileges are granted and activity can be audited.

Even a user who is logged on to the network does not have automatic access to the database. Individual access is controlled through the granting of privileges. Users may be granted individual privileges, though they are normally allocated to groups and inherit the privilege levels of the group. Thus clerks in the sales office would log on with their own user names and passwords but would have the level of access granted to the sales group of users. Privileges are often granted in terms of user schemas but they can sometimes be very detailed, and may for example grant access to individual tables, or restrict the extent to which the user can add, delete or amend data. We will return to this theme in Topic 1.4 when we consider the SQL Data Control Language.

Many database management systems keep a record of transactions and it is possible to audit them. Regular random audits are carried out to identify suspicious or unusual activity, and irregularities can be investigated by a detailed analysis of the transaction record. In large, important databases there may be a continuous real-time audit with 'triggers' producing warnings or even blocking access to the system.

Encryption is a big subject and its detailed consideration is outside the scope of this unit, but it is likely to be a feature of most database systems since the end-user will usually have access over a network. It is particularly important if, as is increasingly common, access is over the Internet. On major systems, the security of packets of data will be ensured using the tunnelling facilities of virtual private networks or the emerging web services security protocols. But even small systems can provide for the security of data in transit using secure socket layers. (If you are unfamiliar with the public key infrastructure that underpins these technologies you might like to read the background paper Transaction_Security.pdf. [Link not available in this medium, please view it in the online course.])

A DBMS should always have a planned and documented backup and recovery system. Data should be backed up locally at frequent intervals, and backed up to secure remote storage at longer intervals. (What those intervals are depends on the amount and sensitivity of the data. It could be minutes and hours, or days and weeks.) Recovery is limited if the database can only be restored to its last backed up state. To safeguard the data fully it is necessary to log all transactions and to ensure that transaction logs are always available for the period since the last backup (or, to be safe, the last-but-one backup). The formulation and management of a logging and backup strategy is part of the Data Security Management Policy, which needs to set out what is done and who is responsible for doing it.

Most database management systems incorporate transaction processing. This is not the same thing as the keeping of transaction logs. Transaction processing is intended to provide for recovery from a problem arising as the database is being updated. It applies to many types of transactions but is best illustrated in an attempt to transfer money between accounts. If the transaction processing facilities of the DBMS are used the data changes resulting from the debit and credit are followed by a **commit** command which confirms both of them. If for some reason the whole transaction cannot be completed, the database is **rolled back** to the state it was in before the transaction was attempted.

© Copyright the University of Oxford and contributors.

1.8. Self-study exercises

Unit 3 Topic 1 exercise questions

We suggest you write your answers to the following questions in a word processor (or using pen and paper) before clicking the link below to see our suggested answers. If you disagree with what we say, please raise it with your tutor or (even better) start a discussion about it in the unit 3 forum.

Question 1. Write a few key words or phrases that suggest the problems encountered with data in an environment of process centred computing.

Question 2. Explain in two short phrases the characteristics of data in a process centred system which give rise to the above problems.

Question 3. Name the three types of schema found in a relational database. (You will no doubt have guessed that the actual answer to this question is not all we are looking for. It is a reminder to check that you are able to explain in broad terms why there are three schemas and how they achieve data independence).

Question 4. List the terms used to describe the types of people in the database environment, and as you list them check whether you would be able to describe their characteristics. Which of them may be unaware they are in fact using a database?

Question 5. What were the names of the two main data models used in database management systems before the introduction of the relational database? What is the generic term for these data models?

Question 6. Name three generic ways in which data in a system can be compromised.

Question 7. What are the broad principles of the UK data protection legislation? (In your own words and in any order).

Question 8. List the systems used in a database management system to secure the integrity of data.

Unit 3 Topic 1 exercise answers

Question 1.

There are many possibilities. Here are some that occurred to me. You may have a quite different list.

- Duplication of data
- Delete and update anomalies
- Inflexible
- No independent control
- No ad hoc queries
- Too many models

Question 2.

- The definition of data is embedded in the applications that use it.

© Copyright the University of Oxford and contributors.

- There is no independent way of accessing, controlling or validating the data.

Question 3.

- Storage Schema
- Logical Schema
- User Schema

Make sure you understand the mapping these schemas carry out between the model of the data seen by the user and the physical representation of the data backing store. The physical organisation of the data may change the logical model. The logical model itself may be extended without any impact on the view of data seen by users.

Question 4.

- Casual User
- Parametric User
- General User
- Database Administrator
- Database Programmer
- Data Administrator

The users who may not know they are using a database are the Casual User and the Parametric User.

Question 5.

- Hierarchical (or Tree) model
- Network (or CODASYL) model

The generic name for these models is Navigation.

Question 6.

There are many possible answers here. I would choose:

- Unauthorised use or disclosure of data
- Corruption of data
- Loss of data

Question 7.

- fairly obtained and used
- accurate, relevant and secure
- used only for stated purpose
- not kept unnecessarily
- not transferred abroad unless protected

Question 8.

© Copyright the University of Oxford and contributors.

Here again there is some scope for different answers but you should include the following points:

- Authentication of users
- Granting of privileges over views and actions
- Transaction audits
- Encryption of data in transit
- Backup and recovery

© Copyright the University of Oxford and contributors.

Unit 2. Relational theory

2.1. Introduction to relational theory

The **Relational Database Management System** (or **RDBMS**) has become the dominant approach to managing data. The relational approach has its limitations and a number of alternative approaches are under development, but it is likely that the RDBMS will remain dominant for many years. It is tried and tested and involves less risk (and often less expense) than using a more cutting-edge product. We take an overview of post-relational databases in Topic 6, but the main focus of this unit is on the theory and practice of the relational database.

In this topic we examine the concepts on which relational databases are founded and the common terminology and languages used to define them.

By the end of this topic you should:

- be able to describe the relational model
- know the definitions and functions of relational keys
- know how to use relational algebra to define relations and constraints

2.2. The relational database – background

The hierarchical and network models were embraced by the major vendors, most of which specialised in one or the other. IBM in particular did not support the CODASYL model and continued to develop its own product, IMS, which was a hierarchical database. One of the researchers at IBM was Edgar F. Codd (Ted Codd), and he became dissatisfied with the accepted view among database designers that the burden of finding information should be placed on users. In a series of internal papers he developed an idea of a better system; then in 1970 he published a landmark paper 'A Relational Model of Data for Large Shared Data Banks'. As a matter of historical interest you will be able to find a reprint of the paper on the Web.

The first prototype relational database (System R) was developed at IBM by Codd and his team, but it was not seen as an important development within the firm and it was not commercially exploited. Meanwhile, other teams had been working on the idea and the first commercial relational database was the Ingres RDBMS (since renamed Postgres), and later the Oracle RDBMS was developed by the Oracle Corporation. Eventually IBM also developed the relational database that was to become the DB2 series.

Apart from the advantages Codd had forecast for the relational database (arising from the decoupling of the data management from applications and users) most of the new database systems could run on mini or even micro computers compared with the mainframes on which most of the previous systems had been implemented.

You do not need to study Codd's paper but it is worth looking at its opening paragraphs, which set out the objectives of the model. For example, he says the relational model

© Copyright the University of Oxford and contributors.

> ' ... provides a means of describing data with its natural structure only – that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.'

Codd 1970, p1

Codd is remembered by a one-line summary of his theory, which has been repeated by countless students over the years:

> '... the key, the whole key and nothing but the key'

By the time you reach the end of this unit you will appreciate that whilst this is far from a sufficient summary of relational theory it does go right to the heart of it.

Database theory is about relations that have attributes, and instances of related attribute values, called tuples. We will examine these terms in the next section. In fact a relation is the theoretical equivalent of the table; an attribute the equivalent of the column; and a tuple the equivalent of the row. When you first study database theory it is often easier to understand something expressed in terms of tables, columns and rows, and even experienced practitioners frequently use the terms relation and table interchangeably. There are differences between a relation and a table, but we deal with those as they arise and in the mean time often use the more familiar terms even when discussing the abstract concept of the relation.

References:

Codd, E. F., 1970 'A Relational Model of Data for Large Shared Data Banks', *Communications of the Association for Computing Machinery*, June, available online at http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf, accessed January 2017.

2.3. Properties of relations

The relational database is based on the relation, which for most purposes can be considered a table containing **tuples** (rows) and **fields** or **attributes** (columns). The term **degree** is used to denote the number of attributes or columns in a relation, and the term **cardinality** is used to denote the number of tuples or rows in a relation.

Although it is often useful to think of a relation as a table, the relation is a more abstract concept and not all tables are relations. One important difference is that the order of the rows and columns in a relational database is not significant. In contrast with navigational databases, where it is meaningful to refer to an item by its position, in a relational database you cannot refer to an item as being in the 'third row' or the 'fourth column'. *Data is accessed only by reference to key fields in the data and the names of attributes.*

Another difference is that all values of an attribute (items in a column) must be of the same domain (broadly the same data type). Thus the tables in the following exercise are not relations:

© Copyright the University of Oxford and contributors.

Exercise

Stock_No	Item	Quantity
1	Wheel	24
2	Widget	Twenty

Why is this table not a relation? Answer

> Because the values in the Quantity column come from two different domains: a number type domain, and a character type domain.

In a relation, each cell must contain a single-value fact, so again the following table would not be a relation:

Stock_No	Item	Quantity
1	Wheel (small) Wheel (large)	24 12
2	Widget	Twenty

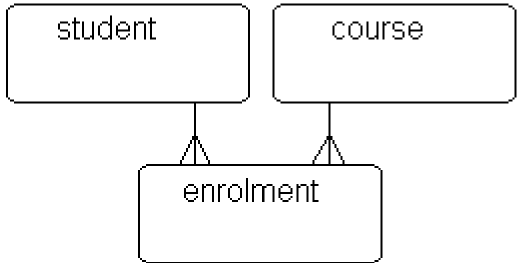
Why is this table not a relation? Answer

> Because the values in the Item and Quantity columns in row one contain multiple value facts (or are not atomic). For example, the item column contains the ‘fact’ of the small wheel and the ‘fact’ of the large wheel.

In relational theory the data in a relation is called the **extension** of the relation. In most circumstances the context makes it clear what we mean when we say relation, and the term extension is not much used in practice. We note its meaning here, but we will ignore it in the rest of the topic.

Relations usually contain organisational data, but sometimes it is necessary to introduce a new relation purely for the purposes of normalising the data. This is seen in the intersection entity pattern:

© Copyright the University of Oxford and contributors.



The enrolment relation may contain nothing more than the keys that link the tables 'student' and 'course'. This kind of relation is called a **key-only relation**, but it must nevertheless exhibit all the qualities required of a relation. (An intersection entity often contains organisational data, for example the module mark for a student or a swimmer's position in a race.)

Domains

A database will usually hold a variety of data types: a character array, a floating point, a Boolean, etc. Relational theory takes this further and requires values to belong to a **domain**. A domain is a set of values with a common meaning, from which one or more attributes in the database is typed. These domains may be no more than statements of the underlying data types, but they often impose further restrictions on the values in order to reflect some of the semantics of the data. For example a Social Security Number domain might prescribe that the length of the character array must be *n* characters. A works number domain might prescribe an integer within the range 1– 5000, or a character array within the range E0001 … E5000.

Domains are important when linking relations using key fields. Two relations may be linked using the fields EmployeeNumber and SalesAssistant if both attributes belong to the (for example) StaffNumber domain. Conversely EmployeeNumber in one relation could not be linked to EmployeeNumber in another, if one of them were typed to the StaffNumber domain and the other simply to the data type Integer.

The concept of the domain and the notation we use to represent domains are part of the abstract definition of the database. It may not be reflected in the way a particular database management system implements the design. It is intended to provide a complete and unambiguous definition of the database. It may or may not be possible to reflect the whole of the design in a particular implementation. (In fact there are few database management systems that can capture all the theoretical constraints used in defining domains.) The usual way to document domains is to list domain names together with their possible values, like this:

```
Domains
   EmployeeNumbers:     String A0001 ...A5000
   Regions:             Integer (1..9)
   LastName:            String
   Grade:               Character (A,B,G,S)
   JobDescription:      String(100)
```

The NULL domain

There is a special **null** domain to which attribute values may default. If a cell in a table is empty its value is in the null domain. The null value is not the same as zero (0) or an empty character string (""). The comparison A = B will answer true if the values of A and B are both 0 or are both "" but it will answer false if they are both null. Null can be thought of as standing for unknown, and it is often found in a database. For example, it is the value of the e-mail attribute where the person concerned does not possess an e-mail address.

© Copyright the University of Oxford and contributors.

If you ask a database to provide a table which includes only those rows in the student table where the value of the 'house' column is *not* 'red', you will get the rows where this column contains blue, green, yellow, etc. but you will not get the rows in which no entry has been made. (Note this is only true when *no entry has been made* in the column: if you make a text entry then delete it, in most databases the value becomes an empty string and this would be recognised as 'not red'.)

It is possible to override the default value for an attribute, for example number fields are sometimes set to default to zero.

Attributes

Attributes are the names of fields in a relation (the headings of the columns in a table). They play a crucial role in a relational database. Recall that position is not significant in a relation, so the following two different tables represent the *same* relation:

Table 1

Stock_no	Item	Quantity
1	Wheel	20
2	Widget	40

Table 2

Item	Stock_no	Quantity
Wheel	1	20
Widget	2	40

In order to extract an item from a relation it must be referred to by its attribute name, for example 'extract from Table 1 the contents of the Stock_No column for each row in which the value in the Quantity column is less than 20'.

An attribute must belong to a domain. In the tiny example above we might define the domains, relations and attributes as follows:

```
domains
    Stock_Reference: Integer 1..10000
    ItemDescriptor: String(100)
    InStock: Integer <5000

relation StockTable
    Stock_No: Stock_Reference
    Item: ItemDescriptor
    Quantity: InStock
```

© Copyright the University of Oxford and contributors.

Tuple

In the context of relational theory, a tuple is a collection of related attributes and values that represent a single instance of a relation. More simply, and just as accurately, it is a row of the relation. Relational theory requires that a tuple must be unique. It is the whole tuple that must be unique: values of some or most of the attributes can be duplicated, provided the whole tuple is unique.

View relation

The relations that actually hold the data are called **base** relations. Views (or **view** relations) are temporary tables derived from the base tables. A view table may be an aggregation of data from several base tables and, since it is temporary, it does not matter if data is duplicated in it.

We will look more closely at views later in the unit when we study relational algebra and Structured Query Language.

Pause for thought

From the discussion and illustrations in this section, list the characteristics of a relation.

Answer

- the order of the attributes (columns) is not significant;
- the order of the tuples (rows) is not significant;
- attributes must have a distinct name;
- all values of an attribute must be from the same domain;
- each cell of the relation holds a single value

2.4. Relational keys

The term **key** is used frequently when working with a relational database and it can mean different things depending on the context. You should normally avoid the generic term and use the more precise terminology described in this section.

Super key

We have seen that one of the characteristics of a tuple in a relation is that it must be unique. In order to refer to a particular tuple we need to provide enough attribute (or field) values to be able to identify it. A field, or combination of fields, whose values can uniquely identify a tuple, is referred to as a **super key** of the relation. There will usually be more than one super key. A value for every field in the relation must produce a unique row (since by definition in a tuple all rows are unique). In most cases there will be other super keys comprising the values of fewer fields.

© Copyright the University of Oxford and contributors.

Candidate key

A **candidate key** is a super key that is irreducible, that is, one that ceases to be a super key, if any of its component fields is removed. In the relation

```
Course(course_code, course_name, CAT_points)
```

all three attributes constitute a joint super key, but this combination of attributes does not constitute a candidate key because it is possible to reduce the key to one attribute (course_code) and still identify a unique tuple. Thus the candidate key can be defined as an attribute or combination of attributes of a tuple which:

- guarantees a unique tuple, and
- is not reducible to a key with fewer attributes.

In the following table there are several possible ways of identifying a particular row.

Works_No	Soc_Sec_No	Name	Phone
W1	AB123456C	Johnson	0123456
W2	ZZ987654B	Robson	0194738

As it stands we might think we could identify a row as the one where Name is Johnson, but of course that would not guarantee uniqueness because there may be a subsequent entry for another employee called Johnson. We could say the combined fields of Works_No and Soc_Sec_No identify a unique row, but that would not satisfy the criterion of irreducibility. So there are two candidate keys in this table, Works_No and Soc_Sec_No, each of which is a single-field key.

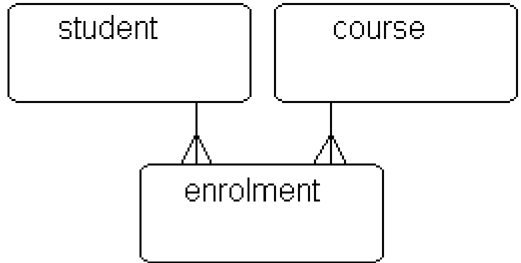
Look at the following table and identify any candidate keys:

Student_id	Course	Date
S1	Java	12/03/05
S2	Java	13/03/05
S1	Web Design	12/03/05

Here none of the attributes will alone identify a unique row and so the only candidate key to this relation is the key formed by the two fields Student_id and Course. This kind of key is called a **joint key**. Joint keys are often called composite or compound keys whatever their type. It rarely matters if we use the terms interchangeably, but strictly these are two distinct types of joint key:

© Copyright the University of Oxford and contributors.

The true **compound key** is composed of fields that are also single-field primary keys in related tables. For example, a candidate key of the enrolment table (probably the only one) is the combination of attributes whose values reference the primary keys of the student and course tables.



A **composite key** is a joint key where one or more of the fields comprising the key is *not* a primary key in a related table. An example would be a table recording students' attendance at optional counselling sessions. The joint key to this table would be the Student_id (a primary key in the student table) and the date (not a primary key in its own right.)

Once the candidate keys have been identified, one of them is 'promoted' to primary key and the rest become alternate keys. Both of these terms are discussed further below.

Pause for thought

Identify the candidate keys in the following relations:

Employee table

Works_No	Soc_Sec_ No	Name	Phone
W1	AB123456C	Johnson	0123456
W2	ZZ987654B	Robson	0194738

Answer

The Name and Phone columns cannot be keys because they may not be unique. Works_No and Soc_Sec_No are both individually candidate keys because each of them can guarantee uniqueness. The combined fields (Works_No, Soc_Sec_No) cannot be a key because it is reducible.

Gala table

Member_No	Race_No	Stroke	Distance	Position	Time

© Copyright the University of Oxford and contributors.

M1	R1	Freestyle	100	1	1.59
M2	R2	Backstroke	50	3	1.01
M1	R2	Backstroke	50	1	0.58
M3	R3	Backstroke	100	4	3.03

Answer

In the Gala table above no single field can guarantee uniqueness. The only candidate key here is the joint key comprising the fields Member_No and Race_No. (It would be possible to guarantee uniqueness by using more than two columns, but that would not produce a candidate key because it would fail the irreducibility test.)

Primary key

The **primary key** is the candidate key that is chosen to be used consistently to identify tuples in the relation. Thereafter any definitions, operations or links that require a key to the relation will use the primary key. The choice of key is not crucial and in many cases any of the candidate keys will produce a suitable primary key. There are, however, sometimes reasons to choose one candidate key over another. One reason is to simplify the control of redundancy (which we saw in Topic 1 was one of the motivating factors for the development of the database approach). Redundancy cannot be eliminated but in a relational database it is controlled, and the most common redundant items are the values of primary keys (because these values are used to link tables). There is no fundamental reason why a long key cannot be used as the primary key, but the use of a short key aids clarity in the design, and in some database environments makes it easier to enforce referential integrity (which we discuss later). Hence, in the following table, assuming we can guarantee that two courses will never have the same name, there are two candidate keys, Course_Code and Course_Name, but the better choice for primary key would be Course_Code. It is less likely to change and has a short value that is easy to handle.

Course_Code	**Course_Name**	**Units**
C100	Introduction to Web Design	30
C200	Programming Networked Applications using Web Services	30

Another consideration when choosing the primary key is the possibility that one of the prima facie candidate keys may in fact turn out *not* to be a candidate key. In the Employee table above we have assumed that the Soc_Sec_No will be unique. That is a fair assumption if it is present, but if it is not present it cannot be used to

© Copyright the University of Oxford and contributors.

identify anything. It may be, in the particular circumstances, that details of prospective employees need to be entered in the table even if their Social Security numbers are not known. The possibility of a null value disqualifies a field (or group of fields) as a candidate key and therefore as a primary key.

Alternate key

An **alternate key** is a candidate key that has not been chosen as the primary key. It can, by definition, identify a unique tuple, but it is normally not used for that purpose (which is now the role of the primary key).

You will sometimes find the term candidate key misused to describe a key, such as the Soc_Sec_No field, which might in fact contain a null value, but which when present is unique. This is incorrect use of the terminology (since a candidate key value can never be null) but it reflects the potential importance of such pseudo-alternate keys in enforcing entity integrity (which we will examine shortly).

Surrogate key

A **surrogate key** is one that is not found within the real world data but is generated within the database as an auto increment or auto number field.

Some data-sets do not possess a natural primary key, or they have a key that is long and unwieldy. Consider this data-set:

LastName	FirstNames	House
Johnson	Bill	Red
King	Sue	Blue

As it stands this data-set cannot be held in a relation because there might very well be two people called Sue King in the blue house but it would not be possible to identify one of them uniquely. In these circumstances most database management systems allow us to use an internally generated auto increment or auto number facility to generate a primary key.

Person_id	LastName	FirstNames	House
1	Johnson	Bill	Red
2	King	Sue	Blue

This kind of key, which is not found in the data itself, is called a surrogate key. A surrogate key is sometimes used even when the natural data contains a primary key, in order to avoid the use of a multi-field primary key. For example, in the following table the only natural primary key is (LastName, FirstNames, Address, Date_of_Birth):

LastName	FirstNames	Address	Date_of_Birth	House

© Copyright the University of Oxford and contributors.

Jones	John	1 The Green	23/4/1941	Red
Jones	John	1 The Green	15/11/1975	Red

We must have all these fields because there could be father and son with the same name, or another John Jones with the same date of birth at a different address.

In these circumstances a surrogate key might be chosen so that links between relations could be based on the small and simple auto number primary key, rather than the four-field natural primary key.

An auto number or auto increment type is available in most relational databases.

Foreign key

The data manipulation facilities of a relational database involve joining relations to produce temporary views of more useful tables of data. Here are three relations from a college database:

Student

Stud_id	Name
S1	Jones
S2	Brown
S3	Patel

Enrolment

Student	Course
S1	C1
S2	C2
S3	C3

Course

Course_id	Name
C1	Java
C2	HCI
C3	Web

The relations can be linked to provide a view table showing the names of courses on which a student is enrolled. This is achieved by linking the Student and Course attributes of the Enrolment relation with the primary keys of the other two tables. (They do not need to have the same attribute name. What matters is that they belong to the same domains.) Where an attribute contains values that correspond to values of a primary key in another table, it is called a **foreign key**. In the above tables the Stud_id and Course_id attributes are primary keys. The Student and Course attributes in the Enrolment table are foreign keys.

Pause for thought

We have identified two foreign keys in the enrolment table. What is the primary key of the enrolment table?

Answer

> The only candidate key in this table is the joint key of student and course, and so that is the primary key.

© Copyright the University of Oxford and contributors.

2.5. Database integrity and constraints

The characteristics of the relations outlined in the previous sections are guaranteed partly by the built-in structures and operations of the database management system, but also by constraints built into the database definition. The areas of database integrity that need to be considered are introduced here, and they are developed further in Topic 3 where we examine the formal specification or definition of a database.

Domain integrity

A database is said to have **domain integrity** when no values fall outside the domains of the database. This is ensured by building **domain constraints** into the database definition. These constraints are also needed to facilitate the relational algebra (and later SQL) expressions that link relations. Enforcing domain integrity involves specifying the values that an attribute can hold. It may be as simple as restricting the values to a data type, but more often it also involves ensuring that attribute values are within a valid range. Domain constraints often prescribe the size of a character-string data type, the minimum/maximum values of a number type, or the restriction of values to a predetermined list, for example a Property domain may be restricted to 'Flat', 'House' or 'Bungalow'.

A common domain constraint is the prohibition of null values. A simple example is the domain governing the name fields in a relation holding data about people. It is not meaningful to hold a record of data about a person without at least name details, so the domain on which person fields are based would be defined as not null.

Where possible, domain constraints are included in the domain definition, but there are times when this cannot be done. Sometimes a constraint does not apply to the whole domain but only to a particular attribute. In the following example all students are allocated to a house on registration but only selected tutors belong to houses:

Student Table

stud_id	name	house
1	John	Red

Tutor Table

tut_id	name	house
10	Sue	

House Table

house	day
Red	Mon

Although the house attributes in the Student and Tutor tables belong to the same domain, only one of them should be constrained as not null. Such closely targeted constraints are declared in the attribute section of the database definition, although they are still described as domain constraints.

Enterprise constraints are similar to domain constraints; indeed the border between the two concepts is fuzzy. Enterprise constraints impose business rules particular to the business function supported. They may be so fundamental or of such universal application that they are imposed as domain constraints in the database definition. For example a Date domain may be restricted not only by its data type but also by prescribed earliest and/or latest possible dates. Much depends on the nature of the business rules and the extent to which it is anticipated the database will be shared, but it is necessary to exercise care in enforcing business rules using domain constraints because they may restrict the flexibility of the database. Some business rules are better imposed in the business layer of the applications programs, leaving the database open to be used for other, perhaps yet unknown, purposes.

© Copyright the University of Oxford and contributors.

Entity integrity

Entity integrity ensures that tuples are unique and can be identified. This is usually imposed through the declaration of a primary key, which by definition must identify a unique tuple in the relation. We say usually because that is only true where the primary key comprises attributes in the natural data, for example a Social Security number or some combination of attributes like Date, Time and Patient that guarantees uniqueness in the real world. If a surrogate key is used (for example an auto number), the entity integrity is purely theoretical because the same set of data could be entered successively but have different primary keys. In these circumstances additional steps are needed to secure entity integrity. It may be possible to achieve entity integrity in practice, if not in theory, by imposing a constraint on a field that fails the candidate key test only because it is allowed to contain a null value.

The National Insurance or Social Security number is again a good example. Although it may be not a candidate key (because it could be null) the nulls are not likely to last long and where there is an entry it will be unique. In these circumstances entity integrity is in practice achieved by prohibiting duplicates (other than null) in the Social Security field. In the last resort, if it is impossible to achieve entity integrity in the defined database it must be flagged up so that the problem can be addressed in the application programs.

Referential integrity

Referential integrity requires all foreign keys to hold values that exist as primary keys in the appropriate link relations. In the table below, the Enrolment relation violates referential integrity because there is a value in the Enrolment.Course_id field (the foreign key) which does not appear in the Course.Course_id field (the primary key).

Student		
Student_id	Name	phone
S1	Jim	0234
S2	Sue	0235
S23	Joe	0456

Enrolment	
Student_id	Course_id
S1	C1
S1	C2
S23	C3

Course	
Course_id	Name
C1	Java
C2	Pascal

Referential Integrity Violated

(Incidentally, the above explanation illustrates how, when talking about two or more attributes with the same name, each attribute is referred to with its qualified name, comprising the **table name**, a **period** and the name of the **attribute**.)

The database management system controls the insertion, deletion and amendment of values to ensure there is no loss of relational integrity. An attempt to insert an invalid value in the foreign key field will fail, as will an attempt to delete or amend a primary key whose value exists in a foreign key field. In most database management systems it is possible to modify this behaviour so that instead of prohibiting the deletion or amendment of 'linked' primary keys, the system cascades the deletion or amendment to all instances of the value in foreign keys.

2.6. Introduction to relational algebra

Relational algebra (RA) is the language used to define the operations of a relational database. The relational model is founded in mathematics and is defined in terms of mathematical symbols and operators. It is based in set theory and if you have some background in this branch of mathematics you will readily appreciate what

© Copyright the University of Oxford and contributors.

follows. But you do not need mathematical knowledge or aptitude because the propositions and operations of relational algebra can be expressed in something very near ordinary language. Apart from noting what they are, we shall not be using mathematical symbols, but will instead define relational expressions using key words and the names of relations and attributes.

A relational expression acts on one or more relations producing a result that is a new relation whilst leaving the original relations unchanged. In everyday terms a simple relational expression might be 'Select all the rows from the Books table where the author is Graham Greene and put those rows in a new table called GreeneBooks'. The operation would leave the original table (Books) as it was but would produce a new table (GreeneBooks) containing only the required rows. But we do not use such everyday language. Databases are defined using relational algebra in order to ensure the definition is precise, unambiguous and in a common format.

Pause for thought

This time not a question for you, but one we suspect you may want to ask. Why bother with relational algebra when we need to use SQL anyway to implement the design. Would it not be simpler to use SQL to define the database?

There may be no altogether satisfactory answer to this question, but we can think of the following reasons for using relational algebra:

- At the point of design we might still not have decided the precise DBMS and so may not know the dialect and scope of the SQL we will use.
- The definition sets out what we want rather than how we will achieve it. Some constraints in RA may not be capable of implementation in SQL and these may need flagging up for the attention of application developers and users.

Assignment of the result of a relational algebra operation can be signified in three ways:

```
[relational algebra expression] giving newRelationName

newRelationName <--- [relational algebra expression]

newRelationName = [relational algebra expression]
```

All are commonly used but in this unit we will use a 'giving' clause to denote the new table. It is often necessary to use a series of expressions to obtain (or define) the required result, and in these circumstances a series of named intermediate tables may be used. On the other hand, since relational algebra has the property of closure it is possible to chain and nest expressions and when you gain more experience you will tend to avoid naming the intermediate tables. To begin with, however, we will name all intermediate tables. Where the operator acts on a single table it is called a unary operator. There are two unary operators: project and select.

The project operator

The **project** operator can be understood as a vertical slicing of a table. It produces a new table containing only the columns specified in the expression. Here is an example of the syntax and action of the project operator.

© Copyright the University of Oxford and contributors.

```
project Widgets over [name, price] giving NameAndPrice
```

Widgets		
id	name	price
1	ortho	60
2	retro	75
3	normal	50
4	micro	65

NameAndPrice	
name	price
ortho	60
retro	75
normal	50
micro	65

This is what you would expect, but consider this next expression and its result:

```
project Cars over [make, colour] giving MakeAndColour
```

Cars			
ID	make	model	colour
1	Ford	Mondeo	Red
2	Ford	Granada	Blue
3	Ford	Escort	Blue
4	Ford	Mondeo	Blue

MakeAndColour	
make	colour
Ford	Red
Ford	Blue

You might have been expecting the new table to have four rows. If so, we are partly responsible for that expectation because we have been using the word 'table' when we really meant 'relation'. Relational algebra deals in relations and though it often helps understanding to use terms like 'table', 'row' and 'column' we must not forget that a relation is a special kind of table. A relation is a set of tuples in which, by definition, each tuple must be unique and so in the resultant table all duplicate rows are removed.

The select operator

The **select** operator , can be understood as a horizontal slicing of the table. It produces a new table containing all the columns of the original table, but only the rows specified in the expression. Here is an example of the syntax and action of the select operator:

```
select Cars where colour='Blue' giving BlueCars
```

Cars			
ID	make	model	colour
1	Ford	Mondeo	Red
2	Ford	Granada	Blue
3	Ford	Escort	Blue
4	Ford	Mondeo	Blue

BlueCars			
ID	make	model	colour
2	Ford	Granada	Blue
3	Ford	Escort	Blue
4	Ford	Mondeo	Blue

Having created BlueCars by that operation we can now use it as the subject of further expressions

```
project BlueCars over [model,colour] giving BlueModels
```

```
project BlueCars over [make,colour] giving BlueFords
```

© Copyright the University of Oxford and contributors.

would give

BlueModels	
model	**colour**
Granada	Blue
Escort	Blue
Mondeo	Blue

BlueFords	
make	**colour**
Ford	Blue

The product operator

The first of the binary operators we shall look at is the **Cartesian product**. (This is the mathematical term for the concept: the relational algebra keyword is just **product**.) The Cartesian product is a concatenation of each row of the first table with every row of the second table. The syntax is TableA *product* TableB *giving* TableC. Applying the product operator to the following tables:

Room		
room_id	**phone**	**it**
101	01234	yes
200	019845	no

Tutor		
tutor_id	**name**	**room_id**
1	John	101
2	Sue	200
3	Mike	101

```
Room product Tutor giving TutorRoom
```

produces

TutorRoom					
room_id	**phone**	**it**	**tutor_id**	**name**	**room_id**
101	01234	yes	1	John	101
101	01234	yes	2	Sue	200
101	01234	yes	3	MIke	101
200	019845	yes	1	John	101
200	019845	yes	2	Sue	200
200	019845	yes	3	MIke	101

Each row in the Room table is joined to each row in the Tutor table. Note that although there are many rows containing several columns of the same data, in no case is there a duplication of a whole row of values, so the resulting table is a relation. The product operator is important for its theoretical underpinning of relational algebra rather than for any practical usefulness.

The inner join operators

The binary operators you are more likely to use are the **inner join** operators. There are several varieties of the join operator, some of which we will only summarise here (though we will look at the same concepts later when we examine Structured Query Language). An inner join operation joins the tables by reference to attributes sharing a common domain, then combines the two attributes in one field of the result. This contrasts with the product operation. In the product example above both attributes sharing the common domain (room_id and room_id) are shown in the result table. In a join operation those attributes would be combined into one field (the first room_id) in the new table.

© Copyright the University of Oxford and contributors.

The most common join operators are the natural join and the theta join. The syntax of the **natural join** expression is Table1 *join* Table2 *giving* Table3. The natural join is normally appropriate when the two tables to be joined have in common one (and only one) attribute from the same domain. In the tables below, the room_id attribute in each table is from the same domain and they are the only attributes that share the same domain.

Room		
room_id	phone	it
101	01234	yes
200	019845	no

Tutor		
tutor_id	name	room_id
1	John	101
2	Sue	200
3	Mike	101

```
Room join Tutor giving TutorRoom2
```

TutorRoom2

room_id	Phone	it	tutor_id	name
101	01234	yes	1	John
101	01234	yes	3	Mike
200	019845	no	2	Sue

The natural join operator is derived from the product and select operators. You need not learn this, but as a matter of interest you might like to demonstrate the derivation by carrying out the following operations on the Room and Tutor tables shown above:

```
Tutor product Room giving TutorRoom
select TutorRoom where Room.room_id=Tutor.room_id giving tempTable
project tempTable over[Room.room_id,phone,it,tutor_id,name]giving TutorRoom2
```

Answer

```
Tutor product Room giving TutorRoom
```

TutorRoom

room_id	phone	it	tutor_id	name	room_id
101	01234	yes	1	John	101
101	01234	yes	2	Sue	200
101	01234	yes	3	Mike	101
200	019845	yes	1	John	101

© Copyright the University of Oxford and contributors.

200	019845	yes	2	Sue	200
200	019845	yes	3	Mike	101

```
select TutorRoom where room_id=room_id giving tempTable
```

tempTable

room_id	phone	it	tutor_id	name	room_id
101	01234	yes	1	John	101
101	01234	yes	3	Mike	101
200	019845	yes	2	Sue	200

```
project tempTable over[room_id,phone,it,tutor_id,name]giving TutorRoom2
```

TutorRoom2

room_id	phone	it	tutor_id	name
101	01234	yes	1	John
101	01234	yes	3	Mike
200	019845	no	2	Sue

Join operations are associative, that is, (Table1 *join* Table2) *join* Table 3 is the same relation as Table1 *join* (Table2 *join* Table3).

The **theta join** is similar to the natural join but it includes details of the keys to be used to join the tables. It is often preferred to the natural join because it is immediately clear which attributes belong to the common domain. It can also be used where there are several common attribute domains in the tables. The syntax is:

```
Table1 join Table2 over CommonDomain.
```

You may also see it expressed as …

```
join Table1,Table2 over CommonDomain.
```

The above natural join illustrated above could have been expressed as a theta join like this:

```
Room join Tutor over room_id
```

© Copyright the University of Oxford and contributors.

Where you need to join more than one table the operation can be carried out in stages using temporary intermediate tables, or the clauses can be aggregated, for example:

```
(Course join Tutor over course_id) join Room over room_id
```

If the attribute names are not the same in the linked tables, you need to show the full tableName.attributeName combinations, for example:

```
(Course join Tutor over Course.course_id=Tutor.Course)
join Room over Room.room_id= Tutor.staffRoom
```

Exercise

What relational algebra expressions are needed to join the following tables so that the result contains details of the student name and course name for all enrolments?

Student Table	
Student_id	Name
s1	Johnson
s2	Bates

Enrolment Table	
Student	Course
s2	c1
s1	c2
s1	c2

Course Table	
Course_id	Name
c1	Java
c2	Web

Answer

There are two possibilities:

1. Using an intermediate relation:

```
Student join Enrolment over Student_id=Student giving A
A join Course over Course=Course_id
```

2. Using a compound expression

```
(Student join Enrolment over Student_id=Student) join Course over Course=Course_id
```

The outer join

The natural and theta joins include in the result only those rows in which the relevant value is present in *both* tables. If we want to include all the rows of one table but only selected rows of the other we need to use an **outer join**. Assume that the college has (unusually in our experience) a surfeit of staff rooms and that some are presently unoccupied:

© Copyright the University of Oxford and contributors.

Room		
room_id	phone	it
101	01234	yes
200	019845	no
201	02834	yes

Tutor		
tutor_id	name	room_id
1	John	101
2	Sue	200
3	Mike	101

We might want to define a relation that shows the rooms tutors occupy but which also shows vacant rooms. We define the required relation using a left outer join:

```
Room left outer join Tutor over room_id giving RoomAvailability
```

RoomAvailability

room_id	phone	it	Tutor_id	name
101	01234	yes	1	John
101	01234	yes	3	Mike
200	019845	no	2	Sue
201	02835	yes		

The right outer join has the same effect except that it works the other way round (including all the rows in the second table but only the linked rows from the first).

Defining views

Views are an essential part of a relational database. The contents of the tables in the database often need to be partitioned depending on the user. An example we considered earlier in the unit is the use of the data in an Employee table. Sales staff need to access the Employee data in order to access things like department and phone number but they should not be able to see tax code or performance details. This is achieved by defining a View, which the Database Administrator will grant to sales staff. I hope you can by now work out the relevant relational algebra operator needed here.

Answer

> It is project, for example:project employee over emp_id,lastname,firstnames, dept, phone giving salesempview.

Another kind of partitioning is used to provide local tables in large organisations. For example the area manager should have access to all fields of the Employee table but only for employees who work in her area. Which operator is relevant here?

Answer

© Copyright the University of Oxford and contributors.

It is select, for example select from Employee where area=OX giving OxfordEmpView.

An application or general user will often need to construct a view based on joins between a number of tables together with a series of project and select operations. Recall that one of the strengths of the database approach is that ad hoc views can be made at any time provided the user has the appropriate level of access to the tables. In many cases, however, these complex views are not ad hoc but commonly used queries which arise time and again. When it is clear at design time that they will be needed the relevant views should be included in the definition of the database and implemented when it is created. Users can then simply refer to the view by name instead of having to construct a complex series of commands each time they need it. Users of a college database would frequently need a view that displays the names of students together with the names of courses on which they are enrolled. How would the view be defined, assuming these base relations?

Student			
Student_id	L_Name	F_names	Phone
1	Smith	John	019465
2	Brown	Sue	019384
3	Green	John	019564

Enrolment	
Student_id	Course_id
1	1
1	2
2	3
3	3
3	2

Course		
Course_id	Name	Units
1	DB	30
2	Java	60
3	Pascal	30

Answer

```
(Student join Enrolment over Stud_id) join Course over Course_id giving StudentCourses
```

If you wanted to include all students. including those who had not yet enrolled on any courses, you would use an outer join:

```
((Student left outer join Enrolment over Stud_id) join Course over Course_id giving
StudentCourses)
```

Another reason for using views is to shield users from changes in the logical database. If an application uses the base tables it could run into difficulty should their structure be changed (for example to cope with a new application). A view can remain constant even though changes are made to the base tables.

2.7. Relational algebra – 2

Relations are based on the mathematical concept of sets and they support set operations including the union, intersection and difference operators. These operators do, however, require the attributes of the two relations to have the same domains. You may think it unlikely that you would have two relations with exactly the same number of columns and data types, but remember that relational algebra has the property of closure and can

string together expressions. You may very well precede a union or intersection operation with project operations that produce the required relations. There may be a considerable difference between the Student and Tutor tables but you can use project statements to achieve the following relations from them:

S	
name	email
John	j@ohn.com
Sue	sue@home
Bill	bill@quay

T	
name	email
Trevor	trev@work
Joan	jo@home
Mike	mik@coll

Union

The **union** operator produces a relation that has the same attributes as the original relations but contains the tuples of both. In effect the union operator simply combines the tuples from the tables. Here is an illustration of S union T giving All:

S	
name	email
John	j@ohn.com
Sue	sue@home
Bill	bill@quay

union

T	
name	email
Trevor	trev@work
Joan	jo@home
Mike	mik@coll

giving

All	
name	email
John	j@ohn.com
Sue	sue@home
Bill	bill@quay
Trevor	trev@work
Joan	jo@home
Mike	mik@coll

I use the word tuple rather than row here as a reminder that we are still strictly in the realm of relations rather than tables. If a tuple in one relation is identical with a tuple in the other relation, only one copy of it will appear in the new relation. Using the following intermediate relations only one of the tuples, Bill 04/08/1980, will appear in the All relation.

S	
name	dob
John	12/04/1986
Sue	03/05/1979
Bill	04/08/1980

union

T	
name	dob
Trevor	03/09/1978
Sue	04/08/1985
Bill	04/08/1980

giving

All	
name	dob
John	12/04/1986
Sue	03/05/1979
Bill	04/08/1980
Trevor	03/09/1978
Sue	04/08/1985

Intersection

The **intersection** operation results in a new table containing only the tuples that appear in *both* of the source tables:

© Copyright the University of Oxford and contributors.

S	
name	dob
John	12/04/1986
Sue	03/05/1979
Bill	04/08/1980

intersection

T	
name	dob
Trevor	03/09/1978
Sue	04/08/1985
Bill	04/08/1980

giving

All	
name	dob
Bill	04/08/1980

The intersection operator is commonly used in defining the semantics of data or imposing enterprise constraints. Suppose in this college a person could not be both a student and a tutor. The constraint definition might include:

```
S intersection T is empty
```

Difference

The difference operation produces a relation containing only those tuples in the first relation that are not present in the second relation:

S	
name	dob
John	12/04/1986
Sue	03/05/1979
Bill	04/08/1980

difference

T	
name	dob
Trevor	03/09/1978
Sue	04/08/1985
Bill	04/08/1980

giving

All	
name	dob
John	12/04/1986
Sue	03/05/1979

Divide

The relational algebra operations we have considered so far have a close counterpart in Structured Query Language expressions, and this simplifies the process of implementing the design. Division is an exception, and for this reason it is rarely used in defining a database; combinations of other operations are used to achieve the same result. We do not use division in this unit, but for completeness we include a brief description of it.

The **divide** operator produces the records in one record set whose values match all the corresponding values in a second record set. Here is an example. Assume that we have already projected the Product table over the id field so that we have a single-field table:

divide

Ordr Items	
order_id	product_id
1	2
1	1
2	2
3	1
4	1
4	2

by

Product
product_id
1
2

giving

Bought both
order_id
1
4

If you have difficulty with the divide operator, leave it for the moment. You will not meet it again in the units or the assignment.

© Copyright the University of Oxford and contributors.

Defining constraints

Many constraints are defined in the structural definition of the database, through domain and entity declarations, but others can only be defined using relational algebra. In the following relations, the structural part of the definition of the relation would make it clear that the mentor_id in the Student table is a foreign key which references the primary key (tut_id) of the Tutor table.

Tutor		
tut_id	name	grade
1	Jim	J
2	Sue	S
3	Mary	J

Student		
stud_id	name	mentor_id
1	Mary	2
2	Jim	2
3	Bob	1

The structural part could also impose a constraint to reflect the real world requirement that all students must have a mentor.

Exercise

How would this constraint (all students must have a mentor) be documented in the structural part of the definition?

Answer

By defining the Mentor_id attribute in the Student table as *not null*.

But there may be other constraints that cannot be captured in the structural definition. We may know, perhaps from a narrative part of the data model, that the mentor must be a senior tutor. This kind of constraint is added at the end of the structural part of the relation definition and consists of one or more relational algebra expressions. For example, here we would include:

```
project Student over mentor_id giving A
select from Tutor where grade = S giving B
project B over tut_id giving C
A difference C is empty
```

Relational algebra is also required to define exclusive relationships. It is difficult to produce illustrations which are at the same time realistic and compact, so please do not try to follow through any other meanings of the following relations:

Page	
page_id	name
1	welcome
2	list of ..
3	Clubhouse

Text		
text_id	text	page_id
1	abcd	1
2	efgh	1
3	and more	3

Graphical		
image_id	filename	page_id
1	club.jpg	3

© Copyright the University of Oxford and contributors.

The Page relation is a projection of a larger relation that records information about pages in a document or site. The blocks of content of the pages are kept in the other relations. For some reason a page can only contain either text or graphical material but not both. This constraint can be defined as:

```
project Text over page_id giving A
project Graphical over page_id giving B
A intersection B is empty
```

2.8. Summary of relational operators

Relational algebra is a mathematical approach to the definition and manipulation of relations and its ultimate expression is in standardised mathematical formulae. We set out some of the formulae here, as background information, but in this course we do not use it. There are, we believe, good reasons why we should here prefer a narrative version of the mathematics:

- It makes it easier to read the constraints included in our definitions of relations.
- It helps to relate the theory of relational algebra to the practice of using Structured Query Language (which we cover in Topic 4).
- There is, in practice, no loss of precision by using words instead of symbols to denote the relational operators.

Though the mathematical expressions are standardised there is some variation in the way narrative versions of them are expressed. You may find other sources that use different conventions, but the meaning is usually quite clear. For example, the expression

```
Table A join Table B over Student_id
```

is often expressed

```
join Table A, Table B over Student_id
```

The summaries here are intended as reminders of the syntax. For examples of their use see Sections 2.6 and 2.7 of this unit.

Unary operators

Project

Project slices a relation vertically, retaining the same number of tuples but restricting the number of attributes.

```
Project TableName over col1,col2,col3
```

$\Pi_{\text{col 1, col 2, col n}}$ (TableName)

Select

Select slices the relation horizontally, retaining the same number of attributes but restricting the rows according to the logical conditions supplied.

© Copyright the University of Oxford and contributors.

```
Select TableName where (condition)
```

$\sigma_{condition}$ (TableName)

Product and join operators

Product

Product performs a Cartesian product on the relations. It joins every row in the second relation to each row in the first relation.

```
Table1 product Table 2
```

Table 1 × Table 2

Natural join

A simple join over a common attribute with the same attribute name. The new relation includes only those instances where the relevant attribute has the same value in both of the source relations.

```
Table 1 join Table 2
```

Table 1 ⋈ Table 2

Theta join

Similar to the natural join but the common attribute(s) are specified in the expression:

```
Table 1 join Table 2 over AttributeName
```

or

```
Table 1 join Table 2 over Table1.AttributeName=Table2.AttributeName
```

Table 1 $\bowtie_F$ Table 2

(**F** in this equation is the function, in effect the 'over' clause from the verbose version of the expression.)

Outer join

An outer join creates a new table containing the attributes of both tables. It includes *all* tuples from one of the tables, but *only* those tuples that have a common attribute value from the other table. (The tuples from the 'other' table, where there is no common attribute value, are set to null in the new table.)

```
Table 1 left outer join Table 2 over Student_id
```

Table 1 $?_F$ Table 2

(The right outer join symbol has the horizontal lines at the right.)

© Copyright the University of Oxford and contributors.

Set operators

Set operations require the attributes of the relations to have the same domains, and thus are usually preceded by projection operations.

Union

The union operation produces a relation that has the same attributes as the original relations but it contains the tuples of both.

```
Table 1 union Table 2
```

Table 1 ∪ Table 2

Intersection

The intersection operation results in a new relation containing only the tuples that appear in both of the source relations.

```
Table 1 intersection Table 2
```

Table 1 ⋂ Table 2

Difference

The difference operation produces a relation containing only those tuples in the first relation that are *not* present in the second relation.

```
Table 1 difference Table 2
```

Table 1 - Table 2

Divide

The divide operator produces the records in one record set that have values that match all the corresponding values in a second record set.

```
Table 1 divide Table 2
```

Table 1 ÷ Table 2

2.9. Codd's twelve principles

In magazine articles in 1985 (Codd 1985a, 1985b), Ted Codd set out twelve rules that a database management system had to obey before it could be properly termed a relational DBMS. Most of these rules reflect what we have already studied in this topic, but they are useful as a summary of the characteristics of the relational model.

1. The information rule.
Data are represented in one way only, as values within columns within rows.

© Copyright the University of Oxford and contributors.

2. Guaranteed access.
Every value can be accessed by providing nothing more than table name, column name and key.

3. Systematic treatment of null values.
Empty records must be distinguished from zero values and empty strings throughout the system.

4. Dynamic relational online catalogue.
The structure of a database must be stored in tables within the database and be capable of being accessed by authorised users as they would any other tables.

5. Comprehensive data language.
There must be a comprehensive language for data definition, manipulation and control. (These days, this means it must implement ISO SQL.)

6. View must be capable of updating.
Some views are inherently not updateable. (Views that lack candidate keys are permissible in many databases.) But where it is theoretically possible to update a table through a view, it must in fact be possible.

7. Must support set-at-a-time updates.
It must be possible to update several rows with one command. (This rule will become clearer later when we have studied the SQL Update command.)

8. Physical data independence.
User programs are not dependent on the physical structure of the database. It must be possible to change the physical organisation of the data without affecting users.

9. Logical data independence.
User programs are independent of the logical structure of the database. It must be possible for the logical structure of the database to evolve without any impact on user programs. For example, the addition of new columns in a table should not affect existing users of the table. (Complying with this rule is easier if all users use views in a user schema. Then even more wide-reaching changes to the logical structure can be shielded from the users who can retain the same views.)

10. Integrity independence.
Integrity constraints should be stored in the database and be available for inspection in the database catalogue (as opposed to being implemented in user programs.)

11.Distribution independence.
We have not yet looked at the distributed database. You can have situations in which parts of the same database are in different physical locations. Any such distribution should be hidden from the user. The user should see the database as a single entity.

12. Non-subversion Rule.
If the DBMS provides additional updating features, beyond those provided by SQL, those features must not be capable of violating the integrity of the database. There should be no back door through which users can bypass the constraints imposed by the DBMS.

In later years Codd added some further rules but they are mainly on esoteric matters. One of these additional rules, often called 'Rule Zero', is, however, worth noting. It provides that a system that calls itself relational but provides additional features (such as object-oriented features) must remain capable of being used as a purely relational database.

© Copyright the University of Oxford and contributors.

References:

Codd, E. F., 1985a 'Is Your DBMS Really Relational?', *ComputerWorld*, October 14.

Codd, E. F., 1985b, 'Does Your DBMS Run By the Rules?', *ComputerWorld*, October 21.

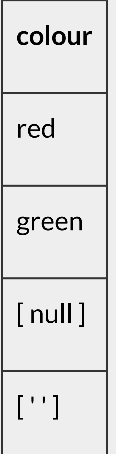
2.10. Self-study exercises

Unit 3 Topic 2 exercise questions

We suggest you write your answers to the following questions in a word processor (or using pen and paper) before clicking the link below to see our suggested answers. If you disagree with what we say, please raise it with your tutor or (even better) start a discussion about it in the unit 3 forum.

Question 1. List the fundamental characteristics of a relation.

Question 2. This is a projection (giving A) of a relation over the colour attribute (with blank cells showing their true state):



colour
red
green
[null]
['']

How many rows would be in the relation produced as a result of:

```
SELECT from A where colour NOT 'green'
```

Question 3. List five types of key likely to be mentioned in the documentation of a relational database.

There is also a key you are unlikely to see documented (the super key) so do not include that in your answer.

Question 4. What are the two fundamental characteristics of a candidate key?

Question 5. Write three short sentences describing the types of integrity we need to address when designing a relational database.

© Copyright the University of Oxford and contributors.

Student			
Student_id	L_Name	F_names	Phone
1	Smith	John	019465
2	Brown	Sue	019384
3	Green	John	019564

Enrolment	
Student_id	Course_id
1	1
1	2
2	3
3	3
3	2

Course		
Course_id	Name	Units
1	DB	30
2	Java	60
3	Pascal	30

Three tables, Student, Enrolment and Course. Each tuple of the Enrolment table contains a Student_id and a Course_id. These are foreign keys referencing attributes of the same name that are primary keys in the outer tables.

Question 6. Given the above relations, how would you define in relational algebra a relation that lists the names of students who have enrolled on the Java course.

Tutor		
tut_id	name	t_region
1	Jim	2
2	Sue	3
3	Mary	4

Student			
stud_id	name	s_region	mentor_id
1	Mary	2	1
2	Jim	3	2
3	Bob	1	3

A Tutor table which includes a column recording the region number, a Student table which includes a column recording the id of the tutor who is the student's mentor, and a column recording the student's region number.

Question 7. Given the above relations, how would you define a constraint that a mentor must belong to the same region as the student?

Question 8. Entity integrity is usually secured by the definition of one or more fields as the primary key. In what circumstances might this action in fact fail to guarantee entity integrity of the enterprise data? What steps might be taken to rectify the position?

When you have completed the questions check your answers with ours. For some of the questions there several possible answers. If you disagree with our answers or have arrived at interesting alternatives, please post a comment about it in the unit 3 forum.

Unit 3 Topic 2 exercise answers

Question 1

- Each cell holds a single value
- Each attribute has a unique name
- Attribute values must be from the same domain
- Order of rows is not significant
- Order of columns is not significant

Note that many systems do in practice allow data to be selected by column location. (For example, you will often see php scripts refer to, for example $theRow[2], which means the contents of the third column in this row.)

© Copyright the University of Oxford and contributors.

Question 2

Two rows – showing red and an empty cell, but not the null cell

Question 3

- It must guarantee the tuples are unique
- It must not be reducible (no part of it is capable of guaranteeing the tuples are unique)

Question 4

```
Candidate Key, Primary Key, Foreign Key, Alternate Key, Surrogate Key
```

These are important concepts. If you are not sure you understand them please refer back to Section 2.4 of this topic.

Question 5

Domain Integrity ensures attributes do not hold values outside a specified domain or type. Entity Integrity ensures that each tuple of a relation is unique. Referential Integrity ensures that for every foreign key value there is a corresponding value in the related primary key.

Question 6

```
Student join Enrolment over Student_id giving A
A join Course over Course_id giving B
select B where Name='Java' giving C
Project C over L_Name,F_names
```

Question 7

```
Tutor join Student over tut_id=mentor_id giving A
Select A where t_region=s_region giving B
A difference B is empty
```

Question 8

When a surrogate key is used as primary key, enterprise data may be duplicated since there will always be a new auto number, regardless of the data inserted. Possible ways of restoring entity integrity are to use a no-duplicates index in another field or, as a last resort, to flag up the potential problem to the application designer.

© Copyright the University of Oxford and contributors.

Unit 3. Database design

3.1. Introduction and objectives

This topic brings together relational theory and methodologies of data design. We expect you to have previously studied data design techniques in a systems analysis unit or course. Here we look at some of the same techniques, but we focus on the way they relate to relational theory and the formal definition of a database.

The central data model, in both the analysis and design of a system, is a model of the entities or classes of data in the system and the relationships or associations between them. Since in most small and medium-sized systems (and many large ones) the data storage will involve a relational database, we concentrate here on the traditional entity-relationship diagram (ERD), and how it is used to inform the formal definition of a database.

Another important approach to analysing and designing data is the process of normalisation of a set of data into an acceptable normal form. Though we do not cover this in the same depth as you would cover it in an analysis and design unit, we look again at the ideas underlying the process, and the way in which they express and impose relational database principles.

The main part of this topic is practical. The formal design of a database involves drawing up a schema based on the data model presented in the ERD, the normalised tables and any associated narrative. It involves documenting domains, relations, attributes and constraints.

By the end of this topic you should be able to:

- appreciate the range of ERD methodologies you may have to interpret
- identify the constraints implied by ERDs and associated text
- relate the normalisation of data to relational theory
- produce a formal schema for a database.

3.2. From design to definition

Systems engineering can be anything from the informal production of a modest application by one person to a large, formally managed project undertaken by a multidisciplinary team. The person or team responsible for the definition and design of the database may have been involved in the original analysis and design process, but that is by no means certain. The special skills of this team may not be employed until the initial design process is completed. Whether it happens as an integral part of the design process or as an entirely separate process, it is always necessary to express the data design in terms of a **database schema**. This involves a critical appraisal of all the design documentation, for example the entity-relationship diagrams, data dictionaries, textual description of the specifications and any database or enterprise constraints that are, explicitly or implicitly, contained in the design.

Ultimately the database schema will be produced in the data definition language of the selected database management system, but the initial schema definition is written in a generic, less syntactically critical format. There are a number of reasons for having this generic schema:

- It ensures that there is a sound theoretical basis for the schema

© Copyright the University of Oxford and contributors.

- It enables the schema to be produced before a final decision is made on choice of DBMS
- It provides information that helps in the choice of DBMS
- The schema may need to be implemented in more than one DBMS
- There may be features and constraints that could not be implemented by a particular DBMS but which still need to be formally defined, so that they can be addressed by applications designers or user guide authors

Although the formal schema is generic and less dependent on strict syntax, it does have common conventions. It sets out the structure of the database including:

- domains from which data values are drawn
- relations
- attributes of relations
- relationships expressed in terms of primary and foreign keys

It also contains a range of attribute, relation, database and enterprise constraints on the data.

A schema will also involve a definition of a **user schema**. This defines attributes in the same way as the definition of the database schema, but it also includes a definition (in relational algebra) of the way the user schema relation is derived from the schema relations. (Attributes of user schema relations are almost always from the same domain or data type as the respective attributes in the main schema, but for clarity and ease of use, domains or data types should be included in the user schema.)

As ever in systems engineering, the precise format to use is the one adopted by your team, but it is likely have the following generic structure:

© Copyright the University of Oxford and contributors.

```
Schema DatabaseName

domains
    domain 1  datatype constraints
    domain 2  datatype constraints
     ……
    domain n  datatype constraints

relations
   relation relationName
      attribute 1 domain (or datatype)  [constraints]
      attribute 2 domain (or datatype) [constraints]
      ……
      attribute n domain (or datatype) [constraints]
   [Primary, alternate and foreign key definitions]
   [Constraints that are confined to this relation]

   relation relationName


database and enterprise constraints

   [definitions, usually in relational algebra, of constraints involving more than one relation]


userschema userschemaName

   relation relationName
      attribute 1 domain (or datatype)  [constraints]
      attribute 2 domain (or datatype) [constraints]
   mapping:  relational algebra expression mapping to the main schema relations(s)
```

Domains

The schema contains a list of the domains from which data values can be drawn. Domains were discussed in Topic 2 and you may wish to re-read the relevant section before undertaking the exercise at the conclusion of this topic.

It is simply a list of data types and acceptable data values. For example:

Domain	Data type	Values
StudentIdentifier	integer	within the range of the DMBS autonumber range
Birthday	Date	Between 1/1/1920 and 31/12/2003
Membertype	text(8)	in (‘junior’, ‘senior’, ‘standard’)

© Copyright the University of Oxford and contributors.

Traditionally the data types used in database schemas were Pascal data types but the precise terms used are not important so long as they are unambiguous. You might, for example, equally use varchar(), char() or String instead of text(). It is, however, important to use generic data types in the schema. For example, even if you knew you would be intending to implement the database in mySQL you would not define the date as 'yyyy'mm'dd' but as a generic Date type.

As well as providing an overview of data types, the definition of domains provides a theoretical underpinning of the primary key/foreign key links that implement relationships. The values of both keys must come from the same domain.

Relations

Relations are defined as a list of attributes and their domains, and any related constraints. Some of the constraints are included in line with the attribute definitions. Others are collected together at the end of the relation in a **constraints** section. In many cases (for example primary, alternate and foreign key constraints), it does not matter whether they are defined in line or in the constraints section. Here is an example showing both approaches:

```
relation Student
      student_id identifierOfStudents
      lastName  text(100) not NULL
      firstName text(200) not NULL
      landPhone phoneDomain
      mobile phoneDomain
      region identifierOf Regions not NULL
    primary key: student_id
    foreign key: region references Region
   constraint:
        A ← SELECT Student  student_id = n
        B ← PROJECT A OVER landPhone
        C ← PROJECT A OVER mobile
        B UNION C is not empty
```

If you have studied this subject before you may have come across the alternative approach (equally acceptable) that defines constraints in line where possible, for example:

```
relation Student
      student_id identifierOfStudents  primary key
      lastName  text(100) not NULL
      firstName text(200) not NULL
      landPhone phoneDomain
      mobile phoneDomain
      region identifierOf Regions not NULL foreign key references Region
      constraint:
        A ← SELECT Student  student_id = n
        B ← PROJECT A OVER landPhone
        C ← PROJECT A OVER mobile
        B UNION C is not empty
```

© Copyright the University of Oxford and contributors.

The definition of the foreign key must show the relation that it references. Since it will almost always reference the primary key of that table, the name of the attribute is usually omitted but some teams like to include it, again for clarity and ease of reading. So the above foreign key definition would be:

foreign key: region references Region (region_id)

Where the relation includes an alternate key, it is defined in the same way as a primary key.

Database and enterprise constraints

This section includes constraints that are inappropriate or not possible within the definition of a single relation. Each constraint has a name and a definition, usually in relational algebra. In order to use examples you have already met in the previous section, these examples are taken from different databases:

The first assumes a schema containing relations named Text and Graphical:

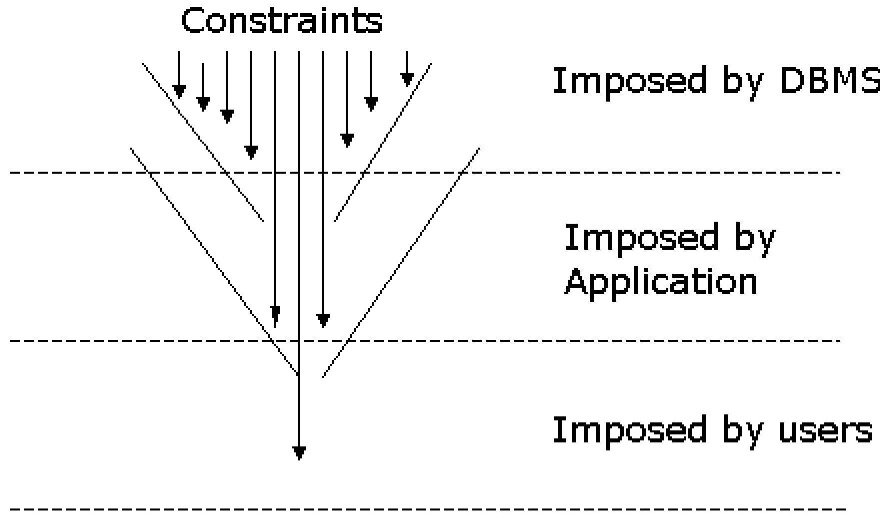
```
constraint: pageContents
    A ← project Text over page_id
    B ← project Graphical over page_id
    A intersection B is empty
```

The second assumes a college schema that includes an Enrolment relation:

```
constraint: maxCourses
     A ← SELECT Enrolments student_id=n
     B ← PROJECT A OVER course_id
     COUNT (B) <4
```

Although this constraint involves only one table it is a constraint that you may decide should be implemented by the application designer or even the user's instruction manual, rather than be embedded in the database.

The idea is to identify here all types of constraints, so that decisions can be made whether to implement them in the DBMS schema, in the applications using the database or in the guides provided to users.



© Copyright the University of Oxford and contributors.

User schema

Here is an example of the way a user schema is defined. The example is based on a Student relation, which holds the id and name and other details of students, and a Counsellor relation, which holds details of a counsellor including the counsellor's office phone, private address and qualifications. The full relations are accessible only by admin staff and senior managers, but all staff should be able to see which counsellor is allocated to which student.

Userschema GeneralStaffView

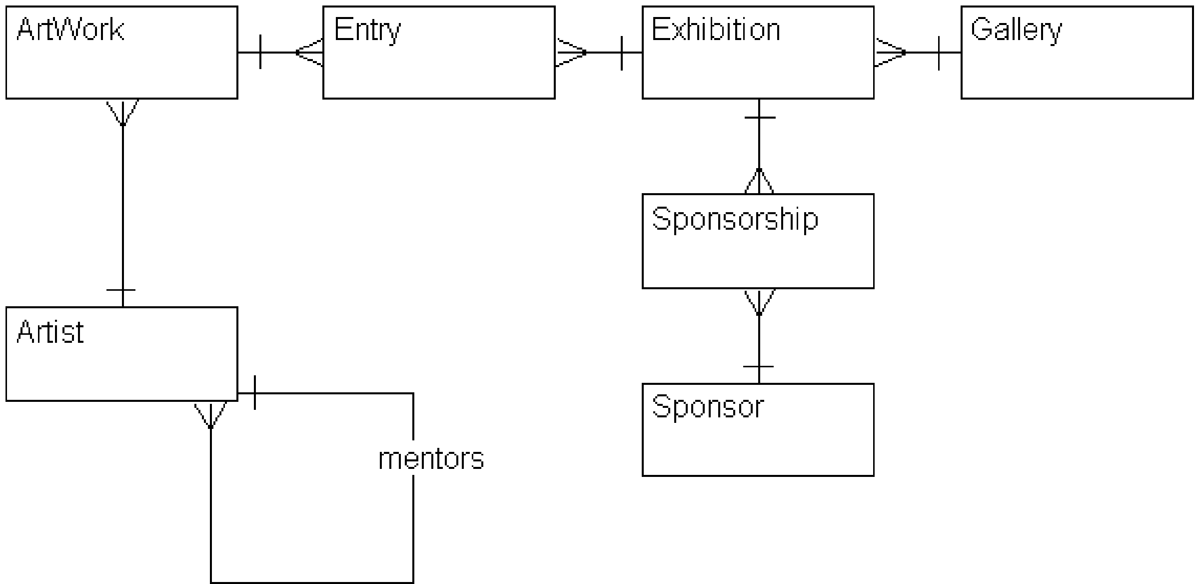
```
relation StudentsCounselled
   StudentLastName
   StudentFirstName
   Counsellor.firstNames,
   Counsellor.lastName,
   Counsellor.officePhone
Mapping:
   Student JOIN Counsellor over Counsellor_id giving A
   Project A over Student.firstName, Student.lastName, Counsellor.firstNames, Counsellor.lastName,
Counsellor.officePhone
```

3.3. Entity-relationship diagrams – cardinality

The way you draw up an entity-relationship diagram depends largely on where you studied the subject or on the organisation in which you work. There are several popular approaches and unfortunately not only do they differ in the symbols they use but also the same symbol can mean different things depending on the model used. An educational course would normally stick to one methodology to avoid confusion (as we do in the Systems Analysis unit), but there are good reasons why we need to depart from this principle here. The design of a database may involve interpreting existing ERDs drawn up by others and it is important for you to be aware of the different methodologies you may meet. This is particularly important because some Computer Aided Software Engineering (CASE) tools do not support the methodologies favoured by some educational institutions.

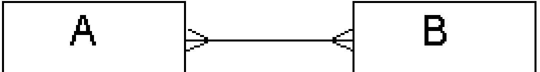
The basic ERD looks pretty much the same in any methodology and whichever one you have studied you should have no problems interpreting the following:

© Copyright the University of Oxford and contributors.

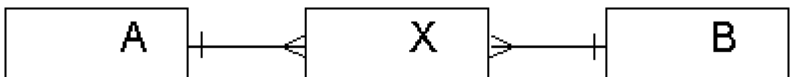


Entity-relationship diagram of an artists' organisation and its exhibitions.

The relationships between real world data entities usually include many-to-many relationships, which are broken down in the data design process into two one-to-many relationships with a new intersection entity. Generalising, we can say that each instance of



in the model of the real world data becomes



in the logical model for the database.

A and B might be entities like Students and Courses, Swimmers and Races, Bands and Gigs or Horses and Jockeys. In each case you need a new entity (and no harm in calling it X for now), which has as a minimum of two attributes: one from each of the domains of the primary keys of A and B. In generic terms, if the primary keys of A and B are A_id and B_id, then each instance of entity X will have foreign keys that reference an A_id and a B_id. To move from the generic to the examples, an Enrolment entity will have attributes Student_id and Course_id; an Entry entity will have attributes Swimmer_id and Race_id, and so on.

Pause for thought

What is the primary key of the intersection entity X?

Answer

© Copyright the University of Oxford and contributors.

The primary key is the joint primary key comprising the two foreign keys. For example, each row in the Entry entity uniquely identifies a particular swimmer associated with a particular race. The swimmer may appear many times and the race may appear many times, but that combination is unique.

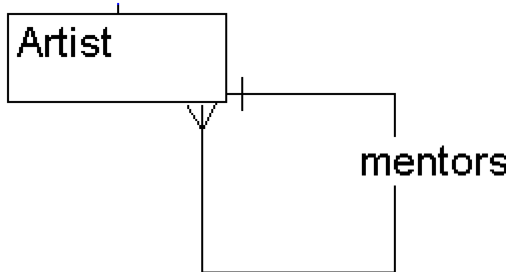
In fact the intersection entity will often have other attributes as well as the two foreign keys. Some essential data can only be held in intersection tables. What other fields do you think would be held in the Entry intersection table?

Answer

I think this would be the place to record a swimmer's time and position in a particular race.

These relationships, and the way we implement them in the relational model, are just a special case of the simple one-to-many relationship which is implemented through the link between two attributes, usually a primary key and a foreign key. You can easily work it out from first principles that the primary key is always at the *one* end and the foreign key at the *many* end.

The foreign key can be in the same table as the primary key, as illustrated in the recursive relationship labelled 'Mentors' in the artists' club diagram



Pause for thought

If we simply implemented this relationship by including a *Mentor* field in the Artist relation it would be possible for the database to record that a mentor mentored himself. This is clearly wrong and so we should define a constraint to prevent this happening. As there is only one relation involved, the constraint can be included as part of the definition of the relation. How would you define this constraint?

© Copyright the University of Oxford and contributors.

Pause for (lateral) thought

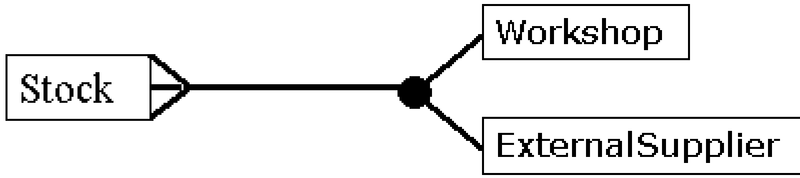
Whilst we are thinking about the mentoring arrangement, consider a constraint that would not be indicated by the ERD but could well be included in the associated text. Assume the Artist entity had an attribute for grade of membership (junior, standard or senior), so that the table based on the entity would include:

artist_id	**etc**	**etc**	**grade**	**mentor**
1			standard	4
2			senior	4
3			junior	1
4			senior	2

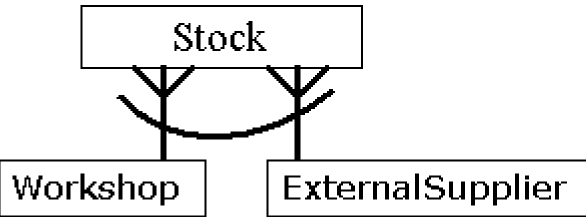
and that a mentor must always be a senior member.

How would you define this constraint?

An ERD will sometimes indicate that two relationships are exclusive. The source of an item of stock might be either the workshop or an external supplier, but never both, This kind of exclusivity may be shown in different ways. In some methodologies it is shown by the use of a filled circle:



In other methodologies it is shown by an arc over the relationship lines:



The stock relation would include workshop and externalSupplier fields, one of which would hold a foreign key referencing the relevant table. But there should never be a tuple with an entry in both fields.

© Copyright the University of Oxford and contributors.

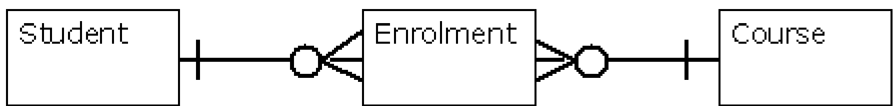
Pause for thought

The database definition needs to capture this exclusivity. Assuming the two fields are defined as being in the same domain, can you work out a way of defining it using relational algebra?

3.4. Entity-relationship diagrams – existence

An entity-relationship diagram also provides information about the existence of relationships. Some relationships are, with respect to an entity, optional, whilst others are mandatory.

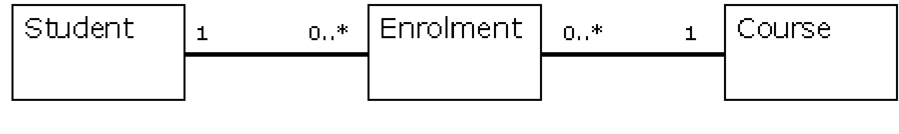
Using the Information Engineering notation (the style we use in this course) the nature of the participation of an entity in a relationship is shown by a symbol at the *opposite end* of the relationship line. You stand in the box, as it were, and look along the line to the symbol which signifies 'your' participation.



Information Engineering Method for an ERD

An instance of Enrolment must participate in a relationship with a Student instance because, looking along the line, there is a vertical bar. A Student instance need not participate in the relationship because, looking along the line, there is a clear circle.

The Martin model, which uses numeric symbols to denote existence, has recently become very popular because it is very similar to the notation used in the Unified Modelling Language (UML) class diagram:



UML-like Method

As with the Information Engineering method, participation of an entity is denoted by what is shown at the opposite end of the relationship line. Thus a Student instance *may* be related to *zero* or *many* instances of Enrolment, but an Enrolment instance *must* be related to one instance of Student. Using this approach it is possible for the diagram to capture more of the semantics of the data. For example, if a member were allowed to participate in a maximum of two races at a swimming gala, this could be captured by the ERD:

More Semantics in this Type of ERD

© Copyright the University of Oxford and contributors.

How would you define the above constraint in the database schema?

For this constraint you will need to use the operation COUNT. This is not part of the pure relational algebra but is usually used as if it were, when defining constraints. The syntax is:

```
COUNT(relation)(attribute)
```

For example,

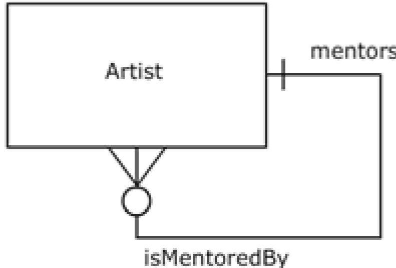
```
COUNT (Member)(mentor)
```

would tell you how many members had mentors.

Answer

```
SELECT Entry WHERE swimmer_id = n giving A
COUNT A (race_id)<3
```

On a similar theme, consider how the Mentors relationship might look in a more detailed ERD:



How would you define this constraint in your schema?

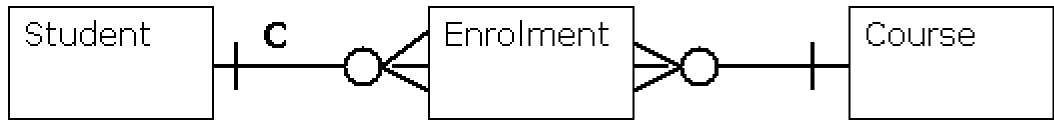
An artist may or may not be a mentor, but each artist must have a mentor.

Answer

You do not need to resort to relational algebra for this one. All you need to do is constrain the Mentor field of the Artist relation as not null.

© Copyright the University of Oxford and contributors.

ERDs do not normally include information about referential integrity, but you may come across the symbol C against an entity if you are working to a model designed using the Structured Systems Analysis and Design Method (SSADM):



C Symbol Representing Cascade Delete

This C symbol indicates that referential integrity should be enforced by causing **cascade** deletes. If a record is deleted from the Student table, it should cause automatic deletion of any related records in the Enrolment table. (The default is to *prevent* the deletion of records that are referenced by a foreign key.) For the practical work in this topic we will present ERDs drawn up in each of these formats for comparison purposes, but we ask you to use the diagram in the Martin style for the exercise because it contains information not carried by the other diagrams.

3.5. Normalisation

Data is normalised early in the analysis and design process. It is the process of decomposing a data-set into relations or, to put it another way, it is breaking the data down into tables each of which satisfy all the relational rules. When analysing documents like the invoice below, the analyst can see at once that there are several entities represented, and the normalisation of the data-set will be reflected in the entity-relationship model.

Engineering Supplies Ltd - Invoice

Order No: 23648 | Customer: John Smith
Date: 23/05/2004 | Address: 23 The Grove
Taken by: Jim Hall | Newtown
NT1 3TN
Phone: 019283

Product	Quantity	Price	Total
Widget	1	45	45
Wheels	4	25	100
Total			145
P&P			5
Invoice Total			150

Invoice Data

This pragmatic approach is common in initial data models. Normalisation is seen as a way of removing repeating groups and redundancy. It is, for example, usually clear from the start that repeating groups need to be taken to a new table, and so this is often the first action taken.

© Copyright the University of Oxford and contributors.

In this unit, however, we are more concerned with the way in which normalisation informs and enforces the principles of the relational database, and we will look at each level of normalisation separately. The data-set we will use for this analysis is rather artificial compared with the above representation, but it contains the same sort of data. There are a few minor changes (I have left out addresses to achieve narrow columns), but the following table contains essentially the same kind of data-set as in the invoices:

Order No	Date	Customer	C_ Phone	Sales Asst	S_PhoNe	Part No	Part	Qty	Price
23648	23/05/2004	John Smith	019283	Jim Hall	017364	P1 P5	Widget Wheel	1 4	45 25
23649	24/05/2004	Bill Giles	012948	Jim Hall	017364	P6 P9 P5	Facia Flange Wheel	2 5 8	75 15 25
23650	30/05/2004	Jane Green	018274	Sue King	017365	P2	Door	1	45
23651	01/06/2004	Sam Dean	013945	Jim Hall	017364	P5	Wheel	2	25
23652	02/06/2004	Bill Giles	012948	Sue King	017365	P1 P2	Widget Door	2 10	45 45
23653	02/06/2004	Bill Giles	012948	Sue King	017365	P10 P6	Strut Facia	20 1	5 75
23662	04/06/2004	John Smith	019283	Sue King	017365	P9 P5	Flange Wheel	10 2	15 25

You will recall that the first stage of formal normalisation is to put the data-set into First Normal Form (1NF). The simple definition of 1NF is that a table is in 1NF if every cell contains a single-value fact (or every cell is 'atomic') and if the table does not contain repeating groups. In the table above, a number of cells are not atomic. Some cells contain multiple values for parts. Strictly the name fields also contain multiple values. In practice much depends on the software being used and the anticipated use of the database: sometimes full names do occupy a single field. But here we take the strict view that last name and other names are distinct facts.

The data-set is transformed into 1NF by introducing new rows or columns. Where the multiple facts relate to the *same type* of fact (like Part No) they are divided into *new rows.* Where they are of *different types* of fact (like people's names) they are divided into *new columns.* It is important to get these the right way round. The solution is quite obvious in the above table, but sometimes you may be tempted to split same type facts into columns instead of rows, for example:

© Copyright the University of Oxford and contributors.

Emp_id	Name	Projects
1	Jones	EJ345 NS345
2	Smith	NS345
3	Brown	JU465 EJ345

→

Emp_id	Name	Project1	Project2
1	Jones	EJ345	NS345
2	Smith	NS345	
3	Brown	JU465	EJ345

Repeating Groups

The new table is still not in 1NF because, although the cells are now atomic, there are repeating groups of the same type of fact. In row one there are two columns each holding the same sort of fact. (It is wrong in principle but you can see it is also wrong in practice. If you have too few columns for Projects you have to redesign the database if more contracts are allocated. If you leave many empty cells to be on the safe side you waste storage space and processing time.)

The orders table, transformed into 1NF looks like this:

Data in First Normal Form

Order No	Date	C_LN	C_FN	C_Phone	S_LN	S_FN	S_Phone	Part No	Part	Qty	Price
23648	23/05/04	Smith	John	019283	Hall	Jim	017364	P1	Widget	1	45
23648	23/05/04	Smith	John	019283	Hall	Jim	017365	P5	Wheel	4	25
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017364	P6	Facia	2	75
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017365	P9	Flange	5	15
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017366	P5	Wheel	8	25
23650	30/05/04	Green	Jane	018274	King	Sue	017365	P2	Door	1	45
23651	01/06/04	Dean	Sam	013945	Hall	Jim	017364	P5	Wheel	2	25
23652	02/06/04	Giles	Bill	012948	King	Sue	017365	P1	Widget	2	45

© Copyright the University of Oxford and contributors.

23652	02/06/04	Giles	Bill	012948	King	Sue	017365	P2	Door	10	45
23653	02/06/04	Giles	Bill	012948	King	Sue	017365	P10	Strut	20	5
23653	02/06/04	Giles	Bill	012948	King	Sue	017365	P6	Facia	1	75
23662	04/06/04	Smith	John	019283	King	Sue	017365	P9	Flange	10	15
23662	04/06/04	Smith	John	019283	King	Sue	017365	P5	Wheel	2	25

It is a very odd table, but it is a useful stage in the normalisation process. It makes clear (as we would know instinctively) that the key to the data-set is a joint key. It also provides a focus for discussing functional dependency. Functional dependency is described using expressions like X --> Y where X and Y are attributes or groups of attributes. X --> Y means Y is functionally dependent on X, or that X determines Y. In the table above we can say OrderNo --> Date.

In fact we can go further and say OrderNo --> Date, C_LN, C_FN, C_Phone, S_LN, S_FN, S_Phone. But that is as far as we can go. Knowing the order number we can be sure of the values of those columns, but we cannot be sure of the values in the remaining columns of the table. We can, however, say Part No --> Part, Price. The values of Part and Price are dependent on the value of PartNo. That leaves Qty. What is that dependent on? That is dependent on two fields, Order No and Part No, or to use the symbolic notation (OrderNo, Part No) --> Qty. It can be helpful to identify dependencies by drawing them on the 1NF table like this:

Order No	Date	C_LN	C_FN	C_Phone	S_LN	S_FN	S_Phone	Part No	Part	Qty	Price
23648	23/05/04	Smith	John	019283	Hall	Jim	017364	P1	Widget	1	45
23648	23/05/04	Smith	John	019283	Hall	Jim	017365	P5	Wheel	4	25
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017364	P6	Facia	2	75
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017365	P9	Flange	5	15
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017366	P5	Wheel	8	25
23650	30/05/04	Green	Jane	018274	King	Sue	017365	P2	Door	1	45
23651	01/06/04	Dean	Sam	013945	Hall	Jim	017364	P5	Wheel	2	25
23652	02/06/04	Giles	Bill	012948	King	Sue	017365	P1	Widget	2	45
23652	02/06/04	Giles	Bill	012948	King	Sue	017365	P2	Door	10	45
23653	02/06/04	Giles	Bill	012948	King	Sue	017365	P10	Struts	20	5
23653	02/06/04	Giles	Bill	012948	King	Sue	017366	P6	Facia	1	75
23662	04/06/04	Smith	John	019283	King	Sue	017365	P9	Flanges	10	15
23662	04/06/04	Smith	John	019283	King	Sue	017366	P5	Wheel	2	25

Data in First Normal Form

© Copyright the University of Oxford and contributors.

Once you have established the dependencies, the identification of at least one candidate key is trivial. It is (Order No, Part No), which can uniquely identify any row of the table. Picturing the dependencies in this way makes it easy to proceed to Second Normal Form. To be in 2NF, all columns in the table must be part of, or wholly dependent on, the primary key. It is clear from the diagram that to be in 2NF the table can only contain the columns Order No, Part No and Qty. All other columns need to be removed to other tables, together with a copy of the part of the primary key on which they are functionally dependent. The colour coding makes it easy to see what these tables are. All the red columns form the Orders Table and all the blue columns form the Parts table.

Pause for thought

Before moving on, look back at the Data in First Normal Form image and identify the dependencies. Can you see any which were not mentioned in the above paragraph? More about this in the next paragraph.

When we identified functional dependencies in the table, you may have seen other things that look like functional dependencies. Is Price dependent on Part as well as on Part No? Is C_Phone dependent on C_LN and C_FN as well as on Order No? I would say Price is 45 because the product number is P1 and that remains so even if we change what we call the part. C_Phone is, however, functionally dependent on the customer names, as well as on Order No. There is a similar dependency of S_Phone on the sales names. This leads to the definition of Third Normal Form (3NF). To be in 3NF, all dependencies must be on key fields.

Another way of expressing this rule is that to be in 3NF there must be no transitive dependencies. Although the phone numbers in the Orders table are dependent on the primary key (OrderNo) they are also dependent on fields that are not keys (the name fields). Or, to put it another way, the phone numbers are only transitively dependent on the key fields. Customer phone is dependent on customer names which are in turn dependent on Order No. So whilst customer phone does depend on Order No it does so only *transitively.* Again it helps if we draw the transitive dependencies on the table, thus identifying the new tables that need to be created:

Order No	Date	C_LN	C_FN	C_Phone	S_LN	S_FN	S_Phone
23648	23/05/04	Smith	John	019283	Hall	Jim	017364
23648	23/05/04	Smith	John	019283	Hall	Jim	017365
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017364
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017365
23649	24/05/04	Giles	Bill	012948	Hall	Jim	017366
23650	30/05/04	Green	Jane	018274	King	Sue	017365
23651	01/06/04	Dean	Sam	013945	Hall	Jim	017364
23652	02/06/04	Giles	Bill	012948	King	Sue	017365
23652	02/06/04	Giles	Bill	012948	King	Sue	017365
23653	02/06/04	Giles	Bill	012948	King	Sue	017365
23653	02/06/04	Giles	Bill	012948	King	Sue	017366
23662	04/06/04	Smith	John	019283	King	Sue	017365
23662	04/06/04	Smith	John	019283	King	Sue	017366

Orders Table in Second Normal Form

© Copyright the University of Oxford and contributors.

In practice the natural primary key of these people tables would involve more columns than used in this example, and so the use of an id number, even if it is a surrogate key, is justified. The Order table is now decomposed producing a 3NF data-set in five tables.

Order

Order No	Date	C_id	S_id
23648	23/05/04	c1	s1
23649	24/05/04	c2	s1
23650	30/05/04	c3	s2
23651	01/06/04	c4	s1
23652	02/06/04	c2	s2
23653	02/06/04	c2	s2
23662	04/06/04	c5	s2

Salesperson

ID	S_LN	S_FN	S_Phone
s1	Hall	Jim	017364
s2	King	Sue	017365

Customer

ID	C_LN	C_FN	C_Phone
c1	Smith	John	019283
c2	Giles	Bill	012948
c3	Green	Jane	018274
c4	Dean	Sam	013945
c5	Smith	John	019283

Part

Part No	Part	Price
P1	Widget	45
P2	Door	45
P5	Wheel	25
P6	Facia	75
P9	Flange:	15
P10	Struts	5

Order Item

Order No	Part No	Qty
23648	P1	1
23648	P5	4
23649	P6	2
23649	P9	5
23649	P5	8
23650	P2	1
23651	P5	2
23652	P1	2
23652	P2	10
23653	P10	20
23653	P6	1
23662	P9	10
23662	P5	2

Data in Third Normal Form

Note how the removal of transitive dependencies makes it possible to avoid duplication of people names and phone numbers. Note also the primary keys and foreign keys in the 3NF tables.

Pause for thought

What are the primary and foreign keys in these tables?

Answer

```
Primary keys are:
  Order No in Order table
  Part No in Part table
  Order No and Part No (joint primary key) in Order Item table
  ID in Salesperson table
  ID in Customer table

Foreign keys are:
  C_id and S_id in Order table
Order No and Part No in Order Item table
```

This brief diversion into normalisation is just one way of looking at the subject and it does not cover a lot of the theory and practice. As mentioned at the beginning of this topic, we do expect you to have followed a previous unit or course on systems analysis that will have included normalisation.

© Copyright the University of Oxford and contributors.

3.6. Names and addresses

Throughout this topic we have always included names in people entities using the attributes **firstNames** and **lastName**. We discussed in our overview of normalisation whether a person's names should be regarded as a single-valued fact, and concluded that in most cases there are two facts, the firstNames and the lastName, but we have skated over a number of other potential facts in a person's name. Similarly we have restricted contact details to a telephone number or e-mail address. This is of course unrealistic but is necessary in order to restrict the number of columns in the tables we are using for the examples. In practice the way you treat names and addresses is an important feature of database design.

Names are capable of such complexity that it is not always possible to capture all their subtleties in your design, but there are some general principles to which database designers generally work. In most databases it is necessary to include a **prefix**, or **title** field and a **suffix** field. You must be able to cope with:

Sir Arnold John Fotherington Smythe MP

In many applications it is enough to have these four fields, but it is quite common to need other ways of referring to an individual. For example, an executive's contact database may require the form of address to use in a letter. The salutation for one person may be 'Dear Mr Swan' whilst for another it is 'Dear Donald'. Another thing you might need if you use a database to extract names for use in internal papers is the short name, or everyday name, to be used in those circumstances. Thus the above gentleman might have letters addressed to Sir Arnold John Fotherington Smythe MP, with the salutation Dear John, and in internal memos be referred to as John Fotherington Smythe.

Some designers would quarrel with my use of field names firstNames and lastName, preferring the terms **givenNames** and **familyNames**. In some traditions, however, the family name precedes the given name and unless you store additional data about the name, and engage in some processing each time you retrieve the data, using these terms produces more problems than it solves.

Is an address, or part of an address, a single-valued fact? It does, of course, depend on the nature of the data and its purpose but generally an address is composed of many facts. The postcode or zip code is a fact we may want to use to partition a table or to produce a mailing list. In some countries the house name or number, combined with the postcode, makes it possible to check the other details of an address in a national database. So the minimum fields needed for an address are:

- houseName
- houseNumber
- otherAddressDetails
- townOrCity
- county
- postOrZipCode

There is an argument for further separation of the otherAddressDetails into, for example addr1, addr2, etc. in order to avoid the storing of carriage returns.

© Copyright the University of Oxford and contributors.

Pause for thought

What constraints should we put on address fields? It is possible to check a postcode or zip code is not obviously wrong (for example if it has too many characters, if it contains impossible symbols or, in the UK, if it starts with a numeric character). In some countries it may be possible to impose more detailed constraints, but care is needed in implementing such constraints in the database because of the possibility that the postal authority may subsequently change the range of values used. One thing you should always consider in address fields is whether nulls should be allowed. In all databases the middle bits of the address must be capable of being null because they may not exist. In some applications (for example log-in details on a website) you would want to capture other details in the relation even if you could not get an address. But where an address is important you would usually want to constrain as not null the post or zip code and the house name or number.

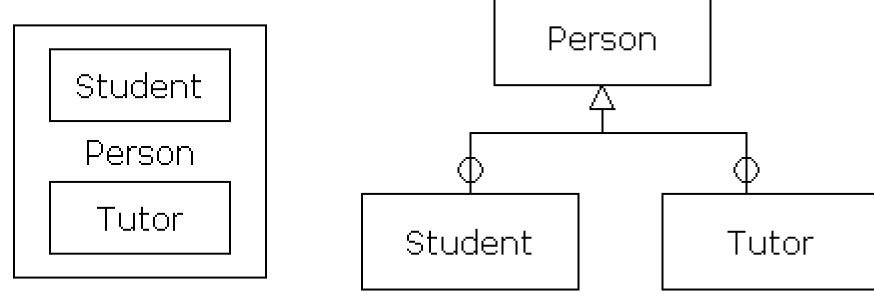
The postcode is easy: we simply define the field as *not null*. The house name or number is more difficult because either of them can be null so we cannot impose the constraint in the field definition. Can you work out a way to apply a constraint on the relation?

Answer

This answer assumes that the houseName and houseNumber fields are from the same domain.

```
SELECT  Person WHERE person_id=n GIVING A
PROJECT A OVER houseName GIVING B
PROJECT A OVER houseNumber GIVING C
B UNION C IS NOT EMPTY
```

Another point we have glossed over in the example in this unit is the use of sub-types of entities and relations. You may have met these in studying systems analysis where you might identify, for example, person as a super type with lecturer and student as sub-types. There is no standard way of expressing this in entity-relationship diagrams but it is often shown in one of the following ways:

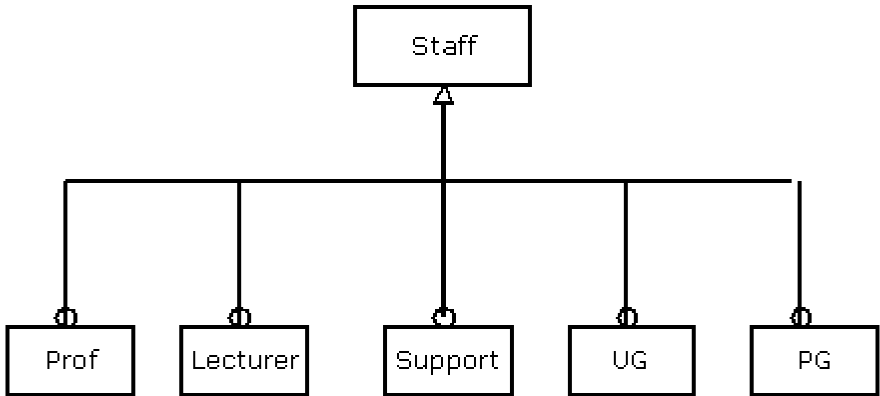


The way sub-types are implemented in a database depends on the degree of duplication that would otherwise occur and how important it is to minimise processing overheads. Data duplication is conceptually wrong and dangerous in practice. On the other hand the keeping of frequently used data in two tables involves a material processing overhead. In a small college it may be that whilst it is possible for a tutor to also be a student, it is not

© Copyright the University of Oxford and contributors.

common and the organisation could live with the risk of the occasional duplication involved in just having Student and Tutor relations. In another context the decision might be that the duplication was unacceptable and the processing overhead was justified.

Apart from the duplication of data, the keeping of separate tables recording names and addresses involves the duplication of attributes which, according to relational theory, should be avoided. (What is meant here of course, is attributes that refer to the same thing. There is no problem using the attribute ‘colour’ in both of the relations Car and Widget. Nor do the use of foreign keys offend this principle.) Imagine a college that needed to record extensive, but different, details of staff within disciplines, for example:



You could imagine one large Staff relation with a large number of fields, some of which would apply to professors, others to undergraduate students, etc., but it would be an unwieldy table to handle in practice.

An alternative would be to accept the possibility of duplication of data and keep all data in the tables Prof, Lecturer, Support, UG and PG. The result would be five tables, each of which had fields for name, address, phone, e-mail, etc.

In these circumstances the sub-types are frequently implemented as relations with attributes appropriate to their own aspects of the person’s data, leaving the core personal data in a Person table.

Staff table

staff_no(PK)	LastName	etc	etc	etc
1	Jones			
2	Smith			

Undergraduate table

UG_id (PK)	staff_no (FK)	course	etc	etc
1	2	2		

© Copyright the University of Oxford and contributors.

2	3	5		

These are difficult design decisions and we do expect you to handle them for the purposes of this course, but you should be aware of the issues involved and be able to recognise databases structured in this way.

3.7. Practical work

Exercise

This is a formative exercise and nothing you do here will affect your mark for the unit. It is, nevertheless an important part of the unit and we want all of you take a full part in it.

As well as helping you to learn the unit materials it introduces transferable skills of more general application.

In Unit 2 we introduced the **Community of Practice** as an important ingredient in management and professional development and discussed how effective learning takes place in a community where:

- There is a shared domain of interest
- The members engage in joint activities
- The group develops a shared repertoire of experiences and ways of addressing recurring problems.

We want you to continue experiencing this sort of community in your Unit 3 tutor group forums. Not only will this make your learning more effective in all the units, it will mean you have acquired the relevant skills by the time you reach unit 5 in which group working forms a major part of the assignment. (We should, though, make it clear that the groups to which you will be allocated in Unit 5 are unlikely to be same as those for Units 2 and 3).

The model with which you are provided for this exercise is a hybrid. It contains a complete entity-relationship model and information about some of the attributes of the entities, but it does not include a complete data dictionary. This is because we want you to practice the interpretation of ERDs and to undertake a critical analysis of the associated textual commentary.

There are a number of domain, database and enterprise constraints implicit in the diagram and text and we want you to identify them and define them in your schema. (Remember you should define constraints in the schema even if you suspect that they cannot or will not be implemented in the SQL schema, so that consideration can be given to implementing them in the business or presentation layer.)

You should download and print Pitkin's Healthy Holidays which contains the ERD and a textual commentary on the model.

We would like you to produce a reference schema for the database, that is, one in generic terms not written for any particular data definition language. There is scope for variation in the layout of the schema and we are not worried about minor details: for example, you may, if you wish, use tables, but the schema should be defined using the categories and terms we have covered in this topic.

© Copyright the University of Oxford and contributors.

The definition of the relations should establish all the relationships in the model and it should be capable of being used without any further amendment (other than translation into a data definition language) to implement the database.

Constraints that apply to more than one relation, or that may be unsuitable for implementation in the DBMS schema, should be grouped in a separate section headed Database and Enterprise Constraints.

This kind of exercise would usually be undertaken by or within a team of developers. That is often important because even the most tightly and carefully defined model can leave some ambiguity, and even seasoned designers can sometimes leap to a wrong conclusion. It is helpful to be able bounce ideas off others and share any misgivings about your solution. We would like you to simulate this informal team approach using the small groups you formed in Unit 2. If you think there is ambiguity or information missing, post your views on the forum and see what others think.

We do not expect you to produce a full group schema, but would like you to exchange examples of specimen domain constraints and definitions of key fields in linked relations.

The definition of constraints is likely to present the most challenges, and we suggest that you post and discuss them one at a time, rather than post a complete list of them.

© Copyright the University of Oxford and contributors.

Unit 4. Database creation and manipulation

4.1. Structured Query Language

This topic is about using the SQL language to manipulate a relational database. There is some theory to study but it is mainly a practical topic. We expect you to send SQL expressions to an online database and view the returned record sets. There are two ways in which we want you to use the online database:

- to enter the SQL expressions we give you in the theory sections, so that you can become familiar with using the interface
- to test the SQL expressions you construct in the practical sections

By the end of this topic you should be able to:

- use Data Definition Language to create a database and tables
- use Data Manipulation Language to interrogate and manipulate a database
- describe the use of Data Control Language to manage access to a database
- describe the principles of transaction processing

4.2. Introduction to SQL

The database approach implies some standard language, which can be used by applications and users to communicate with the database management system. The Network Database Language (NDL) emerged as the standard for the CODASYL DBMS, and Structured Query Language (SQL) was developed alongside the Relational DBMS. Apart from noting its existence, we do not study the Network Database Language in this unit.

The original language for the relational model was developed at IBM in the mid-1970s and was called Structured English Query Language, or SEQUEL (pronounced see-kwell). The name was subsequently changed for legal reasons to Structured Query Language, and the new acronym, SQL, is usually pronounced ess-queue-ell, though some people use the old pronunciation and say see-kwell when referring to SQL.

SQL quickly became the de facto standard for relational databases and in 1987 the International Standards Organisation published a formal standard for it. There have been a number of subsequent revisions to the standard.

Relational Database Management Systems often include extensions to the standard SQL specification, and some systems fail to implement fully the latest ISO standards. Nevertheless over a wide area there is a common language that can be used with any relational database. It is this core language that we study in this topic.

If you wish to pursue your study of SQL beyond the topics covered in this unit, you should have no trouble finding tutorials and manuals using your advanced web search techniques. An excellent electronic book, *Stuctured Query Language (SQL): A Practical Introduction* can be downloaded as a PDF.

© Copyright the University of Oxford and contributors.

Statements in SQL consist of reserved words, which obey syntactical rules and have fixed meanings, and user defined words, generally table and column names. SQL is normally case insensitive and it ignores white space and carriage returns. From the machine's point of view appearance is therefore not important, but it is common to use upper case for SQL keywords and lower case for user words, because SQL is also read by humans. It is also common to start each clause of an SQL command on a new line. Thus whilst:

select name, address, from people where lastName='Brown';

is acceptable to the machine, the command is generally written:

```
SELECT name, address
FROM people
WHERE lastName='Brown';
```

The SQL standard requires each command to end with a semi-colon, but most implementations do not in fact insist on it being used.

The ISO standard recognises two components of SQL, the Data Manipulation Language (DML) and the Data Definition Language (DDL). SQL also contains standards for the control of access to data and the management of databases. The commands used to implement them are usually referred to as the Data Control Language (DCL).

If you have developed a database application using a fourth generation development environment (such asVisual Studio.NET) or application (such as Microsoft *Access*), you may wonder whether there is any point in learning SQL because database applications can be made using graphical user interfaces without the need for any SQL. Solutions can be developed in this way and, at times, especially for rapid prototyping, they are well worth using. But SQL cannot be ignored for the following reasons:

- You may have to work in a different environment where you have no GUI in which to work, or at least a less sophisticated one.
- Applications made using database wizards do not have the flexibility and power of custom-designed solutions.
- If a solution needs fixing or extending, it can be very difficult, if not impossible, to discover how the wizard has produced the application.

We start our tour of SQL by looking at the Data Manipulation Language which is used to extract, update and insert data.

4.3. Using the practice database

In each of the following sections we will be examining the SQL expressions used to create and manipulate relational tables. Most of the examples and exercises use the database practice sites described in this section.

The object of the practice site is to help you learn about using SQL, but you need to use SQL in order to learn how to use the site. The way out of this circle is to start by following a set of instructions mechanically, and observing the results. You may not understand fully what you are doing at first but it will soon fall into place when you study the later sections.

Open the practice site in a new window. (This should happen automatically but, to be sure, hold down the shift key when you click the hyperlink.)

© Copyright the University of Oxford and contributors.

[Link to the practice site]

The opening page contains a single text area into which you can enter messages for the database management system. When you click the Submit button (or, if you prefer to use the keyboard, when you press Tab then Enter), your message is sent to the database. Try it now. Send the following message:

```
show tables
```

You should get a list of tables in the database, something like this:

Tables_in_hncd_gigs
Band
Gig
Lineup
Musician
Spot
Venue

Do not worry if there are more tables in your list. For the moment the above are the tables with which we will be working.

Now replace the expression in the text area with

```
SELECT *
FROM musician
```

You should get a display of the musician table showing all its field names and data, like this:

Musician_id	lastName	firstNames	phone
1	Johnson	Rick	0204756
2	King	Lionel	020459
3	Presley	Fred	029029
4	Murphy	Holly	02789029

© Copyright the University of Oxford and contributors.

5	Quigley	Chuck	020485

Use the same technique to examine the field names and contents of other tables in the database.

Most of the expressions you will be entering will be standard SQL expressions but there are two expressions that are commands peculiar to the mySQL database management system. One of them you have met already (the *show tables* command); the other is the following command, which allows you to see the data types and other constraints on the tables:

```
describe musician
```

Field	Type	Null	Key	Default	Extra
musician_id	int(11)		PRI		auto_increment
lastName	Varchar(100)				
firstNames	Varchar(100)				
phone	Varchar(15)	YES			

Again, use the DESCRIBE command to look at the data types in some of the other tables.

You could, at any time, remind yourself of the structure of the database by using the SHOW TABLES and DESCRIBE [TABLE NAME] commands, but you will find it more convenient to download and print this file which contains all the tables and also an entity-relationship diagram of the data model:

musicians.pdf [Link not available in this medium, please view it in the online course.]

If you have a broadband connection it will be worth having the practice site open throughout your study of this topic. If you cannot remain online for long periods we suggest that you enter any expressions you want to test into Notepad or some other simple text editor, from which you can copy and paste when you go online. A word of caution about copy and paste: some word processors replace the standard single quote ' ' marks the database is expecting with smart quotes ‘ ’. If you paste in an expression and it does not work as intended, replace any quote marks in it and try again.

Later in the topic we will use another database (college), which can be accessed through a link on the musicians database page. When the time comes you may wish to download and print the following document, which contains an ERD and the table structure:

college.pdf [Link not available in this medium, please view it in the online course.]

We start our tour of SQL by examining the way records are extracted from tables.

© Copyright the University of Oxford and contributors.

4.4. Data Manipulation Language

The SQL **Data Manipulation Language** looks a lot like relational algebra. The format and syntax of SQL is different but the underlying logic is the same. SQL is used both to select rows and columns and to join tables together to produce a customised data-set.

You do not need to be online in order to study this section, but if you are online it is worth entering the SQL expressions we use here so that you will be used to using the interface when you come to the practical work.

The SQL SELECT statement is similar to the relational algebra SELECT statement, except that in SQL you can, indeed you must, include the columns you want to project:

```
SELECT name,genre
FROM band
```

name	genre
Hunkydory	Indie
High Kickers	Rock
Krashers	Indie
Silver Birds	Pop
Spanish Nights	Classical

If you want to select all the columns in the table you use the wildcard *

```
SELECT *
FROM band
```

band_id	name	genre	phone	email
1	Hunkydory	Indie	012345	intray@hunkydory.org.uk
2	High Kickers	Rock	0192834	hard@rock.comm
7	Krashers	Indie	2345	us@thequarry.com
8	Silver Birds	Pop	3245255	fly@featherz.org

© Copyright the University of Oxford and contributors.

9	Spanish Nights	Classical	532456	jose@sierranevada.es

You select rows by using a WHERE clause, in the same way and with the same result as in relational algebra:

```
SELECT *
FROM band
 WHERE genre= 'Indie'
```

band_id	name	genre	phone	email
1	Hunkydory	Indie	012345	intray@hunkydory.org.uk
7	Krashers	Indie	2345	us@thequarry.com

Note that the value of the genre column shown in the WHERE clause is enclosed in single quotes. All character type values (varchar, text, char) need to be surrounded by single quotes. Numbers (int, decimal, float) do not need quotes. The formatting of dates differs between DBMSs, but in mySQL, which is the DBMS on our practice site, dates are treated like character strings and need quotes round them.

```
SELECT *
FROM gig
WHERE date = '2004-04-21'
```

gig_id	venue_id	date	Time	cost
3	2	2004-04-21	20:30:00	5.00
4	1	2004-04-21	21:00:00	7.50

A series of conditions in a WHERE clause can be combined using the logical operators AND or OR:

```
.... WHERE band_id=1 OR band_id=2
.... WHERE band_id=1 AND gig_id=2
```

You can also use the operators != (does not equal), < (less than), >(greater than) and the expression BETWEEN value AND value, for example:

```
SELECT date,time
FROM gig
WHERE date BETWEEN '2004-05-01' AND '2004-05-31'
```

date	time

© Copyright the University of Oxford and contributors.

2004-05-21	21:00:00
2004-05-23	20:30:00

You can also use wildcards in character string types by using the keyword LIKE with a % symbol or an underscore. % means zero or any number of characters and an underscore means a single character. Thus:

```
...WHERE venue.name LIKE 'A%'
...WHERE venue.name LIKE '%R%'
...WHERE venue.name LIKE 'Assembly Room_'
```

would all find 'Assembly Rooms'. The first two might well find other venues as well. (The symbols used as wildcards are fairly standard but this is something you should check when you use a new DBMS.)

The rules about null (or empty) cells we met in Topic 2 are applied by SQL. The usual logical operators will always return false if one of the operands is null. There is, however, a function ISNULL(columnName) which returns true if the operand is null.

```
SELECT name from band
WHERE ISNULL(phone)
```

would return a list of bands for whom we have never had a phone number. Note that a null value is not the same as an empty string. In many databases a deleted character string becomes an empty character string and if you wanted a list of bands for whom there is currently no phone number you might need to say:

```
SELECT name from band
WHERE ISNULL(phone)
OR phone=' '
```

Joining tables

The theta (or inner) join is represented in SQL by including the names of all the required tables in the FROM clause and identifying the linked fields in the WHERE clause.

```
SELECT date, name
FROM gig,venue
WHERE gig.venue_id = venue.venue_id
```

date	name
2004-03-21	Assembly Rooms
2004-04-22	The Dog and Gun
2004-04-21	Dog and Parrot

© Copyright the University of Oxford and contributors.

2004-04-21	Assembly Rooms
2004-05-21	Hard Rock Cafe
2004-05-23	The Jazz Cafe
2004-09-22	The Jazz Cafe

Note in the above example that where we used an ambiguous column name (venue_id) we prefaced it with the table name and a period.

If you have long table names you can use short references to them (usually called aliases). For example, the above expression could have been written:

```
SELECT g.date, v.name
FROM gig g, venue v
WHERE g.venue_id = v.venue_id
```

You often need to join several tables to obtain the data you want, and for each join you need a separate condition in the WHERE clause. For example, to obtain a list of the dates of gigs and the bands booked for them, you need to join the band, spot and gig tables:

```
SELECT date, name
FROM band b, spot s, gig g
WHERE b.band_id=s.band_id AND s.gig_id = g.gig_id
```

date	name
2004-03-21	Hunkydory
2004-04-21	Hunkydory
2004-04-21	High Kickers
2004-04-22	High Kickers
2004-05-21	Krashers
2004-05-23	Silver Birds
2004-04-21	Spanish Nights
2004-03-21	Spanish Nights

© Copyright the University of Oxford and contributors.

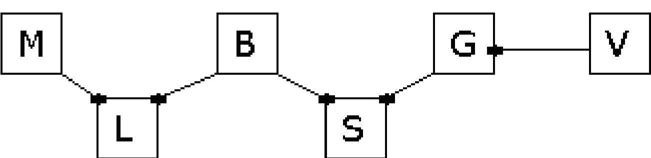
If you add a further table (for example to add the venue name), you need an extra condition in the WHERE clause:

```
SELECT date, b.name, v.name
FROM band b, spot s, gig g,venue v
WHERE b.band_id=s.band_id
AND s.gig_id = g.gig_id
AND g.venue_id = v.venue_id
```

Date	Name	name
2004-03-21	Hunkydory	Assembly Rooms
2004-04-21	Hunkydory	Assembly Rooms
2004-04-21	High Kickers	Dog and Parrot
2004-04-22	High Kickers	The Dog and Gun
2004-05-21	Krashers	Hard Rock Cafe
2004-05-23	Silver Birds	The Jazz Cafe
2004-04-21	Spanish Nights	Assembly Rooms
2004-03-21	Spanish Nights	Assembly Rooms

Note how it was necessary in the above statement to qualify the name column because now that we have included the band and the venue table, name becomes an ambiguous reference.

You often have to join a number of tables, and the need to include all the joins in your statement can at first seem a bit intimidating. It helps if you jot down a diagram of the table structure to work from. Suppose we want some information about musicians that have played at the venue called Assembly Rooms. In order to link musicians to venues we need to include these tables:



If you trace your way through the relationships identifying the primary key/foreign key links it helps you identify the join conditions in the WHERE clause. The link between the musician table and the lineup table is through the primary key of the musician table and the corresponding value in the foreign key in the lineup table. Successive

© Copyright the University of Oxford and contributors.

links are made in the same way until the final link, which consists of the primary key of the venue table and the corresponding value in the gig table. The statement is built up line by line until all tables are linked and is then completed by the final part of the WHERE clause indicating which row(s) of the joined tables is required:

```
SELECT firstNames, lastName, b.name, v.name
FROM musician m, lineup l, band b, spot s, gig g, venue v
WHERE m.musician_id = l.musician_id
AND l.band_id = b.band_id
AND b.band_id = s.band_id
AND s.gig_id = g.gig_id
AND g.venue_id = v.venue_id
AND v.name='Assembly Rooms'
```

firstNames	**lastName**	**name**	**Name**
Rick	Johnson	Hunkydory	Assembly Rooms
Lionel	King	Hunkydory	Assembly Rooms
Fred	Presley	Hunkydory	Assembly Rooms
Holly	Murphy	Hunkydory	Assembly Rooms
Rick	Johnson	Hunkydory	Assembly Rooms
Lionel	King	Hunkydory	Assembly Rooms
Fred	Presley	Hunkydory	Assembly Rooms
Holly	Murphy	Hunkydory	Assembly Rooms
Holly	Murphy	Spanish Nights	Assembly Rooms
Lionel	King	Spanish Nights	Assembly Rooms
Holly	Murphy	Spanish Nights	Assembly Rooms
Lionel	King	Spanish Nights	Assembly Rooms

Look carefully at this table. It illustrates another difference between relational algebra and SQL. An SQL query does not result in a relation but in a **data-set** (often called a **record set**), which is not bound by the rules that govern the underlying relations. If executing the query produces duplicate rows, they will be included in the data-set. If you do not want duplicate rows you can avoid them by using the keyword DISTINCT in the SELECT clause;

© Copyright the University of Oxford and contributors.

```
SELECT DISTINCT firstNames, lastName, b.name, v.name
FROM musician m, lineup l, band b, spot s, gig g, venue v
WHERE m.musician_id = l.musician_id
AND l.band_id = b.band_id
AND b.band_id = s.band_id
AND s.gig_id = g.gig_id
AND g.venue_id = v.venue_id
AND v.name='Assembly Rooms'
```

firstNames	lastName	name	name
Rick	Johnson	Hunkydory	Assembly Rooms
Lionel	King	Hunkydory	Assembly Rooms
Fred	Presley	Hunkydory	Assembly Rooms
Holly	Murphy	Hunkydory	Assembly Rooms
Holly	Murphy	Spanish Nights	Assembly Rooms
Lionel	King	Spanish Nights	Assembly Rooms

We have seen how to cope with ambiguous names in the SQL expressions, but what about ambiguity in the returned record sets? If we ask for a list of bands and venues we will get two columns, each headed 'name'. We can force the record set to use specified column names instead of those inherited from the tables, by using the AS keyword in the SELECT clause.

```
SELECT b.name AS Band, v.name AS Venue
FROM band b, spot s, gig g, venue v
WHERE b.band_id = s.band_id
AND s.gig_id = g.gig_id
AND g.venue_id = v.venue_id
AND g.gig_id=1
```

Band	Venue
Hunkydory	Assembly Rooms
Spanish Nights	Assembly Rooms

The order of the columns in a record set is the order in which the column names appear in the SELECT statement (except where the wildcard * is used when the order is inherited from the underlying tables).

© Copyright the University of Oxford and contributors.

The order of the rows can be manipulated to some extent by using the ORDER BY keyword. Simple alphabetical order on one field is achieved by:

```
SELECT *
 FROM venue
 ORDER BY name
```

venue_id	name	phone
1	Assembly Rooms	0194756
2	Dog and Parrot	01938475
3	Hard Rock Cafe	01928375
8	Park Amphitheatre	1234
7	The Cave	019283
5	The Dog and Gun	01928374
4	The Jazz Cafe	0192383764

If you need to order by two columns (say you had a data-set with regions and employees), you would separate the sort columns by a comma:

```
ORDER BY regionName,employeeName
```

If you want to sort values in descending order (or character types in reverse alpha order) you use the expression ORDER BY columnName DESC:

```
SELECT date
FROM gig
ORDER BY date DESC
```

You may remember from our review of relational algebra that there is also a Cartesian product that joins tables by linking each record in the first table to all records in the second table. The Cartesian product is implemented in SQL, where the syntax is the same as a theta join, except that there is no WHERE clause. I cannot imagine when you would want to use it, but it is worth knowing about. Check what happens if you use the command:

```
SELECT * from musician, band
```

If you did this using a database that had large numbers of records, the processing cost would be high and the output a very large table.

© Copyright the University of Oxford and contributors.

Before looking at other SQL commands, work through the following exercises to ensure you understand and can use the commands covered in this section.

SQL practice

We would like you to use the Gigs practice database to obtain data-sets similar to those shown below.

The database is live and may become changed so do not worry if the actual records are different from those shown, but you should get the same field headings. If you think you have the right SQL expression and you get the field headings but no data, please post a message about it in your tutor group forum so that we can check whether data has been lost.

You may need to debug your expressions before getting a successful result. If, despite all your efforts you are unable to obtain the required data-set, send a message to your tutor, enclosing the relevant SQL string.

In the following exercises, you are required to retrieve various data-sets from the Gigs practice database .

1. List the dates and venue name of all gigs at the Assembly Rooms.

date	Name
2004-03-21	Assembly Rooms
2004-04-21	Assembly Rooms
2004-01-01	Assembly Rooms
2004-03-06	Assembly Rooms

2. List the dates on which the band Hunkydory has been involved in gigs.

date
2004-03-21
2004-04-21

3. List the names and phone number of musicians who play with the band Hunkydory.

firstNames	lastName	Phone

© Copyright the University of Oxford and contributors.

Rick	Johnson	0204756
Lionel	King	020459
Fred	Presley	029029
Holly	Murphy	02789029

4. An enquirer has heard of a good venue but can only remember the name began with an H. Print out a list of all details of venue whose name starts with H.

venue_id	**name**	**Phone**
3	Hard Rock Cafe	01928375
10	Helios	0194765

5. Produce a list of bands and the venues where they have played. To make the output clearer make sure the relevant column headings show ‘Band’ and ‘Venue’ rather than both showing ‘name’.

If you get a table like this …

Band	**Venue**
Hunkydory	Assembly Rooms
Hunkydory	Assembly Rooms
High Kickers	Dog and Parrot
High Kickers	The Dog and Gun
Spanish Nights	Assembly Rooms
Spanish Nights	Assembly Rooms
Silver Birds	The Jazz Cafe

© Copyright the University of Oxford and contributors.

Krashers	Hard Rock Cafe

change the expression so that you avoid duplicate rows:

Band	**Venue**
Hunkydory	Assembly Rooms
High Kickers	Dog and Parrot
High Kickers	The Dog and Gun
Spanish Nights	Assembly Rooms
Silver Birds	The Jazz Cafe
Krashers	Hard Rock Cafe

6. List, in date order, the dates and venue names for all gigs that have taken place at the Dog and Parrot.

date	**name**
2004-04-21	Dog and Parrot
2005-03-06	Dog and Parrot
2005-04-18	Dog and Parrot
2006-03-06	Dog and Parrot

7. List the last name of the musicians and the band names of all who have performed at the Dog and Parrot.

lastName	**Band**	**Venue**
Johnson	Hunkydory	Dog and Parrot

© Copyright the University of Oxford and contributors.

King	Hunkydory	Dog and Parrot
Presley	Hunkydory	Dog and Parrot
Murphy	Hunkydory	Dog and Parrot
Murphy	Spanish Nights	Dog and Parrot
King	Spanish Nights	Dog and Parrot

8. List the dates, bands and venues for gigs that took place in the year 2005.

date	**Band**	**Venue**
2005-02-31	High Kickers	The Jazz Cafe
2005-04-18	The Cave	Dog and Parrot
2005-12-31	Hunkydory	Hard Rock Cafe

4.5. Inserting and amending data

SQL Data Manipulation Language (DML) includes statements to insert, amend and delete data.

New Records

The INSERT INTO statement adds a new record to the table. The syntax is:

```
INSERT INTO tableName (colName,colName)
VALUES('value',value)
```

For example:

```
INSERT INTO venue(name, phone)
VALUES ('Blue Steel', '0176487')
```

The first clause of the statement lists the columns to be updated (always omitting any auto increment field) and the second clause lists the values to be inserted in the columns.

An INSERT command does not need to contain all the fields in the table, but it must contain all fields that are constrained to be not null.

© Copyright the University of Oxford and contributors.

```
INSERT INTO gig(venue_id,date,time,cost)
VALUES (2,'2005-04-18','20:00',7.50)
```

In the above statement venue_id and date are required fields since they are constrained to be not null. You could omit the references to time and cost since they are allowed to be null.

The column names do not have be in the order in which they appear in the tables, but the values do need to match the order of the column names in the INSERT statement. The above command would work equally well if it were:

```
INSERT INTO gig(date, venue_id,cost,time)
VALUES ('2005-04-18',2,7.50,'20:00')
```

Where the table does not include an auto increment field and you are updating all the columns, you can, in practice, miss out the column names in the statement, for example:

```
INSERT INTO lineup
VALUES(2,3,'Bass Guitar')
```

If you omit the field names, of course, you must present the values in the order of the fields in the table.

Removing records

The DELETE command is similar to the SELECT command:

```
DELETE
from Venue
WHERE venue_id >15
```

Field names are not required in the first clause because DELETE only operates on whole rows. *The WHERE clause is very important*. If you omit it, all rows in the table will be deleted (usually without any 'Are you Sure' warning from the DBMS).

Pause for thought

Why do you think most DBMSs fail to warn you before deleting all the rows in a table?

Answer

Most SQL commands will be received from applications that interface with parametric users. (The interface application should have already carried out such a check.) Our practice database includes a simple dialogue that asks you confirm any delete operation.

© Copyright the University of Oxford and contributors.

Amending records

The UPDATE command allows you to change the content of named cells or fields in the table. As with the DELETE command, care is needed because if it is used without a WHERE clause it affects all the rows in the table. It is usually used to update a particular cell, for example:

```
UPDATE band
SET phone='0192834'
WHERE band_id=2
```

UPDATE is also used when there is a need to change the content of a field in a number of rows: for example, if there is change of duties of mentors you might need to amend the Student table with:

```
UPDATE student
SET mentor_id=4
WHERE mentor_id=2
```

Boundary changes may require a database to be updated with:

```
UPDATE customer
SET county = 'NewShire'
WHERE county = 'OldShire'
```

UPDATE is occasionally used without a WHERE clause to update all records: for example, one way of forcing a general change of password might be:

```
UPDATE user
SET pwd_expires = '2006/01/01'
```

Before learning more about SQL, work through the exercises below to make sure you understand and can use the commands covered in this section.

SQL practice

We would like you to use the Gigs practice database again.

You will be altering the database so please take care. Also make a careful note of any insertions you make so that you can delete them before you finish the exercise.

1. Insert a new entry into the musician table and use a SELECT expression to find the musician_id allocated.

2. Insert a new band into the band table and again find out the band_id allocated.

3. Use the ids you have noted to add the new musician to the new band.

4. Join the three tables in a SELECT expression to check the insertions were, in aggregate, as intended.

5. Amend the data in the table so that the telephone number *for the new entry you have made* becomes 9999.

© Copyright the University of Oxford and contributors.

6. Delete the entries you have made in each table. Make sure you start with the intersection table, because you may need to refer to the other two tables to obtain the relevant ids.

Make sure you use a 'WHERE' clause in your expression.

4.6. Data Definition Language

So far we have been looking at the SQL implementation of a Data Manipulation Language (DML), which we use to manipulate the enterprise data within the data structures. We now turn to the SQL **Data Definition Language (DDL)**, which is used to manipulate the data structures themselves. It is used to create, amend and destroy tables. It is also used to create views and constraints.

Although all relational databases support a core subset of the DDL, there are differences in the way other features are implemented and some RDBMSs are limited in the constraints that can be applied.

A new table is made using the CREATE TABLE command.

```
CREATE TABLE tableName
  (
    attributeName attributeType,
    attributeName attributeType,
  );
```

When setting the attribute type you can also enforce some constraints in line, for example a primary key or a not-null constraint:

```
CREATE TABLE module
  (
   module_id int auto_increment PRIMARY KEY,
   name varchar(100) NOT NULL,
   points int
  );
```

The primary key can also be set with a separate clause in the statement:

```
CREATE TABLE module
  (
   module_id int auto_increment,
   name varchar(100) NOT NULL,
   points int
   PRIMARY KEY (module_id)
  );
```

You can also use this approach to define foreign keys, for example:

© Copyright the University of Oxford and contributors.

```
CREATE TABLE module
  (
   module_id int auto_increment PRIMARY KEY,
   name varchar(100) NOT NULL,
   points int
   leader_id int,
   FOREIGN KEY (leader_id) REFERENCES tutor
  );
```

The foreign key constraint does not need to identify a field in the target table (here the tutor table), because the foreign key must always reference the primary key.

If you wish to use an alternative key you can implement it using the keyword UNIQUE:

```
CREATE TABLE employee
   (
    emp_id int auto_increment PRIMARY KEY,
    Soc_Sec_No varchar(12) UNIQUE,
   );
```

You can add a column to an existing table with:

```
ALTER TABLE module
ADD (startDate DATE);
```

and remove a column with:

```
ALTER TABLE module
DROP COLUMN (startDate);
```

To delete a whole table you use the expression DROP TABLE tableName:

```
DROP TABLE module;
```

Views are created using the CREATE VIEW statement. Suppose there is a Tutor table that contains information about a tutor's status, experience, grade, etc. We can restrict students' access to the information by requiring them to use a view of the table defined as:

```
CREATE VIEW studentView
AS SELECT firstName, lastName,roomNumber
     FROM tutors;
```

Views can also be created that restrict the rows of the table presented to the user:

```
CREATE VIEW fullCredits
AS SELECT *
     FROM modules
     WHERE points=30;
```

© Copyright the University of Oxford and contributors.

Most RDBMSs have commands that provides metadata, for example a list of the tables in the database or of fields in a table. You have already discovered how to do this in a mySQL database using the SHOW TABLES and DESCRIBE commands.

The creation of structural and enterprise constraints varies a little between RDBMSs, and you need to refer to the relevant manual, particularly for the imposition of referential integrity constraints. There is also some difference in the way systems deal with domains. The larger enterprise systems allow the CREATE DOMAIN statement, which closely mirrors the schema design we looked at in Topic 3, for example:

```
CREATE DOMAIN genderType
AS varchar(1)
CHECK value in ('F','M');
```

In most systems, if you are not able to create a domain as such, you can impose domain constraints in the definition of the table using another version of the CHECK clause, for example:

```
CREATE TABLE module
  (
   module_id int auto_increment PRIMARY KEY,
   name varchar(100) NOT NULL,
   points int CHECK (points >9 AND points >121)
  );
```

Note that this section is just an introduction to the Data Definition Language. If you become involved in the creation of enterprise databases, you will find that there are more sophisticated ways of creating tables and views and alternative ways of including constraints.

SQL practice

You will not be able create views and constraints in the practice database, but you can create and destroy a table, and add and remove fields.

1. Create a table with a unique name – I suggest you include your name in it. Include in the table an autonumber id field, a varchar field and a date field.
2. Insert into it a couple of rows of specimen data.
3. Check the data has been inserted.
4. Then, *carefully*, drop the table.
5. If you have any problems, check your syntax carefully. If you still cannot get it to work, post a message in the unit 3 forum. Someone may spot the problem, or it may be that the problem lies in the database and will be affecting others too.

© Copyright the University of Oxford and contributors.

4.7. Aggregation and group functions

Aggregation functions

SQL contains aggregation functions, which allow you to carry out operations on specified columns. You can ask for the number of values in the column (COUNT), or the sum (SUM), average (AVG), minimum (MIN) or maximum (MAX) value in the column. These functions operate on a single column and they return a single value.

COUNT returns the number of columns that have a value (that is, columns that are not null). The syntax is illustrated in this example:

```
SELECT COUNT(name)
FROM band
```

count(name)
5

SUM returns the sum of the column. Note that you can still use the AS keyword to specify your own column heading:

```
SELECT SUM(minutes) AS PlayingTime
FROM spot
WHERE gig_id =4
```

PlayingTime
85

The AVG, MIN and MAX functions are used in the same way. They can be used together. For example, if we had a table of employee salary rates we might ask for:

```
SELECT MIN(salary) AS lowest,
       MAX(salary) AS highest,
       AVG(salary) AS average,
       COUNT(salary) AS NumberOfEmployees
FROM salaries
```

lowest	highest	average	NumberOfEmployees
17000	58000	33600.0000	5

© Copyright the University of Oxford and contributors.

The COUNT, MAX and MIN functions can be used with numeric, character and date fields. AVG and SUM apply only to numeric fields.

Group functions

Aggregation functions are often used in conjunction with grouping. A GROUP BY clause aggregates the rows in a table by reference to the value of a specified column and produces a table containing rows which are summaries of each group.

When you use a GROUP BY clause, you are restricted in what you can include in your SELECT clause. You can only include the column used to perform the grouping, and aggregation functions (which may be based on any column). To put it another way, since the result of a GROUP BY clause is a grouping of data, it does not make sense to ask for any data that cannot be summarised. The example here is taken from the NorthWind sample database that comes with *Access*. Here is an extract from the Employees table:

Last Name	First Name	Title
Davolio	Nancy	Sales Representative
Fuller	Andrew	Vice President, Sales
Leverling	Janet	Sales Representative
Peacock	Margaret	Sales Representative
Buchanan	Steven	Sales Manager
Suyama	Michael	Sales Representative
King	Robert	Sales Representative
Callahan	Laura	Inside Sales Coordinator
Dodsworth	Anne	Sales Representative

I sent this command to the Jet DB Engine:

```
SELECT Title, COUNT(Title) as NumberOfEmployees
FROM Employees
GROUP BY Title
```

and the following was returned:

© Copyright the University of Oxford and contributors.

Title	NumberOfEmployees
Inside Sales Coordinator	1
Sales Manager	1
Sales Representative	6
Vice President, Sales	1

When a grouping has been made using the GROUP BY clause, you can restrict which groups are shown by using a HAVING clause. HAVING has the same syntax and effect as the WHERE clause but it acts on the grouped data. Suppose we only wanted to know numbers for categories where there was more than one employee. We could amend the SQL command to:

```
SELECT Title, COUNT(Title) as NumberOfEmployees
FROM Employees
GROUP BY Title
HAVING COUNT(Title) >1
```

Title	NumberOfEmployees
Sales Representative	6

Concatenation and arithmetic

Though they are not aggregation functions, here is useful place to look at the use of some other operators and functions on rows of the table. An operator that is frequently used in SQL statements is the concatenation operator. For example, a person's name will often be stored in two fields but is usually displayed in one. The concatenation operator or function varies between DBMSs, and you need to find out what it is. *Access* uses an operator (the ampersand symbol, for example. *firstNames & " " & lastName*) whereas mySQL uses a function, for example *concatenate(firstNames, " ",lastName.*) To obtain the full names of the people in the Musician table we would say:

```
SELECT CONCAT(firstNames," ",lastName) as name
FROM musician
```

name
Rick Johnson

© Copyright the University of Oxford and contributors.

Lionel King
Fred Presley
Holly Murphy
Chuck Quigley

You can also use arithmetic operators, which are more standardised. For example, if you had a table containing quantity and price, you could use the * (multiply) operator to produce a total cost column:

```
SELECT quantity, price, quantity * price as TotalCost
FROM orderitem
```

quantity	price	TotalCost
3	50	150
2	70	140
3	90	270

SQL practice

We would like you to use the Gigs practice database again.

1. Produce a list showing the genre categories and the number of bands in each category:

genre	number of bands
Classical	1
Indie	2
Pop	1
Rock	2

2. List the musicians' full names (in one field) and their phone numbers.

© Copyright the University of Oxford and contributors.

Name	phone
Rick Johnson	0204756
Lionel King	020459
Fred Presley	029029
Holly Murphy	02789029
Chuck Quigley	020485

3. A list showing for each band the aggregate amount of time (in minutes) of their spots at gigs.

   This is quite a difficult one. Think it through and write down the stages in less structured terms; then convert it into SQL.

name	Aggregate Appearance Time
High Kickers	80
Hunkydory	115
Krashers	40
Silver Birds	50
Spanish Nights	108
The Cavemen	40

4.8. Further SQL

This section consists of a miscellaneous group of SQL statements which, apart from the UNION command, often depend on a particular dialect of SQL.

We should like you to be able to use the UNION operator, but the other operators mentioned here are included only as background reading since they vary in the way they are expressed and handled by different RDBMSs.

© Copyright the University of Oxford and contributors.

Set operations

Many implementations of SQL permit set operations. The UNION operator, which we met in relational algebra, works in the same way in SQL. For example, the command:

```
select name from band
union
select name from venue
```

produces

name
Hunkydory
High Kickers
Krashers
Silver Birds
Spanish Nights
Assembly Rooms
Dog and Parrot
Hard Rock Cafe
The Jazz Cafe
The Dog and Gun
Blue Steel
The Cave
Park Amphitheatre

Many RDBMSs also support the INTERSECTION operation, though in SQL the keyword is INTERSECT.

© Copyright the University of Oxford and contributors.

Nested queries

The WHERE clause of an SQL command can itself be an SQL command. For example, the intersection operation can in some systems be achieved with:

```
SELECT name
FROM band
WHERE exists
(SELECT name FROM venue WHERE venue.name=band.name)
```

Outer joins

The syntax for outer joins differs between databases. Sometimes it is very similar to relational algebra. For example, in mySQL:

```
SELECT venue.name, gig.date
FROM venue LEFT OUTER JOIN gig
ON  gig.venue_id = venue.venue_id
```

produces

name	date
Omitted some	rows
Hard Rock Cafe	2004-05-21
Hard Rock Cafe	2004-09-23
The Jazz Cafe	2004-05-23
The Jazz Cafe	2004-09-22
The Jazz Cafe	2005-02-31
The Dog and Gun	2004-04-22
Blue Steel	
The Cave	
Park Amphitheatre	

© Copyright the University of Oxford and contributors.

The same syntax would be used in Microsoft SQL Server, but in other RDBMSs, for example Oracle, the command would be:

```
SELECT venue.name, gig.date
FROM venue LEFT OUTER JOIN gig
where  gig.venue_id = venue.venue_id(+)
```

4.9. Concurrency

Databases are shared resources, and they typically deal with many queries at the same time. The DBMS processes all current requests using a threading system, which means that the same data may be in use at any given time by more than one query. This could have serious consequences. Think of two otherwise unrelated queries each of which wants to modify a value, say to reduce the value of the in-stock attribute of a tuple in the Product table:

```
UPDATE product
SET inStock = inStock-5
WHERE product_id=23
```

```
UPDATE product
SET inStock= inStock-10
WHERE product_id=23
```

Assuming the amount of stock of product number 23 before these transactions was 100, the intended result is that stock is now 85.

In order to give effect to the first UPDATE statement the DBMS must read the value of inStock, carry out arithmetic on it, and then write the new value back to the database. If, in the meantime, it started processing another thread containing the second statement the result could be:

```
Statement 1 - read inStock value  (100)
Statement 2 - read inStock value  (100)
Statement 1 - reduce the read value by 5
Statement 1 - write the new value of inStock (95)
Statement 2 - reduce the read value by 10
Statement 2 - write the new value of inStock (90)
```

What has happened here is that the **serialisation** of the processes has been lost. They should have been processed in sequence but because of the multitasking design of the DBMS, they have been processed in parallel. To avoid this problem, database management systems implement virtual serialisation of processes by using **locks**.

When a thread accesses data a lock is applied to the data. A lock may be a **write lock** or a **read lock**, where the write lock is typically more stringent than a read lock. Locks also vary in their **granularity**, that is, they can be applied to the whole table, or tables, involved in the query; to a page or block of data; to a few contiguous rows in the table; or to the single row that contains data. Locking is not something associated with an SQL statement: many databases allow choices about the kind of locks applied but they are set by the Database Administrator and the settings apply to all processes. There are many forms of locking technology and they can involve complex algorithms. In essence, however, locking takes one the following forms.

© Copyright the University of Oxford and contributors.

Pessimistic locking

This approach to locking assumes that conflicts are likely. Under this protocol a process intending to write something obtains a write lock from the start of the process, which prevents any other thread from accessing the data at all until the process is completed. Where a process has obtained a read lock, other processes are allowed a read lock but no process is allowed a write lock. This kind of locking is easy to implement and guarantees that conflicts will not occur, but it is only suitable when concurrent usage is low. If there are many users, or if the transactions are large and complex, pessimistic locking causes an unacceptable drop in performance.

Optimistic locking

This approach assumes heavy use or complex queries but assumes that the chances of an actual collision are small. Concurrent processes will usually be accessing different data items so conflicts are in practice unlikely to arise. Under this protocol the locks are applied for much shorter periods. A process intending to amend an item applies a read lock, reads the item, and it then releases the lock, having flagged the item as being subject to an operation. Having done the necessary processing, it again reads the data to check whether any other process has flagged the item in the meantime as involved in a process. If not, a write lock is obtained, the write completed and the flag removed. If on the second read there is another in-process flag, the whole operation will be abandoned and re-started. (In practice the position is more complex and there are collision resolution strategies that may mean the process is not simply abandoned.)

Overly optimistic locking

In this system the locking lasts only during the actual read or write process, and no attempt is made to detect collisions. It is appropriate in single-user systems but very dangerous if used in even small multi-user systems.

Database transactions

So far we have considered the conflicts that can arise because of loss of serialisation of a single statement. There is a further danger where multiple statements carry out connected operations. Consider the following related statements:

UPDATE myAccount SET balance = balance –100

UPDATE yourAccount SET balance = balance + 100

If something had gone wrong after the first statement completed but before the second statement completed, the intended transaction would not have taken place, and the database would hold inaccurate information. In effect these two statements form a single transaction. Most DBMSs are able to monitor the whole transaction and ensure that it exhibits what are called ACID characteristics. The ACID acronym is derived from the qualities: Atomicity, Consistency, Isolation and Durability.

Atomicity
Either the execution of the whole transaction must be completed *or* the database must be left as if it had never started. It is either all or nothing. If there is a crash or conflict the processes so far must be rolled back.

Consistency
This is really another way of saying the same thing. A transaction *either* creates a new and valid state of data *or*, if any failure occurs, returns all data to its state before the transaction was started.

Isolation

© Copyright the University of Oxford and contributors.

Transactions must behave as if they were completely isolated from each other. This is usually implemented by the DBMS through the use of the locking mechanisms discussed above.

Durability
Committed data is saved by the system in such a way that, even if there is a failure and system re-start, the data is available in its correct state. This may be achieved by immediate clearing of buffers or, more usually, by transactions logs.

The level of transaction support depends on the DBMS and, sometimes, on the internal structure of the data. For example if you use mySQL you can choose from a number of internal structures, only some of which support transactions.

A database may default to having **autocommit** switched on or off. (The practice database we use is set to autocommit.) Where it is on by default, the database will automatically treat only single SQL statements as transactions. The default status can be changed using the commands:

```
SET AUTOCOMMIT OFF

SET AUTOCOMMIT ON
```

If autocommit is *off*, an SQL statement will not be committed to the database unless it is ended with the statement:

```
COMMIT;
```

In practice, databases are often left with autocommit off, because it is possible to override the default behaviour for any particular transaction using the statement BEGIN. After this command, nothing will be committed to the database until a COMMIT command is received, so the transaction we looked at above would become:

```
BEGIN;
UPDATE myAccount SET balance = balance-100;
UPDATE yourAccount SET balance = balance + 100;
COMMIT;
```

If both processes succeed, they become permanent. If the second fails, the state of the database will be rolled back to how it was before the first process was executed.

As well as automatic rollback in the event of database failure, the same commands can be used by the operator or by an application to cancel a previously submitted command. For example, the UPDATE commands in this transaction would be processed by the DBMS but the values of the relevant attributes would remain unchanged:

```
BEGIN;
UPDATE myAccount SET balance = balance-100;
UPDATE yourAccount SET balance = balance + 120;
ROLLBACK;
```

© Copyright the University of Oxford and contributors.

4.10. Data Control Language

Although it was not part of the original specification, Data Control Language (**DCL**) is now a standard way of passing commands to an RDBMS in order to set rules about the levels of access (usually called privileges) of users and groups. We do not go into this subject in any detail and we do not have any practical work associated with this section. It is, however, worth knowing in outline the kind of language used to control access.

The main commands are self-explanatory:

```
GRANT
REVOKE
DENY
```

They can be applied to individuals or roles that individuals fill in the use of the system. For example:

```
GRANT ALL PRIVILEGES ON sales TO managers
GRANT SELECT, INSERT ON sales TO Bill
REVOKE ALL PRIVILEGES ON sales FROM  Jim
```

If a privilege is granted using the key phrase WITH GRANT OPTION, it means that the user can grant the same privileges over the same table to others, for example:

```
GRANT SELECT, INSERT ON sales TO Bill WITH GRANT OPTION
```

means that Bill can give others permission to see and insert records in the sales table (but not to do anything else).

Databases usually have built-in roles. For example, they all have **public** (which defaults to no privileges) and **db_owner** (which defaults to full privileges.) A new role can be create using the CREATE ROLE statement:

```
CREATE ROLE musician
GRANT SELECT, INSERT ON band TO musician
```

And using the GRANT command you can give individual users the privileges accorded to a role:

```
GRANT musician to Brett, Greg, Fiona
```

There is more to DCL, and if you become involved in administering an enterprise database you will have to pursue the topic further, but for the purposes of this unit the above is all you need to know about it.

4.11. Self-assessed questions

More SQL practice

The best way to learn SQL is to use it to interrogate a database. Here are some more exercises that you will be able to try out using the on-line databases.

© Copyright the University of Oxford and contributors.

Since you can try out, and debug, your expressions using the live database we think you will eventually manage to solve these problems yourself and we have not provided a key. If you do find yourself unable to solve any of them, however, you can ask your tutor to send you the relevant solution(s).

We provide example tables for most of the questions but remember that, though your tables should contain the same fields, the data in the rows may be different because of subsequent insertions and deletions.

Connect first to the College database.

1. Produce a single list of names (in alphabetical order) of all people in the system (that is, of tutors and students).
2. We need to check that the allocation to houses is even. Produce a list showing the house name and the number of students in the house.

blue	2
green	3
red	3
yellow	3

3. There is a suggestion that people in the Yellow house have a different pattern of module choice than other students. Produce a list showing for each enrolment the house name and the module name (in house order).

house	**Module**
blue	Visual Basic
blue	Databases
blue	Java
green	Databases
green	Visual Basic
green	Java
Red	Java

© Copyright the University of Oxford and contributors.

Red	Databases
Red	Visual Basic
yellow	Visual Basic
yellow	Web Design
yellow	Visual Basic
yellow	Databases

4. Show modules on which there are enrolments and the number of students enrolled on each module.

Module	**Number Enrolled**
Databases	4
Java	3
Visual Basic	5
Web Design	1

5. List the students who have registered for one or more courses and the total points for the modules they are studying.

Student	**Total Points**
Banks George	30
Brown John	60
Clark Jim	30
Farmer George	15

© Copyright the University of Oxford and contributors.

Field Helen	30
Hall Henry	15
Kent Fiona	30
King Mark	30
Robson Joan	45
Singh Amit	15

6. A fire in room 101 has destroyed all the assignments. Assuming the scripts were in the rooms the module lecturers occupy, list the students that are most likely to be affected.

Student	**Module**
John Brown	Java
Jim Clark	Java
Helen Field	Java
John Brown	Java
Jim Clark	Java
Helen Field	Java
John Brown	Databases
Joan Robson	Databases
Mark King	Databases
George Banks	Databases

7. For the next two questions, change to the Gigs database.

© Copyright the University of Oxford and contributors.

The Assembly Rooms have discovered some kit left behind by a drummer. They are not sure when it happened but it was sometime in March 2004.

Produce a list of musicians who may own the kit.

At the time I tried the query, the answer was:

firstNames	**lastName**
Fred	Presley
Lionel	King

8. Display, in descending order of number of appearances, the names of bands who have been booked on spots and in each case the number of appearances.

Name	**Number of Appearances**
Krashers	10
The Cavemen	10
Silver Birds	10
Spanish Nights	9
High Kickers	8
Hunkydory	8

9. Create a new table to hold your favourite books. Many students will be using the same database so you need to choose a table name that is likely to be unique. In most cases incorporating your name will be enough (*buckleybooks*) but you may need to add an initial or number to be sure. These are the fields to include.

Book_id	**title**	**Author**	**ISBN**
1	Google Hacks	Calishain & Dornfest	0596004478

Make book_id an auto_increment primary key (you may have books without an ISBN). Make the other fields varchar fields.

© Copyright the University of Oxford and contributors.

Insert a row of data into your table, then use a SELECT expression to produce output like the above.

Then, reluctantly, but this is necessary, destroy all your good work by dropping the table.

4.12. References

If you use your advanced research skills you will find some excellent SQL tutorials on the Web. At the time of writing there was an electronic book, *Stuctured Query Language (SQL): A Practical Introduction* (in pdf format).

© Copyright the University of Oxford and contributors.

Unit 5. Implementing database solutions

5.1. Implementing database applications

So far in this unit we have looked at the design, definition and creation of databases and we have manipulated a database using a simple SQL command interface. We have been able to concentrate on the core subject without having to worry about programming paradigms and environments.

Most databases are, however, used by applications that provide customised interfaces for parametric users, and those applications are written in a variety of languages on a variety of platforms.

In Topic 1 we recalled how a legacy application would declare and use a language-specific data structure, then save that data structure in the same logical form on backing store. It would do so using the file management protocols of the chosen programming language and operating system. Where the database approach is used the application must ask another application (the DBMS) to store and retrieve data and it needs to be able to communicate with it.

In this topic we look at ways of connecting applications with databases and we illustrate the theory using practical examples. We do not expect you to do this practical work yourself, though we would *encourage* you explore the topics covered. The code examples are often isolated extracts provided to illustrate the principles of connectivity. We hope you will find them helpful but we do not want you become sidetracked into worrying about the detail. The learning objective of this topic is to achieve an understanding of the broad principles of connectivity. We hope the programming examples will help you achieve this, but if they do not, you can ignore them.

If you become involved in the implementation or use of enterprise systems in a large organisation it is likely you will use a very powerful Relational Database Management System (RDBMS) like Oracle, DB2, or SQLServer, and will need to familiarise yourself with the system's own interfaces. Since the principles involved are the same for all systems our examples will use more accessible RDBMS packages like the open source mySQL package and the MSJet/MSDE database engines within the Windows operating system. (We do also cover briefly the Express editions of SQL Server and Oracle and if you wish to do so you can download these applications and experiment with them on your local machine.)

By the end of this topic you should be able to:

- describe the purpose and use of Open Database Connectivity (ODBC) drivers
- describe database connectivity in the context of the Java development kits
- describe database connectivity in the context of Microsoft Visual Studio
- describe the way server side scripts connect to and use a database
- explain and evaluate the operation of a database driven web site
- describe and evaluate the development of database applications using Microsoft Access

© Copyright the University of Oxford and contributors.

5.2. Database connectivity

A database management system acts as a server with applications acting as clients. This is sometimes hidden from the user. Database management systems come with a range of facilities including GUIs which enable users to create and manipulate databases, but these are all provided by client applications. It may all happen within the same apparent application (Microsoft Access is a good example) but the facilities are nevertheless provided by clients that need to make contact with the DBMS server. Even at the very basic command-line level, there is this client/server relationship. Here is how I might use the mySQL DBMS on my home PC. I start the DBMS by opening a command window and typing in the name of the server program: *mysqld*.

There are two things to note here:

- Although I have the mySQL software on my machine it is not a DBMS until it runs. I have to start it running before I can use it (though I could, of course, install it as a Windows Service or include it in the start-up folder so that it would run automatically on boot-up).
- After entering `mysqld` the system command prompt returns. There is no database prompt or other suggestion that I can use mySQL. Though the DBMS *server application* is now running I cannot communicate with it without loading a *client application*.

The client application I use here is provided in the mySQL package, and involves typing in a command window `mysql -u userName -p password`. This time the prompt changes to the mySQL client application prompt and I can now access a database:

```
mysql> use jdbc_example;
Database changed
mysql> create table demoTable (id int not null primary key, name varchar(100));
Query OK, 0 rows affected (0.14 sec)
mysql> insert into demoTable (id,name) values(23,'Oxford');
Query OK, 1 row affected (0.03 sec)
mysql> select * from demoTable;
+----+--------+
| id | name |
+----+--------+
| 23 | Oxford |
+----+--------+
1 row in set (0.00 sec)
```

As it is provided by the mySQL package this simple client application knows how to make a connection to the server and I do not need to worry about how it does it. If I were to write a new front-end application, however, I should need to know how to make a connection from it to the mySQL server.

Most DBMSs have similar dedicated clients that can be used 'out of the box' to manage databases. But once you move away from the dedicated applications, you need to be able make a connection between your client application and the DBMS server.

Although it is possible to hard code any application to interface with a particular DBMS, that would involve the costs and risks of the additional programming and would anyway defeat some of the key advantages of using a database.

© Copyright the University of Oxford and contributors.

Pause for thought

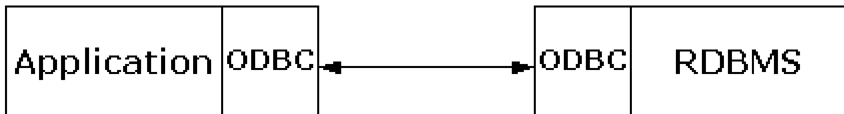
What principles of the database approach might be put at risk by this kind of hard-coding approach?

Answer

It would make the application less flexible (it could not easily use a different database management system) and might even threaten data independence.

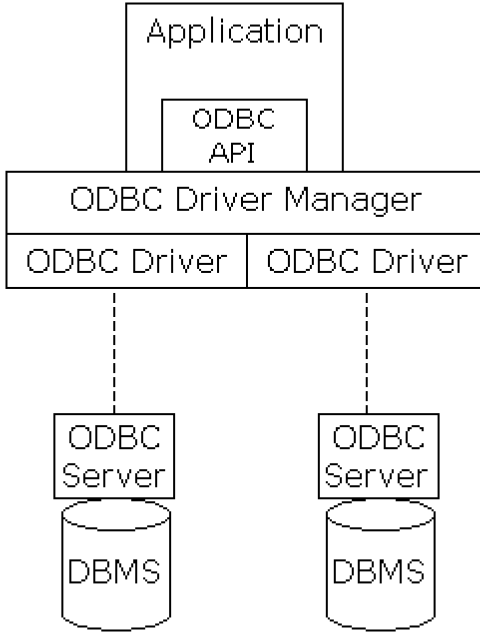
In order to avoid this, some kind of middleware is needed to standardise the interface between applications and databases.

In pursuing this objective Microsoft pioneered the Open Database Connectivity (ODBC) standard. Alhough it was introduced by Microsoft, it is an open standard and has become the de facto standard for relational database systems. It provides a common interface for accessing any relational database. ODBC is a kind of middleware, but the 'middle' is not meant in spatial terms but in conceptual terms. It comes between the application and the database, but parts of it are located at each end.



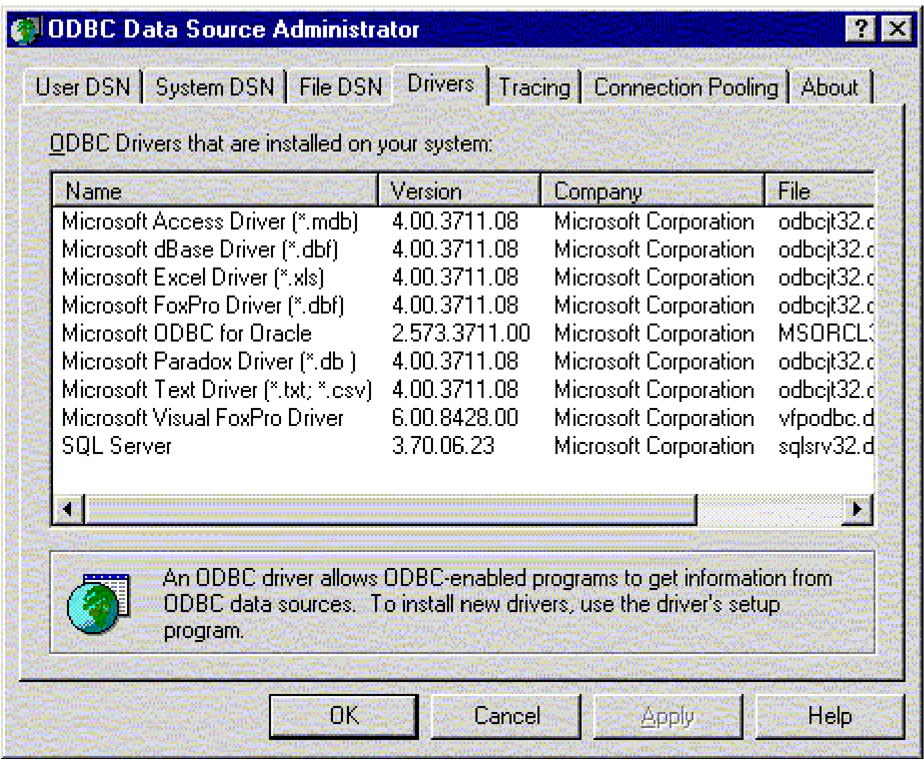
You can imagine this in terms of an operating system and a printer. There are international standards that determine how characters are represented and so forth, which are followed by printers and operating systems but to use a particular printer it is of course necessary to load a printer driver. The same is true when using ODBC to connect to a database:

© Copyright the University of Oxford and contributors.



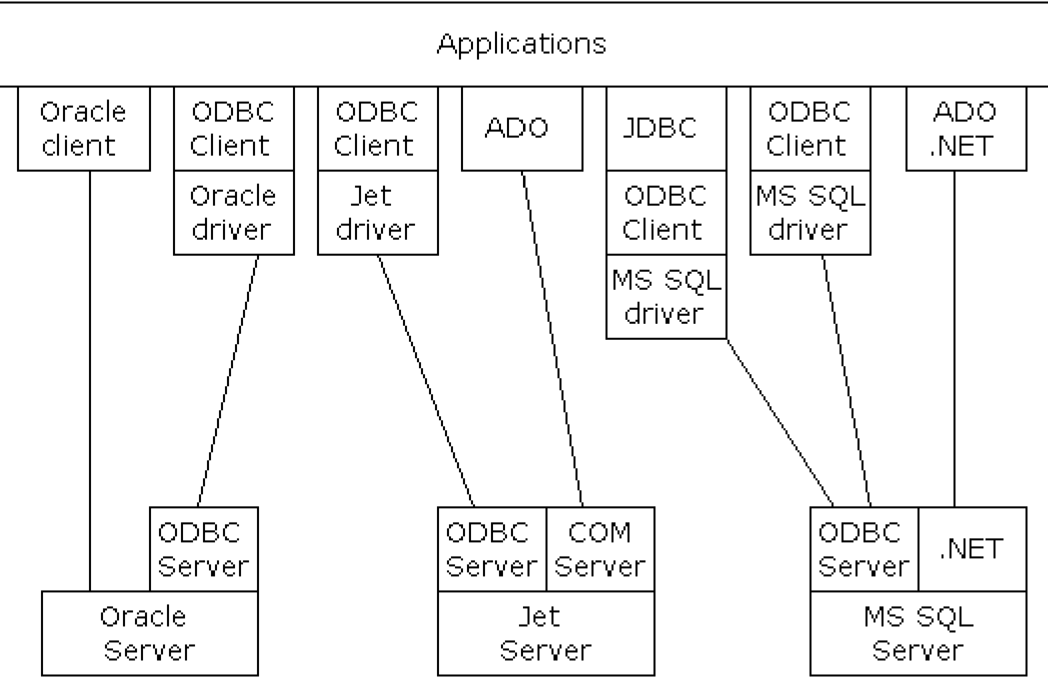
For example you might write an application that is intended to run on Jet (Access) in one context and on Oracle in another. In theory all you would need to change would be the ODBC driver. (The qualification 'in theory' is significant because there remain some troublesome differences in the date formats expected by different DBMSs.)

ODBC drivers can be incorporated into the application itself but they are more commonly incorporated into the operating system. If you are using Microsoft Windows you will have several installed on your machine. You will probably be able to see them if you look in Control Panel -- Administrative Tools -- Data Sources. If you cannot find this option, do not worry: occasionally the operating system is loaded without them. The display looks like this:



If you need a new driver you purchase it from a vendor or, in many cases, download a free open source driver. Drivers are usually packaged in an install program.

Although ODBC is a widely used technology there are other connectivity models that are often used instead of or in conjunction with it. In particular the Java development kits contain Java Database Connectivity (JDBC) classes. Also when programming on the Microsoft platform, connections are often made through ActiveX Data Objects (ADOs) which are part of the Component Object Model or the .NET framework. We will look at some of these in the following sections, but for the moment here is a broad overview. Do not try to learn or even understand everything at this stage, since it is the broad picture you need here. It will be worth returning to this diagram at the end of the topic when you have studied some of the technologies mentioned and gained more knowledge about them.



5.3. Relational database management systems

When you come to use a DBMS you will need to spend time reading the manuals and may well need to undertake training and certification in the chosen product. It is not our intention in this unit to look at particular systems in any detail. We use mySQL in most of our examples because that is the system you can most readily use to explore interfacing a wide range of applications to a database. The principles are, however, the same for any DBMS and in case you want to explore other systems yourself we provide notes on the free Express versions of Oracle and SQL Server. Please note, however, that we cannot provide support for these systems, beyond discussing issues with you in the Forum.

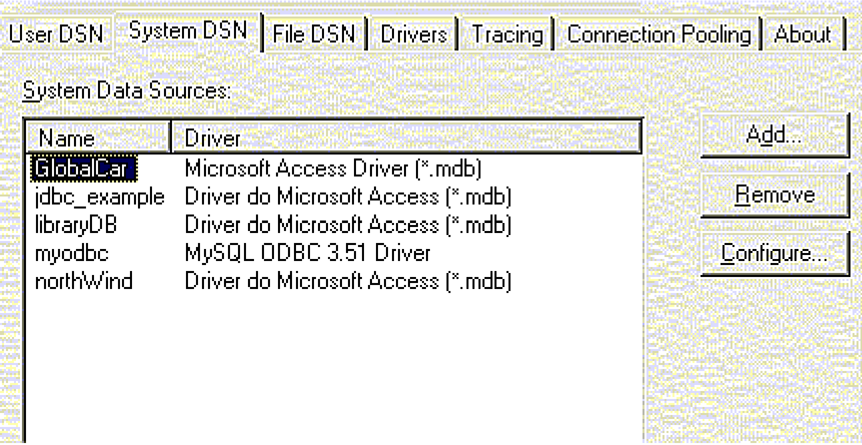
© Copyright the University of Oxford and contributors.

mySQL

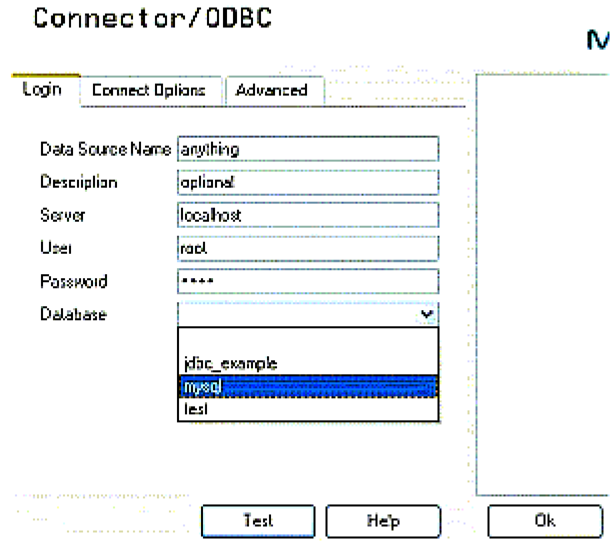
This is a true client/server DBMS which performs well under load. It is very popular and is used quite widely. It is open source, relatively easy to install and there are ODBC drivers for it. The chief reason for highlighting mySQL, however, is that most budget web hosting companies include a mySQL database and the ability to run scripts to access it.

MySQL can be obtained from [MySQL Community Downloads] as can a comprehensive manual and installation instructions. From the same source you can obtain a mySQL ODBC driver. If you search the web you will find articles and tutorials on setting up and using the product. MySQL provides a database server which has to be started before the DBMS can be used. On Windows machines, however, it is usually installed as a Windows Service that starts automatically and runs continually in the background.

In order to connect to a mySQL database from an application you need to set up the relevant ODBC connection. The simplest way to do this on a Windows machine is to set up a Data Source Name (DSN). In the ODBC dialog (the one we looked at in Section 5.2) there is a page tabbed System DSN:



Clicking Add brings up a list of ODBC drivers which, assuming you have installed it, will include the mySQL driver. Selecting that driver brings up a dialog in which you enter the username, password and database name:



Once set up in this way you can connect to the chosen database by referring to its Data Source Name in the application program. (We will look at some examples of how this is done in Java and Visual Basic in later sections). Once you are familiar with the basic approach you can explore other ways of setting up ODBC that do not require the inclusion of passwords in the DSN.

Microsoft Jet

This is the database engine that is shipped with Microsoft Access but it is a separate product and most Windows systems will have it installed. f you connect to Jet using a Microsoft programming environment you can create ActiveX Data Objects (ADO) which will connect to a Jet driven database. (We look at ADO later in this topic).

ODBC drivers are available for the Jet engine and are usually already installed in Microsoft operating systems. If you want to use a Jet database in a programming language that does not support ADO, say in a Java program, you can set up a Jet ODBC Data Source much as we did above for mySQL. When setting up the Jet Data Source you are able to create a new database, which means you can use the same type of database as used in Access even if you do not have Access installed on your machine. We create a database in this way later in the topic when we illustrate accessing a database from Java. Jet is a file server engine which runs as part of the operating system and so you do not have to install it as a service or run it. As soon as you connect to an .mdb file, you are running in the Jet engine. The downside of this approach is that it soon slows down or goes wrong if you have more than about 20 concurrent users. For small business applications Jet, whether through Access or a dedicated application, is very effective and is the most widely used. Access uses a new file format (.accdb) but still uses the Jet engine.

Microsoft Database Engine (MSDE)

This is a cut down version of Microsoft's enterprise DBMS (SQLServer). It is not usually installed in Windows operating systems but it is available on the installation discs. The main advantage of using MSDE is that is easy to scale up to the Microsoft SQLServer enterprise product. It uses the same dialect of SQL and operates as a true client server database. It has performance limitations programmed into it and is designed for prototyping rather than production; but it is a very useful bridge when moving from Jet to SQLServer.

MSDE was an important development tool prior to the release by Microsoft of the Express edition of SQL Server (see below) which is now the preferred development environment for scalable systems.

Postgres SQL

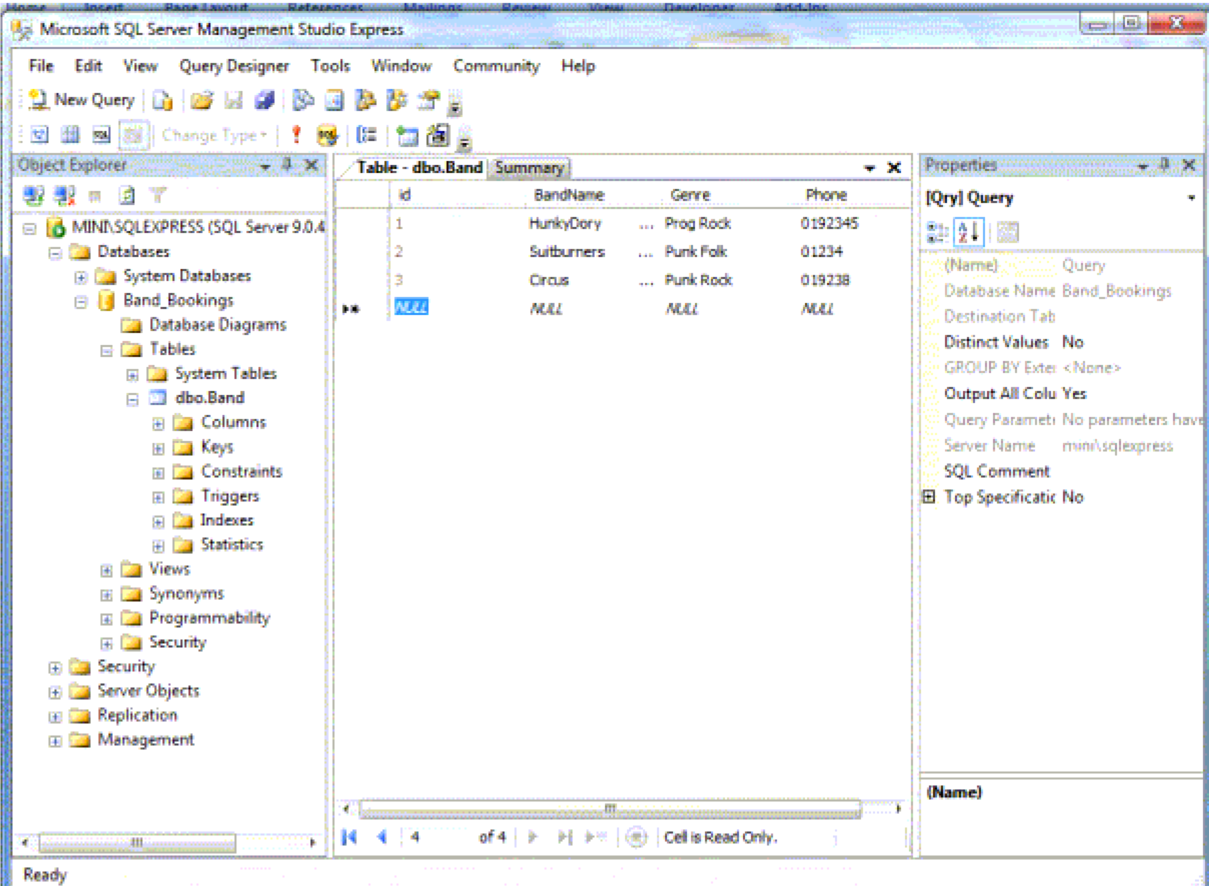
This DBMS is derived from the POSTGRES package written at the University of California at Berkeley. It is a very powerful client/server object relational database. It is, I think, the most advanced open-source DBMS available. Although it is now possible to run PostgresSQL on Windows machines it was developed for Unix platforms and that is its chief user base. It is more difficult to set up and administer than mySQL and is only rarely provided in basic web hosting packages.

Microsoft SQL Server

SQL Server is the Microsoft flagship DBMS. It is costly to install and its operation and maintenance involves technical and management resources. It can cope with very large databases and large numbers of users. It also has many built in applications and functions enabling it to support Web Services, distributed databases and data warehousing.

© Copyright the University of Oxford and contributors.

It is, however, possible to obtain a free version (SQL Server Express) which contains most of the features of the full system and is an ideal environment for developing a system which needs to be scalable. SQL Server Express, and its Management Studio application, can be downloaded from the Microsoft site. As you are running a true DBMS, you need to set the system running before you can access a database. You can start it from Programs as you do any other application, but it is simpler and more trouble free to set the system up as a Windows Service. The install package gives you the option to do this. Here is a screen shot of the management application.



Oracle

The Oracle Corporation specialises in database and database related software. The latest Oracle DBMS is much more than a DBMS incorporating its own Java Virtual Machine and the ability to run user programs on the server. It has object oriented capabilities and extensive facilities for data warehousing. As with SQL Server there is a price to pay for the product's power and functionality, not only in its initial acquisition but also in the costs of its management.

Oracle also provides a free Express version of the system, which can be downloaded, together with a management application, from the Oracle site. Like SQL Server it is best run as a service.

DB2

DB2 is the IBM flagship database for enterprise applications. It is highly customisable and can include a wide range of advanced features. It is the database used in many e-commerce applications driven by the IBM Websphere application server. Again, it is a very powerful product but one which requires a lot of investment in development and training as well as in acquisition, and needs to be managed by an administrator with experience and product knowledge.

© Copyright the University of Oxford and contributors.

5.4. Working in Java

Although the design and implementation of the database is a central part of most projects, it is far from the whole story. You also need applications to provide the business logic and to interface with the users, and these will be developed in an appropriate programming environment. In this section we look at how a database can be integrated into a project developed using one of the Java developments kits.

The Java development kits include classes that provide ODBC-like connection objects called Java Database Connectivity (JDBC) objects. If you are accessing a DBMS which has a JDBC interface and you have the relevant JDBC driver available to the client application, you can access the database using the JDBC.

From their introduction, however, the JDBC classes included a JDBC:ODBC bridge which enables Java applications to access databases through existing ODBC clients and drivers. Since ODBC drivers for the more common databases are already present in most Windows machines the use of this bridging technology is very common. The simplest way of setting up a Java application to access a database is to create a Data Source Name (DSN) in the ODBC set-up dialogue. We have met the DSN earlier when we were discussing mySQL. The way in which you would setup the JDBC:ODBC bridge would depend on the operating system in use and we do not cover it in detail, here. What follows assumes that the relevant JDBC:ODBC drivers have been installed.

The Java development kit includes the SQL package that contains the classes needed to connect and use databases. You can get access to these classes by importing:

```
import java.sql.*;
```

A connection via the JDBC:ODBC bridge is initiated by loading the class that provides the JDBC client application interface:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Do not worry about the odd expression or the complexity of its argument. It simply means load the code which will set up the JDBC client and the ODBC bridge. The JDBC.ODBC client in itself does not connect us to a database. We need to make a Connection object which connects through the JDBC:ODBC bridge to the required database. When you use Data Source Names the process is relatively straightforward. You use the getConnection method of the DriverManager class like this:

```
Connection cn = DriverManager.getConnection("DSN Name","","");
```

The DSN Name is the name you gave the DSN when you set it up in Control Panel. (The empty Strings in the arguments are placeholders for the username and password when you use more complex JDBC:ODBC connections.)

Once the connection is made, communicating with the database involves creating and executing Statement objects:

```
Statement st = cn.createStatement();
```

This creates an object in the right form for processing through ODBC. We then provide the statement with an SQL expression and an execute command:

© Copyright the University of Oxford and contributors.

```
st.execute("The SQL expression");
```

There are slight variations depending on whether or not you need to get something back from the database. Here is a very simple program which connects to an empty database, creates a new table in it, inserts a row of data and then obtains a record set of the new table:

```
import java.sql.*;
  public class DB_connect
  {
    public static void main(String[] args)
    {
     Connection cn;
     ResultSet rs;
     Statement st;
     String sqlstr;
     String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
     String dbURL="jdbc:odbc:jdbc_example";
       // or if we were using the mySQL DSN we would use the name
       // we gave when we created it, e.g.  dbURL="jdbc:odbc:myODBC";
     try
       {
       //load jdbc bridge
       Class.forName(jdbcDriver);
       // load the data source
       cn = DriverManager.getConnection(dbURL,"","");
       sqlstr="CREATE TABLE band(name text, phone text)";
       st = cn.createStatement();
       st.execute(sqlstr);
       sqlstr="INSERT INTO band (name, phone) VALUES('Hunkydory','0191345')";
       st.execute(sqlstr);
       sqlstr="SELECT * FROM band";
       rs=st.executeQuery(sqlstr);
       System.out.println("Name: " + rs.getString("name")+"Phone: "+ rs.getString("phone"));
       }
    catch(Exception e)
      {
        System.out.println("Could not execute command because " + e);
      }
    }
}
```

The output from the program when run with either DSN is:

```
Name: Hunkydory Phone: 0191345
```

In a real application you would, of course, construct a GUI and would need to use loops to move through the ResultSet as different records were selected, but the basic idea would be the same. Once you have set up a DSN you simply send SQL expressions and get back ResultSet objects.

© Copyright the University of Oxford and contributors.

5.5. Working in Microsoft Visual Studio

The information in this section is illustrative and is intended to give an overview of the way application programming builds connections with relational databases. The precise commands used to connect to SQLServer and Access databases differ depending on the version of Windows and Visual Studio in use. There is no practical work in this topic of the unit, and there is no need to go beyond the generic examples provided here. If, however, you nevertheless wish to carry out experiments in MS Studio please first make a relevant web search (e.g. “vb connect to sqlserver”) and select a link to Microsoft to ensure you have up to date information about the commands needed.

The examples in this section are in Visual Basic, because that tends to be more readable to non-programmers, but the System objects and connection strings would be the same in C#, and the run-time would be identical.

The .NET framework contains built-in database connectivity, the loading and configuring of ODBC drivers, or of dedicated Microsoft Active X objects, being undertaken transparently by objects of classes in the System.Data package. By saying something simple, like:

```
connection= New SqlConnection(connectionString)
```

you can create a Connection object that encapsulates the relevant connectivity behaviour.

The .NET framework also contains classes that handle the data once it is retrieved from the database. These include data adapters which allow users to work on a local copy of the data, with updates to the database being done only when a session is completed, or at intervals during a long session. Where there is a lot of network traffic or a slow network it can be an advantage to work on a local copy, and the data adapter will automatically throw exceptions if the relevant tables in the database have been changed since the local copy was made. These features are not universally used, however, and may not always be the best solution. The delay in updating the database increases the likelihood of the database being changed in the meantime, and the use of a complex ‘wizard’ to update your data makes debugging problems more difficult. We will not be covering these additional features here. You can, however, use classes in the System.Data package to create and use a simple database connection in much the same way you would using simple ODBC, or JDBC.

The following example shows a very basic connection to a SQL Server database.

First import the package that contains the relevant system classes:

```
Imports System.Data.SqlClient
```

Create a code module and declare the instance variables we will use:

```
Module SQLServerTest
    Dim myConnection As SqlConnection
    Dim myCommand, As SqlCommand
    Dim dr As SqlDataReader
Sub Main()
```

Create a String holding details of the DBMS and the database:

```
connectionString = " Data Source = MINI\SQLEXPRESS; Initial Catalog=Band_Bookings;Integrated
Security=True"
```

© Copyright the University of Oxford and contributors.

Make a new SqlConnection object, connected to the database:

```
myConnection = New SqlConnection(connectionString)
myConnection.Open()
```

Send an SQL expression, putting the data returned in a data reader:

```
myCommand  = New SqlCommand("Select * from dbo.band", myConnection)
dr = myCommand.ExecuteReader()
```

Write the data to the console:

```
    Console.WriteLine("---------------------------------")
    Console.WriteLine("List of bands")
    Console.WriteLine("---------------------------------")
     While dr.Read()
            Console.WriteLine("id" & vbTab & dr("id"))
            Console.WriteLine("name" & vbTab & dr("BandName"))
            Console.WriteLine("genre" & vbTab & dr("Genre"))
            Console.WriteLine("phone" & vbTab & dr("Phone"))
            Console.WriteLine("---------------------------------")
 End While
End Sub
```

When the program is run, the following records are retrieved from the SQL Server database and are printed to the screen:

```
List of bands
-------------------------
id      1
name    HunkyDory
genre   Prog Rock
phone   0192345
-------------------------
id      2
name    Suitburners
genre   Punk Folk
phone   01234
-------------------------
id      9
name    Circus
genre   Pop
phone   01923
-------------------------
```

If we then use different SQL expressions in the command object:

```
myCommand = New SqlCommand("Insert into dbo.band (BandName,Genre,Phone)
values('Kicks','Pop','01923')", myConnection)
myCommand.ExecuteNonQuery()

myCommand = New SqlCommand("delete from dbo.band where BandName='Circus'", myConnection)
myCommand.ExecuteNonQuery()
```

we can add or delete records. After carrying out the above fragment of code, the original code was rerun to provide the following output:

© Copyright the University of Oxford and contributors.

```
List of bands
---------------------------
id       1
name     HunkyDory
genre    Prog Rock
phone    0192345
---------------------------
id       2
name     Suitburners
genre    Punk Folk
phone    01234
---------------------------
id       10
name     Kicks
genre    Pop
phone    01923
---------------------------
```

If you were using a database implemented in Microsoft Access the coding would be identical except that you would import the System.Data.OleDb package and would create a new OleDbConnection object, with a connection string something like:

```
connectionString = "provider=Microsoft.Jet.OLEDB.4.0;" & "data source = Band_Bookings”
```

If you were using Oracle the coding would be identical except that you would import the System.Data.OracleClient package and would create a new OracleConnection object, with a connection string something like:

```
connectionString = "user id='hr';password='hr'; data source= 'MINI'"
```

In practice you would, of course, use forms to display the data and code to move between records, and there is rather more to implementing a database application in .NET than the bare bones covered here. But we hope you can see that the basic creation and use of a database connection in .NET is quite straightforward.

5.6. Databases and web sites

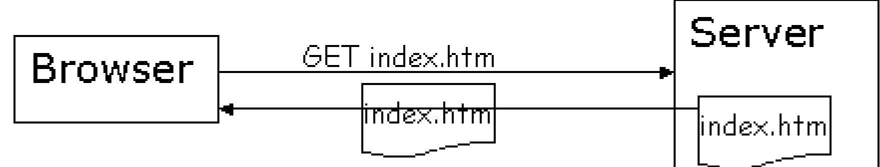
First a note about terminology. A phrase widely used for the generic dynamic web site is '*active server pages*' but this is also the name of the Microsoft server-side scripting environment. When I am talking about the generic active server pages I will use lower case, and upper case for the Microsoft Active Server Pages (ASP).

Databases have been used in conjunction with web pages since the early days of the commercial exploitation of the World Wide Web. They were used by sites that provide information services, and by sites undertaking transactions with customers. As these services and enterprises have grown so has the incidence of database linked web pages. In recent years, however, there has been a huge growth in the use of databases because of a change in the kind of web site demanded by small and medium sized enterprises (SMEs). In the mid to late 1990s many SMEs took the plunge and paid to set up their own web site, many of them aided by grants from government organisations like Business Links. They were initially happy with the sites but soon realised that they needed updating from time to time, or in some cases at frequent intervals, and the updating could only be carried out by a skilled practitioner. Whether done in-house or by an agency the costs of this maintenance were substantial and many businesses decided to take down their sites or opt for a simple 'presence' site.

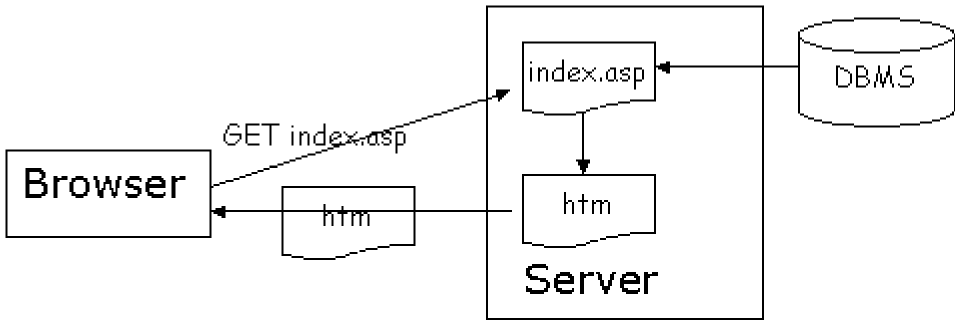
Today it is possible to produce web sites that can be maintained by existing office staff who have received very little extra training. This is achieved by storing the content of the site in a database and using active server technology to generate web pages as required. If an organisation adds a new product line, it can be entered into the database using a simple form in a web page. A news item to customers can be added in the same way.

© Copyright the University of Oxford and contributors.

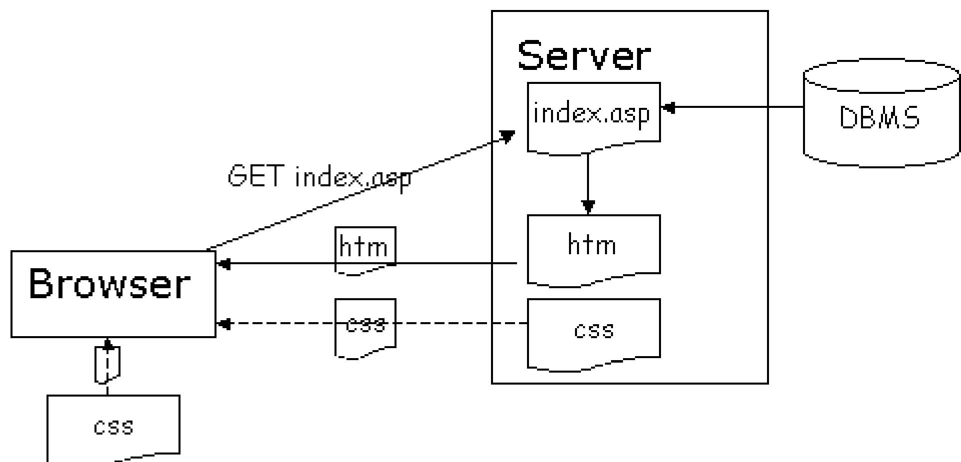
Active server technology is part of the operating system of the Web server which is able to run code in program scripts and to create new HTML pages as required. When a request is made to a passive server for an HTML page the server finds the page and returns it to the browser:



When the browser requests a file from an active server, the requested file, for example index.asp or index.php, is not returned to the browser. Instead the server runs the code in the file and, usually, writes a new page in HTML and returns that to the browser. There is clearly an opportunity in this chain of events to insert into the newly created web page some information obtained from a database.



This is typical of contemporary software systems. There is a DBMS which manages the data, a business tier which manages the processing of the data and a presentation tier or layer which interacts with the user. The complexity of the presentation layer varies but it is usually not a very complicated piece of software. Lightweight presentation layers, or thin clients, are increasingly common in database systems. The precise way in which the presentation layer displays data in such a system can be very flexible indeed. In many systems the thin client is a browser and the layout of the web pages is determined by stylesheets. The server will send the default stylesheet, along with the HTML page but the local browser (or other software for handling HTML) can be configured so that it loads a local stylesheet instead.



Thus the same HTML file provided by the server may appear as a complex display with images, a text only display or even be presented in a Braille reader. The advantages of this three tier approach together with the ease with which scripts and stylesheets can be programmed is changing the way much software is designed and used. Systems that previously had a complex presentation later, written in a conventional programming language, are increasingly using browsers as thin clients with all the processing carried out on an intranet server and displays governed by stylesheets. Where the highest levels of security are not required it is possible to tailor both the pages available and the stylesheets used depending on individual log-in id and password.

A large commercial site may use an enterprise database like Oracle or SQL Server with server applications running in the Java Virtual Machine or the NET Framework. It is the sort of system we examined earlier in the topic: there are applications programs which connect, through JDBC, ODBC, ADO etc., to the DBMS. Such systems may also connect with other corporate systems, for example accounting systems and enterprise resource planning systems. Major software suppliers have generic systems that can be customised to provide the required functionality, (For example IBM offer the *WebSphere* technology which provides the basis for a wide range of e-commerce applications). Nevertheless, the planning and implementation of enterprise web sites involves practitioners who are trained and experienced in the relevant applications - it is a major investment.

For smaller scale operations, however, setting up a database driven web site is relatively straightforward and inexpensive.

The Microsoft Internet Information Server includes the Jet database engine and can handle ActiveX Data Objects (ADO). A script written in the Microsoft scripting language ASP can create ADO and manipulate an Access database that has been uploaded to the server. An ASP script is similar to a Visual Basic module with some specific server related functions. For example the expression:

```
Request.Form("controName")
```

can be used to obtain the information entered into the controls of an HTML form, and:

```
Response.Write("Some html code here")
```

can be used to add text and pictures to the HTML page which will be returned to the browser.

The connection to the Jet database engine using ADO is made in the same way the connection was made in the Visual Basic program we saw earlier in the topic. The ASP script would create an ActiveX Data Object with a line like:

```
Set Cn = Server.CreateObject("ADODB.Connection")
```

It would then would open the Connection object cn by using its Open method with the Jet data source as a parameter:

```
Cn.Open("Provider=Microsoft.Jet.OLEDB.4.0;" & "Data Source=" & Server.MapPath("../../../db/
college.mdb") & ";" & "Persist Security Info=False")
```

Here again, do not let the complexity of these connection strings worry you. Once you have started using the technology you will always have somewhere from which you can copy and paste most of the code.

© Copyright the University of Oxford and contributors.

The script can now obtain a record set and extract from it the rows and columns needed to write the HTML page. (How it does this is beyond the scope of this unit, though we do walk through the coding of part of an active site which uses the php scripting language in the next section.)

The most popular server/database combination for small organisations and individuals is the *Apache server* running the mySQL DBMS. Any scripting language can be used to connect to mySQL through ODBC, but the PHP scripting language is by far the most widely used in this area. It is open source software and is usually provided as part of web hosting packages. (PHP was the acronym of a more modest precursor, the 'Personal Home Page' application. The acronym has been retained despite radical changes in its purpose and power, and it is now an abbreviation of 'PHP Hypertext Pre-processor'). PHP has built-in functions to connect to the mySQL DBMS and to open and manipulate databases. The PHP equivalent of the ASP code shown above would be:

```
mysql_connect("localhost", "userName", "password");
mysql_select_db("college")
```

Again, do not try to go deeply into this here. We will look at these expressions again in the next section. The important thing is that we have a scripting language that makes it very easy to use a DBMS that is running on our server.

If the site is running on Microsoft Internet Information Server, and it has all the relevant dynamic link libraries loaded, it is possible for 'power users' to create database-linked web pages without knowledge of ODBC or scripting languages. A wizard in Microsoft Access builds Data Access Pages which are HTML-like documents (in fact mostly in XML) which allow interaction with tables in an Access database. ead-only forms work well, even with linked tables and cascaded subforms, and they are fairly easy to set up. Pages that add or amend data can be problematical when the Data Access Page is based on linked tables. Data Access Pages provide a valuable resource for advanced office users but they are not suitable for a serious software project. They do not achieve anything that could not be achieved using ASP or PHP, they rely heavily on extra layers of server software, and they are subject to limitations in functionality.

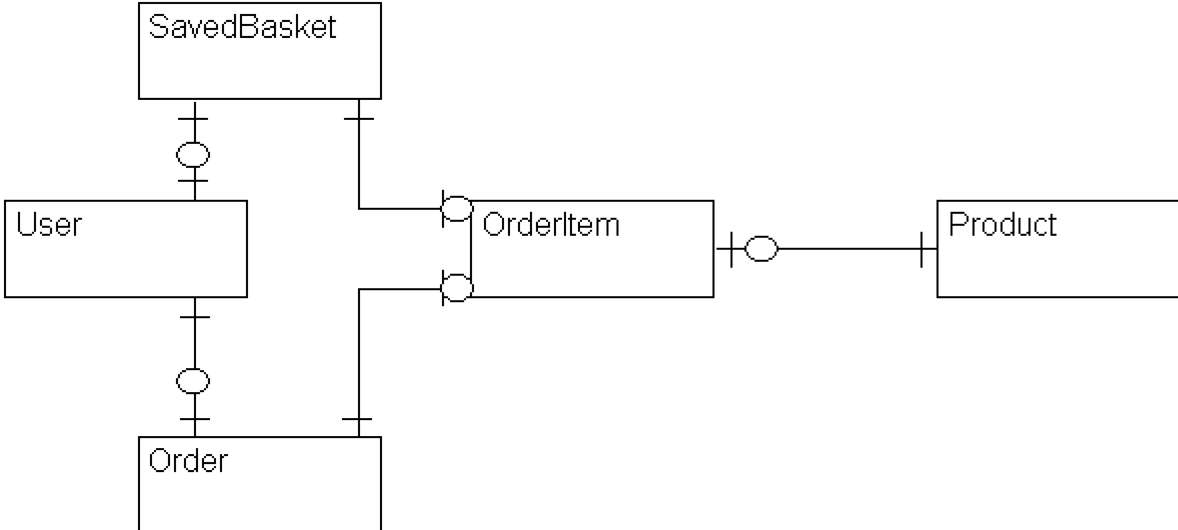
5.7. Web application example

In this section we look at an example web site which serves pages containing information drawn from a database, and examine the techniques used to construct it. The example is used to highlight database technology. We do not cover web authoring in this unit, and give only sufficient excerpts to suggest how the database is used to serve the web pages.

The site is one used by students studying web design courses, and lacks much of the design and functionality of a commercial e-commerce site. It does, however, enable you to appreciate how this kind of database application works.

The model for the database is:

© Copyright the University of Oxford and contributors.



In this overview, however, we will just be looking at the transactions in the Product table.

Before continuing with this section, open the example practice site.

Use the site first as a casual user, viewing and ordering products. (There is no danger you can actually order anything or damage the database: this application is incomplete and does not actually record your orders). Register (using any name and password and a fake email). Note that even though you are a registered user, you are still not able to access the pages that add or edit product details. Log in using *admin* as both the user name and password. You now have access to the Add Product page, which will allow you to add an image and details of a new product. Make a new entry (you can upload any jpg, gif or png file) then check to see that it is included in the Products page. When you have finished investigating this function please use the Edit Product page to delete your product (unless you think there is some reason - artistic merit or humour – to leave it there.)

The starting point for this kind of site is a host that provides a server that runs a scripting language and a DBMS. We could have used other configurations, for example a Microsoft Internet Information Server running ASP scripts to connect to SQL Server, but in this example we have chosen to use an Apache server running PHP scripts to connect to a mySQL DMBS. This particular system is easy to acquire and is not expensive. (You can obtain hosted facilities for as little as $40 a year.)

Before we turn to the script for the example site, recall the way active servers work.

When a page is requested from a dynamic web site the server does not return the page requested, but instead runs the code within it. This means that between receiving a request from a browser and returning a web page to it, we can open our database and retrieve from it data to include in the web page.

Since we know we will be using a database the first thing we have to include in the script is something to set up the client-server arrangement with the DBMS. If you are using PHP as your scripting language you do not need to worry about setting up an ODBC connection. The PHP engine contains library files one of which contains the driver code to connect to a mySQL database. All you have to do is call the relevant PHP functions. So near the beginning of each PHP page there will be code to connect to the mySQL DMBS:

```
mysql_connect("localhost", "userName", "password");
```

and code to connect to the relevant database:

© Copyright the University of Oxford and contributors.

```
mysql_select_db("dataBaseName");
```

With a live connection and a database in place we can now use SQL to manipulate the database. When a category (here, Widgets,) is selected on the Products page, a GET message is sent by the browser to the server:

```
http://kwesi.conted.ox.ac.uk/xxxx/ bg/products.php?cat=wg
```

The server has an array of variables holding the values of any parameters sent with a GET command. In this case it will hold the value 'wg' in the variable $_GET['cat']. The script can therefore obtain the required product details using an SQL query like:

```
SELECT  id, description , price , inStock
FROM product
WHERE cat='$_GET[cat]'
```

This query is assigned to a variable ($sqlString) and a the details obtained using code like:

```
result = mysql_query($sqlString);
```

The variable $result then holds an array of values – in effect a product table containing the relevant rows.

The table is iterated using:

```
while($myrow=mysql_fetch_array($result) )
  {
              Make an html row containing
              myrow['description']   myrow['price']   (myrow['inStock']
  }
```

This produces an HTML page that lists the products on the screen. There is some more HTML code wrapped around the product description so that it produces a link to the showProducts.php script, sending as a GET parameter the id of the product.

When a product name is clicked the browser sends a message like:

```
http://kwesi.conted.ox.ac.uk/xxxx/ bg/showProduct.php?id=12
```

The script on the server responds by sending to the database an SQL string., which is used by the PHP script to obtain the relevant record from the database. The code used is similar to the following (in practice it has more quotation marks and other symbols in it):

```
result = mysql_query("SELECT * FROM product WHERE id=$_GET['id']");
```

This time there is only one row in the table, but we still need to extract it so we again use:

```
$myrow=mysql_fetch_row($result)
```

to get the data for the selected row.

© Copyright the University of Oxford and contributors.

The position of the details on the returned page are determined by the style sheet, so there is in fact more HTML code here, but in essence the picture, for example, is included in the returned page with code like:

```
echo “<img src=”. $myrow[‘image’].”>”
```

So far the communication between client and server has been by using parameters in the URL. When we add something to the database we use a different approach. When an HTML form is submitted to a server, the browser sends with it the values in each of the controls of the form. This is held by the server in an array called $_POST[ ]. So, for example, if you complete a form entering “Jim” in the *name* control and “enigma” in the *password* control, on the server, $_POST[‘name’] will contain “Jim” and $_POST[‘password’] will contain “enigma”.

This is what happens when the Add Product form is submitted. There is an intermediate stage, in which the image file is uploaded, then the data base is updated using the values in the $_POST[] array. It tends to be a long string, and involves a lot of quotation marks, but here is the essence of it:

```
INSERT INTO product
(cat, image, description)
VALUES ($_POST[‘cat’],$_POST[‘imageFile’], $_POST[‘description’
```

5.8. Microsoft Access

You do not need to have Microsoft Access to study this section. The illustrations we provide show the kind of interfaces the developer is able to use to implement tables and views and that is all that is required for this unit. If, however, you have the Access software and are not already familiar with its use, you may like to follow the walk-through of form creation in the exercise at the end of the section.

Microsoft Access is widely used by office 'power users' and computer enthusiasts to create simple database applications. Its ease of use and powerful design tools enable people with good IT skills to produce simple working solutions even though they have little background in computer science. But it is not limited to simple applications. Professional application developers use Access for the creation of prototypes, and for small business systems it is often the system of choice for the final version. It works best in standalone mode, though it can be set up for shared access over a network provided the number of users is small and speed of response is not critical.

Access combines the server and client functions into one package. You can create tables, views and user interfaces using simple graphical user interfaces and wizards. This is one of the features that make it so suitable for prototyping. Where the eventual solution is going to be in some other database the overhead in creating the database and application interfaces can be considerable, and it should not be built until the design is completed and agreed. It is, however, often useful to have an early indication of the final product so that its functionality can be discussed with the client. Access enables a developer to create a rapidly developed prototype as a 'proof of concept' to demonstrate what the final solution will look like.

For stand-alone applications, or in small office networks, Access can create powerful and reasonably secure database applications. It can cope with a lot of data - Microsoft quote a maximum file size of 2 GB and a maximum table size of 1 GB. It can be set up to operate user-level security, effectively restricting individual users and groups to defined views (possibly read-only) of the data. Developers can purchase a license to include a run-time version of Access with their products, enabling them to use Access to produce stand-alone solutions.

© Copyright the University of Oxford and contributors.

Tables can be created using SQL, but it is at best difficult to program constraints this way and most people use the visual development screens. There are even easier ways - table wizards - but developers usually create tables in the Table Design View, which looks like this:

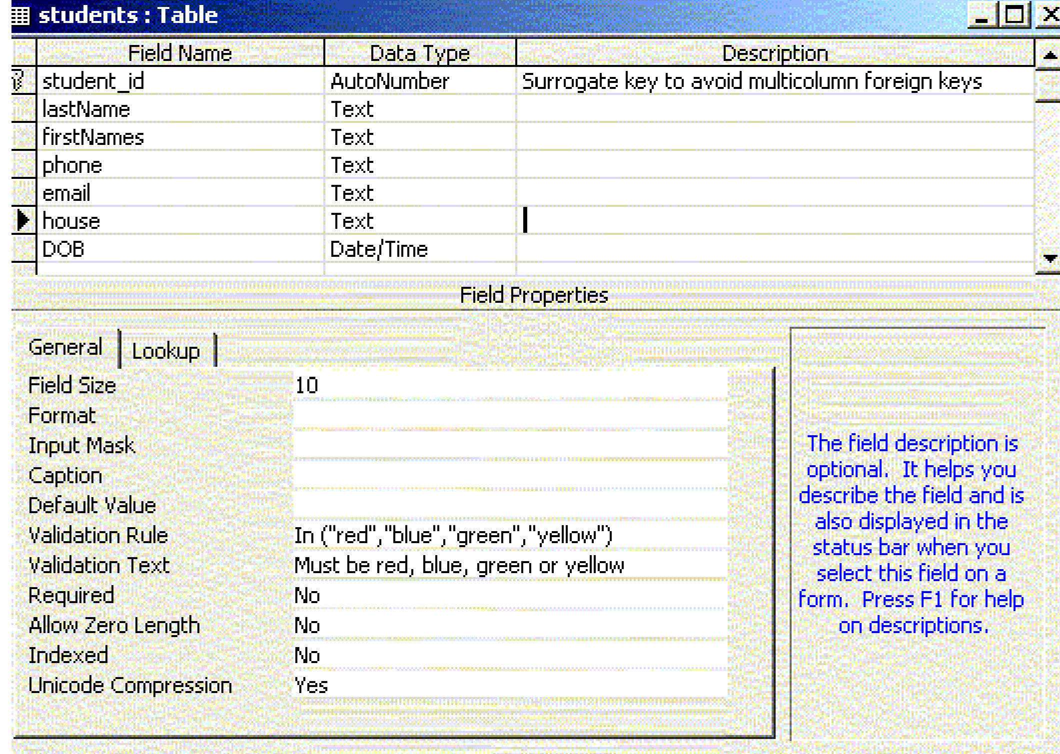


Table Design View

Although there is no separate domain definition in Access you can effectively impose domains in the field properties. In the image above the row containing the house attribute is highlighted and the field properties for house indicate the domain of values for the field. The field properties can also include NOT NULL (though here the word is Required), default values and an input mask. (An input mask can be used to ensure, for example, that in a new record the date field is displayed as __/__/____ and that no other format is accepted.) The key in the button to the left of student_id constrains that field as the primary key of the table. When you are working in tables there is a toggle button at the extreme left of the Access button bar and this enables to change rapidly between the above design view and the following datasheet view:

students : Table

student_id	lastName	firstNames	phone	email	house
1	Johnson	Brian	01245375	brian@somewhere.com	Yellow
2	Green	Arthur	1345893	art@crafty.co.uk	Blue
3	King	Henry	02345	hal@string.com	Red
4	Lockwood	Susan	0134567	sue@theDrive.org	Yellow
5	Butler	George	0143256778	jed@oklahoma3com	Blue
6	Robson	Andrew	01234534	andy@pandyTeddy.org	Blue
7	Sinclair	Elizabeth	0143285986	liz@electriccars.com	Red
8	Charlton	Marion	012345634	marion@thisISP234.co.	Green
9	King	John	0123456	john@magnacarta.org	Red
(AutoNumber)					

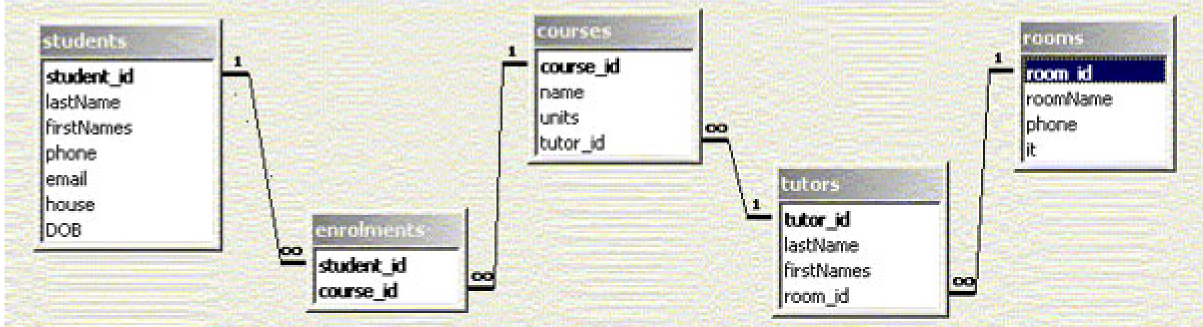
Record: 1 of 9

Table Datasheet View

© Copyright the University of Oxford and contributors.

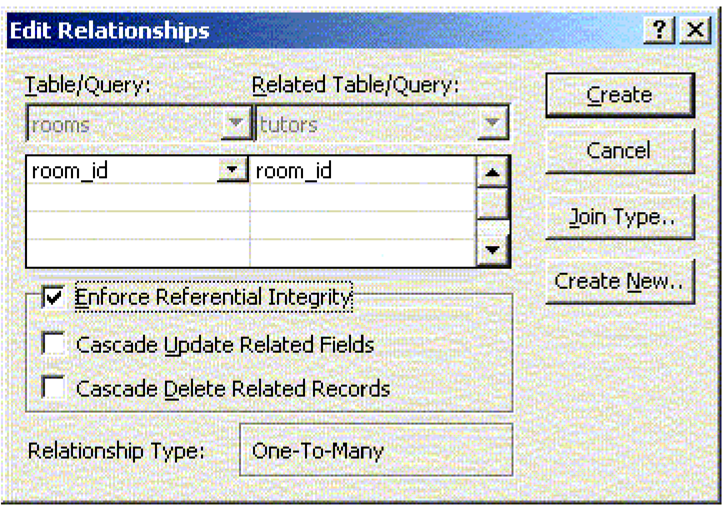
If you examine business software produced in Access you will probably find the tables and other objects named using the Leszynski Naming Convention which involves prefacing each name with a three letter abbreviation of the type of object, for example tblStudents, frmTutors or qryRooms. This naming convention is often used within the Microsoft community, but is not widely used elsewhere. You often find plural nouns are used for table names (as we have here) in Access applications. This is something you would not normally see in tables created in other systems.

Once tables have been designed you can get a graphical display of them, and set referential integrity rules using Database Tools which provides a window in which you can display all the tables:



Relationships manager

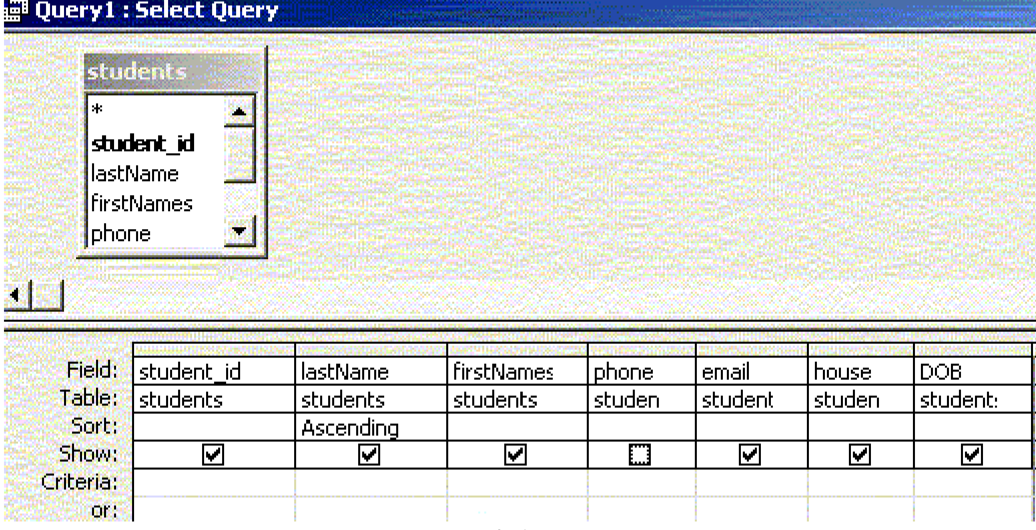
When you drag the join lines between the tables a dialog appears which enables you to set referential integrity and, if required, cascading of delete and update actions on primary keys:



Setting referential integrity

Most developers work from views rather than tables. A view can be automatically sorted each time it is used and can include calculated fields which would be redundant in the table. Views in Access are called queries and are produced using another GUI in which you drag and drop fields into a grid:

© Copyright the University of Oxford and contributors.



Query design screen

This query produces the same datasheet as the table, except that the rows are sorted by lastName:

student_id	lastName	firstNames	email	house
5	Butler	George	jed@oklahoma3corr	Blue
8	Charlton	Marion	marion@thisISP234	Green
2	Green	Arthur	art@crafty.co.uk	Blue
1	Johnson	Brian	brian@somewhere.c	Yellow
9	King	John	john@magnacarta.o	Red
3	King	Henry	hal@string.com	Red
4	Lockwood	Susan	sue@theDrive.org	Yellow
6	Robson	Andrew	andy@pandyTeddy.	Blue
7	Sinclair	Elizabeth	liz@electriccars.cor	Red

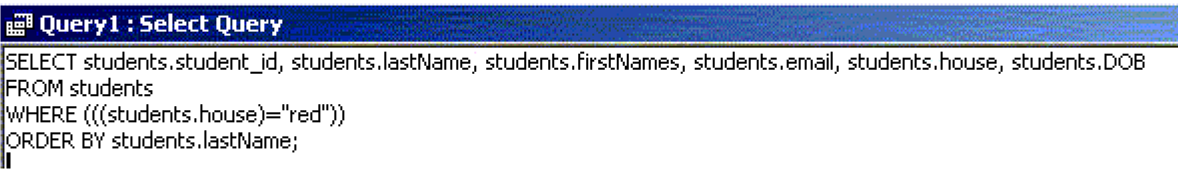
Datasheet view of the query (1)

The criteria row in the query grid is where you enter the criteria to be used to filter the dataset. In the above query if we had entered ‘red’ in the cell where the Criteria Row crosses the house column, the datasheet would have shown:

student_id	lastName	firstNames	email	house
9	King	John	john@magnacarta.o	Red
3	King	Henry	hal@string.com	Red
7	Sinclair	Elizabeth	liz@electriccars.cor	Red

The house field criteria set to ‘red’

If, with this query open you were to select SQL View from the View Menu, you would see ...


Query1 : Select Query

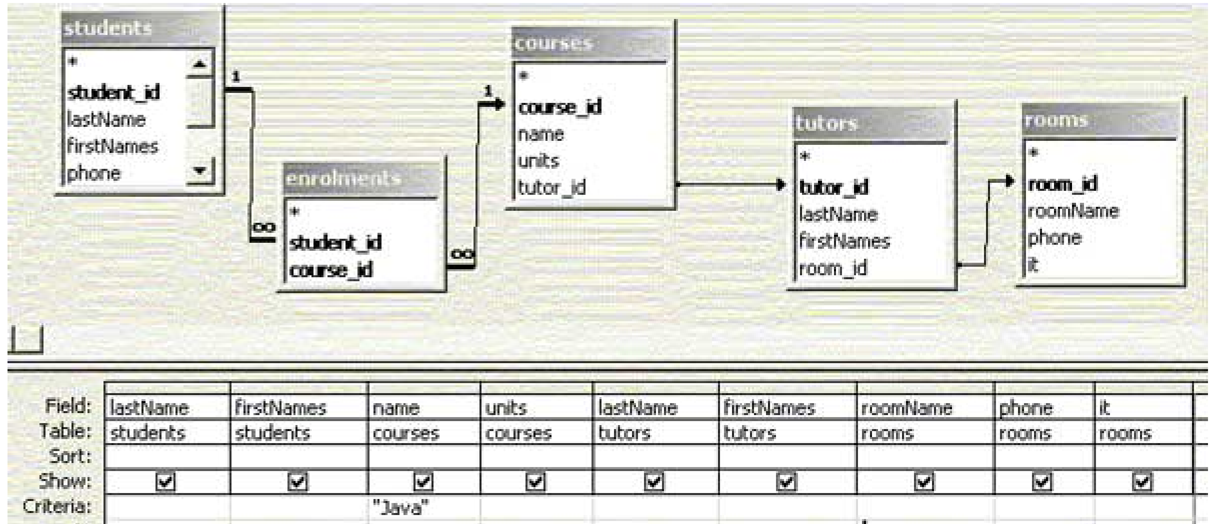
```
SELECT students.student_id, students.lastName, students.firstNames, students.email, students.house, students.DOB
FROM students
WHERE (((students.house)="red"))
ORDER BY students.lastName;
```

SQL view of the query

... which will, I hope, look familiar.

The Access query interface can be used to join tables, for example:

© Copyright the University of Oxford and contributors.



Query design screen joining 5 tables

produces the datasheet:

_qryTestJoins : Select Query

students.	studen	name	units	tutors	tutors.fi	roomName	phone	it
Johnson	Brian	Java	30	Hill	Vince	A100	4567	☑
Green	Arthur	Java	30	Hill	Vince	A100	4567	☑

Datasheet view of the query (2)

which could be used directly or might be used as the dataset for a form. The database approach envisages applications asking for a given dataset and leaving it to the RDBMS to join together the required tables. In practice table joins are sometimes undertaken in the application program and, since the client and server are so closely coupled in Access, it is particularly prevalent in Access applications. Forms are constructed on single tables, and the forms are then linked within the application.

Forms in Access can be created rapidly by dragging and dropping form widgets and data fields. The best way of learning to use forms (and indeed Access generally) is to experiment. If you have the Microsoft Access application we encourage you to work through the following exercise, which is a walk-through of one way of designing a form based interface. (If you do not have Access it will still be worth reading through the exercise to gain insight into how Access works.)

The exercise is based on the scenario used in the illustrations on this page. The database stores information about students, courses and tutors (ignore the rooms in this exercise). A student may take more than one course. A course can only have one tutor, though tutors can teach several courses. We want an interface that will allow us to view for each student a list of courses, and for each course a list of students.

From this scenario draw up an entity relationship model (which should not take very long) and implement it in Access tables. When you have created all the tables and shown their joins in the Relationships window, open Forms_Exercise.pdf [Link not available in this medium, please view it in the online course.] and compare your design with the version provided. Then follow the walkthrough to create a user interface for the tables.

© Copyright the University of Oxford and contributors.

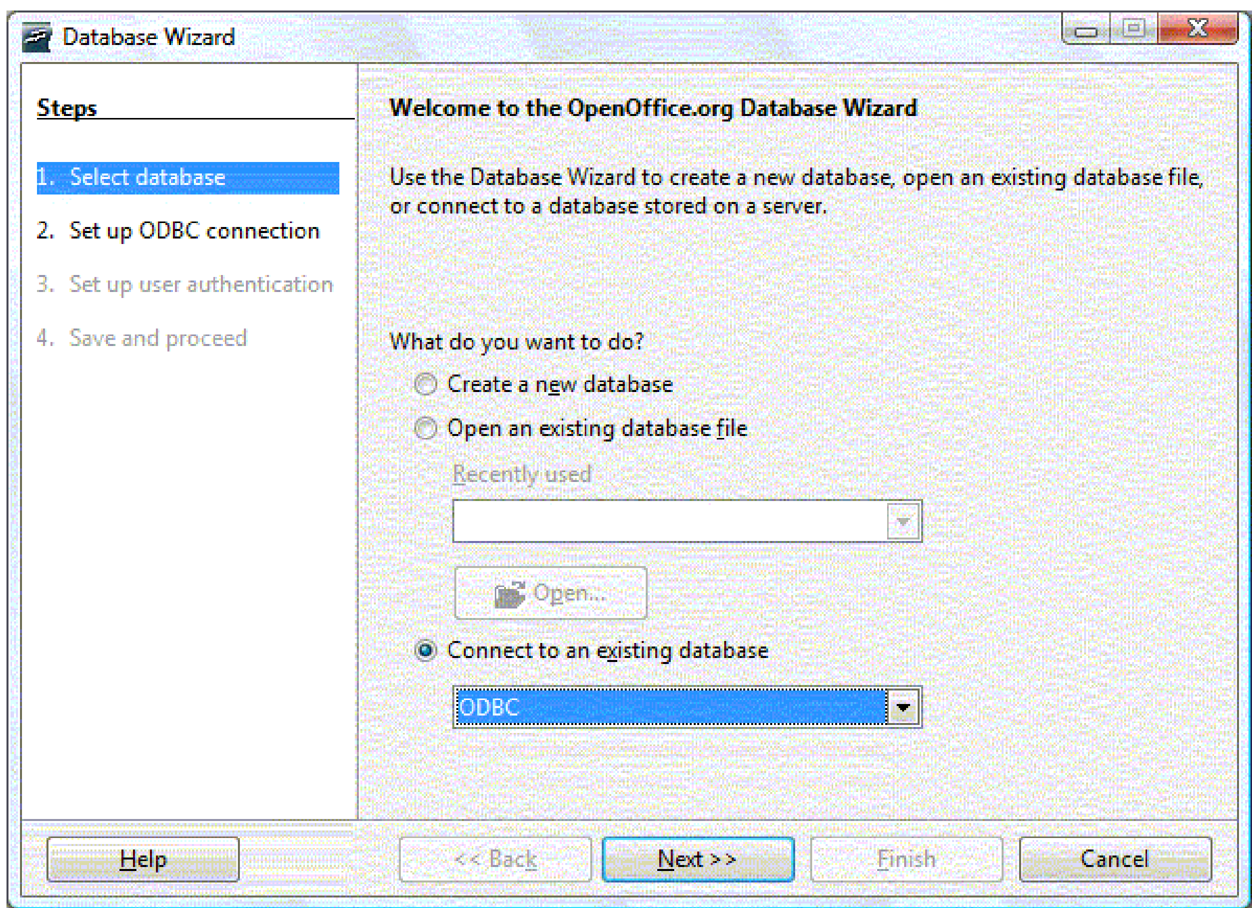
5.9. Databases and Open Office

We conclude this topic with a brief review of using Open Office to connect to and manipulate a database. This is an Open Source application which can be downloaded free of charge from the Open Office website. We do not expect you to download and install it, and you do not need to carry out any of the practical work below (though if you already have it installed or would like to try it we would encourage you to use these notes as a basis for exploring it). It is, however, worth noting the facilities in Open Office since you may in the future come across a situation in which it could be a suitable solution.

We have seen earlier in this topic how a front end to a database can be implemented using a variety of programming environments. Though this is in principle fairly straightforward it does require programming skills and is quite time consuming. It would be useful to have a ready made front end we could use to link with any database. Open Office Base provides such a facility. You would not use it for a serious application because it lacks the flexibility and display capacity of the programmed solution, but there are times when simple forms or tabular views are all that you need.

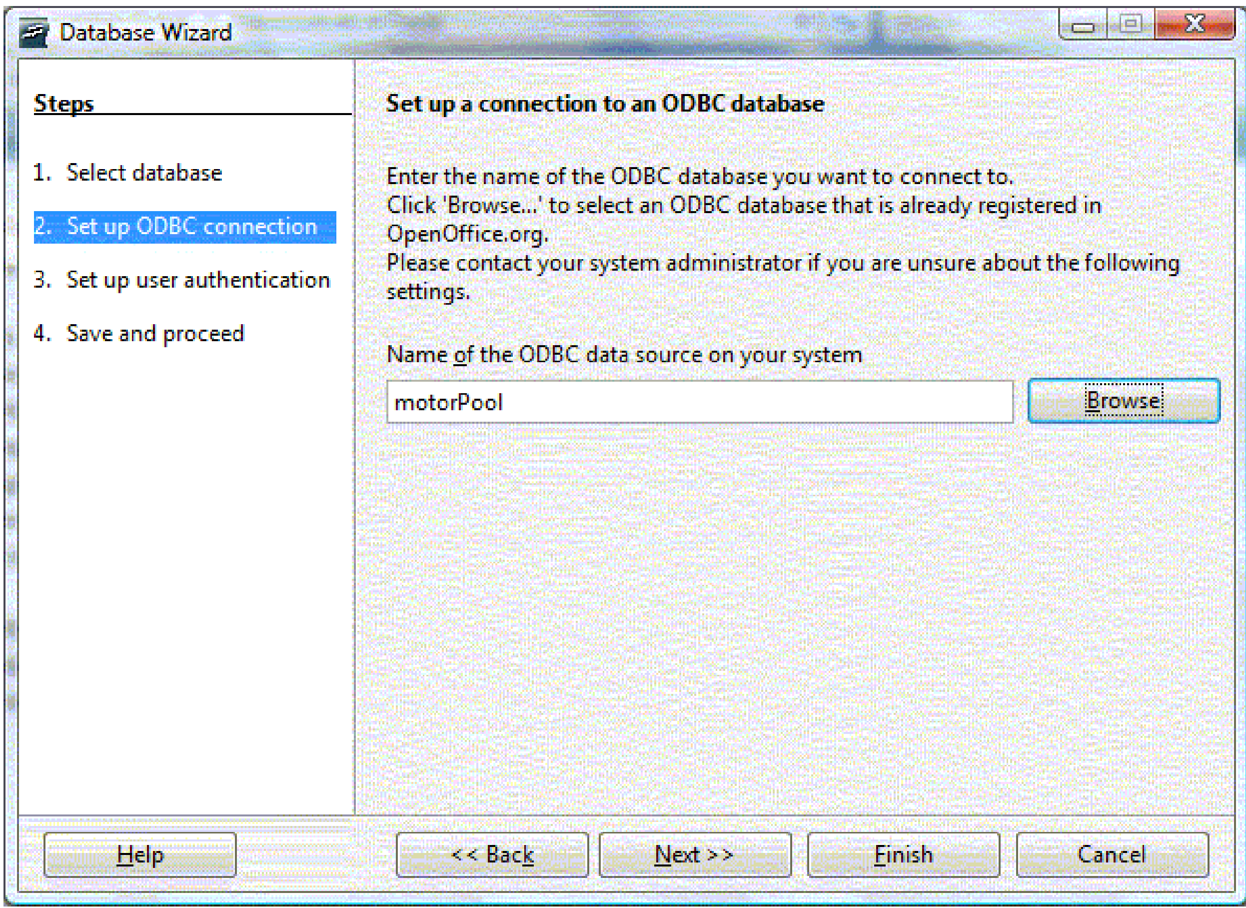
First you need an ODBC connection to your DBMS and database. We looked at this in sections 5.3 and 5.4 of this topic. You can use SQL Server, Oracle, MySql or, as in this example, Jet and a database made in Access.

When you ask Base to create a new database the dialogue box gives you the option of connecting to an existing database through ODBC, as shown in the following image:



Assuming you have an existing ODBC connection set up as a data source on the local machine you can select it in the next dialogue box.

© Copyright the University of Oxford and contributors.



You can now manipulate the database using the table, query and form interfaces of Open Office base, but all changes are made in the underlying database.

The forms and reports generation facilities are very limited but it is possible to produce quite attractive datasheets. You can also use the data in other parts of the application, for example in a spreadsheet or as the address list for a mail merge.

These notes have assumed a Microsoft operating system since that is the platform required for this course, but Open Office, since it runs in the Java Virtual Machine, will run on any platform and is commonly used on Linux platforms. Setting up the JDBC or JDBC:ODBC bridge in other operating systems is not quite so straightforward as in a Microsoft system, but it is not especially difficult and you will find plenty of advice on the Web.

Recent versions of Open Office Base also include a built-in database management system and you can use it to create a database, much as you would in Microsoft Access.

© Copyright the University of Oxford and contributors.

Unit 6. Post-relational databases

6.1. Introduction

In this topic we look at some of the ways in which database technology is evolving to meet the needs of modern software applications.

There is a mismatch between the concepts of the relational database and those of the object technology used for many contemporary applications. We review the emergence of the object-oriented and object-relational databases as potential solutions to this problem.

The growth in the amount and complexity of data held in databases provides scope for business support systems that carry out very detailed analysis of data and synthesis of information. Much of this work is incompatible with the needs of a database involved in transaction processing and we look at how data warehouses have evolved to take over the analytical processing.

We observed at the beginning of this unit that the database approach involves the centralisation of all data in one place, but one of the emergent properties of networks is to move data and intelligence away from the centre. One of the technologies implementing this migration of data is the distributed database where there is a common model but the actual data is distributed over several databases.

A more radical departure is the storing of semantic metadata (that is, information about the meaning of data) in systems that are very loosely coupled. We look at the concept of tuple space data stores and how it is evolving into a technology to drive the biggest database in the world, the World Wide Web.

By the end of this topic you should be able to describe the *broad concepts* of:

- the object-relational impedance mismatch
- object-oriented and object-relational databases
- data warehouses and data mining
- distributed databases
- web services
- tuple spaces
- the semantic web

6.2. Persistent data in object-oriented systems

If you are familiar with the object-oriented paradigm you will realise that the description of it in the introduction to this section is incomplete and simplified to the point where it is not entirely accurate. This is deliberate and intended to provide sufficient insight into the paradigm to highlight the problems it presents for persistent data storage, without covering in detail material more appropriate to another unit or course.

© Copyright the University of Oxford and contributors.

An object-oriented system is a collection of 'black box' **objects** that cooperate, using the sending and answering of messages, to provide the functionality of an application. These objects are based on **classes** that are like templates setting out the attributes and other characteristics of the objects defined. For example a Student class would define objects with a *name*, *date of birth*, *address*, etc., whilst a Course class would define objects with attributes *title*, *points*, *level* etc. For the system to operate these objects have to be created and initialised. The creation and initialisation process allocates for each object a block of data (somewhere in the working memory) formatted to represent the object's attributes in terms of the template provided by the relevant class. Each of these blocks has an **Object Identifier** (or **OID**) that is unique to the particular object. (The programmer is unaware of these OIDs: the system keeps a lookup table which links the location of objects to program variables, for example *thisStudent* or *students[23].*) The precise location of the memory blocks depends on the host operating system and may change during the running of a program, but all objects retain their OIDs and remain accessible through program variables.

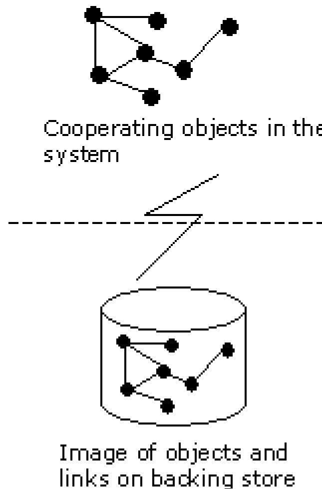
Since each object is referred to either by its memory location or by its OID, it is possible to keep several collections of the same object without duplicating the data itself. It would be possible (not necessarily appropriate, but possible) for each Course object to have an attribute that was a list of Student references. There would be no duplication of enterprise data, merely of references to it. It follows that the object-oriented paradigm can implement many-to-many relationships. You could link Student with Course, and Course with Student, without needing an Enrolment object.

Another feature of the object-oriented paradigm is that, as well as the data itself, objects have methods (functions and procedures) which operate on the data. One of the ways the 'black box' principle is maintained is that an object can require that its attributes only be accessible using the object's own methods . So, for example, if you wanted to set the *points* attribute of a Course object to 30, you would have to send a message to the object asking it to do it, for example with *thisObject.setPoints(30*), which allows the object to check the data and possibly transform it before storing it.

Finally (in this very sketchy outline) it is possible to define classes that inherit attributes and methods from other classes. Thus the Manager class might be defined as inheriting everything in the Employee class plus some additional attributes and methods.

If you envisage an object-oriented application as a collection of objects which, through their behaviours and message passing, provide the functionality of the system, you can see how it might be possible to provide persistent storage by transferring all the objects and references to them onto the backing store. The OIDs would not change and, since memory is referenced in terms of an offset from a base address, you can even store the relative memory locations of objects. When the application opens, it obtains the objects and their current states from the backing store, and re-saves them there when the application is closed:

© Copyright the University of Oxford and contributors.



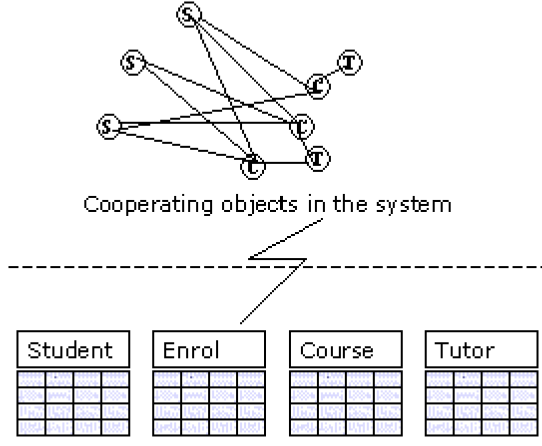
Another method of providing persistent storage of objects is serialisation, which saves information about the objects and their relationships in the form of a text file, which can be used by the application to recreate the objects and their states when it next starts up.

There are many problems with these approaches. Some of the problems arise from implementation and are outside the scope of this topic, but three of them are operational problems which can be described very simply. These are:

- new and amended data remains in main memory (where it is more at risk than data in the backing store) for a protracted period
- updated and amended data are only available to a single instance of the application
- you are limited to the amount of data that can be retained in working memory.

For this reason, most object-oriented applications involve some kind of database that is accessed in real time as the application runs.

One possibility is to store the objects as tuples in a relational database. The Student, Course and Tutor objects and their attributes could be stored as rows in relations of the same name:



The problem is clear: there is a mismatch between the objects and the way they have to be represented in a relational database. Whilst in an object-oriented application, each Student object can have an attribute that contains a list of references to course objects, the relational model requires an intersection relation (Enrol) in order to record the relationships. This mismatch, normally referred to as the **impedance mismatch**, means there is a high processing overhead associated with the storage of objects in a relational database. The mismatch can be accepted in a simple application. One possibility is to structure the object-oriented application so that there is no mismatch. For example, in the above example it could include Enrol objects. Another approach is to stay with the list of courses in the Student object but to write additional code to cope with the mismatch. For example, whenever a Student object is retrieved from the database, a query is made on the Enrol table and the result used to set the Courses attribute of the Student object.

Complex data-sets

Modern computing often involves complex data-sets with large numbers of record types and relationships. A Computer Aided Design (CAD) or Computer Aided Manufacturing (CAM) application often involves a large number of object types with complex groupings and relationships. Changes in one object need to be propagated through a large number of connected objects, often in alternative forms to represent different versions of the same product. Object technology is ideal for this sort of application, but the object-relational impedance mismatch becomes so great that a relational database is simply not an option. Similar problems arise with software that plans and controls networks. A node in the network may be simultaneously involved in a number of network links (and vice versa), all of which need to be recorded and read in real time.

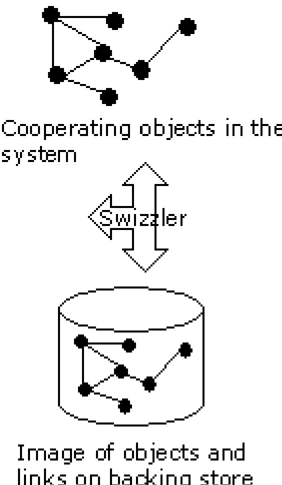
BLOBS and CLOBS

Many relational databases now support Binary Large Objects (BLOBs) and Character Large Objects (CLOBs). These are data types that are simply collections of binary or character data. They do not contain a primary (or any other type) of key, and fields of this type cannot normally be used in the WHERE or HAVING clauses of an SQL expression. (In fact, the latest version of the SQL standard includes limited searching on CLOBs.) A BLOB field may, for example, contain the bits that represent an image, but the database management system has no way of mapping the bits to anything nor of carrying out any kind of processing on them. Again, the object-oriented paradigm is the best approach to processing this kind of information. Objects can have associated with them methods that can process the data, whether to change it (to re-size or change the format of a picture) or to query it (to find pictures containing large areas of red). Here the problem with the relational database is not the impedance mismatch of the attributes and fields (which is often trivial) but the inability of the relational database to manage object methods.

Object-oriented database management systems (OODBMSs)

In essence, the object-oriented database goes back to the original design we considered at the start of this section: saving on disk what is effectively an exact image of the application running in main memory. But here, instead of doing it only at the start and end of each run of the application, the image is changed continually as data is accessed and updated. Objects (including their attributes and methods) exist in the database and are only brought into main memory as they are required. The application still refers to objects using pointers and OIDs, but the system includes a ‘swizzling’ mechanism that recognises when reference is made to an object that is not in memory, pages it into the relevant memory and swizzles the pointer to reference it.

© Copyright the University of Oxford and contributors.



It is somewhat like the virtual memory used by Windows operating systems. Data is paged in and out as required so that the application operates as if the whole of the data were currently in main memory. This gets round the problem of working with very large data-sets and of keeping the only copy of data in volatile memory, but it comes at a price. In order to provide this degree of functionality, the database is once again closely coupled to the application: the database model is a replica of the application model. It is possible to share the data-base, but it is not easy and often not desirable because the close coupling of a second application might have unexpected side-effects on the first. It is also typically more expensive to develop and implement an object-oriented database, not only because of the cost of the OODBMS package but also because of the cost of customising it to the relevant application.

Object-relational database management systems (ORDBMSs)

Relational databases have dominated the field for many years and are still the most common type of DBMS, but the emergence of object-oriented database management systems prompted the relational community to address the problems we considered above. A number of proprietary RDBMSs have been extended to include features of the OODBMS, and the latest SQL standard (SQL 3) includes a number of object-oriented features like user-defined types and accessor methods. It also supports the searching of CLOBs (but not BLOBs).

Most object-relational systems are extensions to or developments of existing relational DBMSs, but there are some products that are designed as hybrid systems. For example, the Cache © DBMS is based on three-dimensional tables and can be used as a conventional relational database, but it also supports many-to-many relationships and sub-classes.

6.3. Data mining and warehousing

These are topics at the cutting edge of database research and, although there are many areas in which they are currently implemented and used, this area is an evolving science. They are also quite complex topics, involving a range of programming applications as well as database management systems. This overview, while oversimplified, outlines the reasons for data warehousing and mining and gives some insight into the principles involved.

© Copyright the University of Oxford and contributors.

Data warehousing

A typical commercial database is heavily used. A retail establishment may need to refer to a database to find current details of stock levels and prices, may need to enter sales and returns etc., and all that needs to be done instantly. It is vital that there are no serious bottlenecks in the system, which would prevent either a customer from being given information or a sale from taking place. This sort of database user wants a transaction processor. The transaction might be what we normally call a transaction, like the recording of a sale, but it could also include the 'transaction' of obtaining information about a specific product needed for customer interaction. The system needs to be able to protect the database from the potential errors that would arise if two people accessed the same data at the same time. A database which meets these needs is often called an **Online Transaction Processing (OLTP)** system. This kind of database needs locking and transaction systems that can preserve the integrity of data despite the actions of multiple users and interrupted transactions.

Database management systems are also used in supporting decision-making. A decision-maker may want information to assist in deciding which product to phase out, or in which new area to market a product. A regulatory organisation may want to use a database to help in targeting resources to areas where non-compliance is most likely. These people are more interested in trends than in transactions and want the database to carry out some analytical processing. They are looking for an **Online Analytical Processing (OLAP)** system. It may often be possible to undertake that analysis using large and complex SQL queries, but sometimes more is needed. Even when it is possible to use ad hoc SQL queries, the nature of the information required is likely to involve joining a large number of tables and aggregating data into a very complex view. There is a real danger that the use of a transaction processing system as a decision support system will slow down the system and at times may even make it unavailable to the transaction users.

One answer is to have two copies of the database. The second database is not concerned directly with transactions, and is updated once a day by a special program which copies to it transactions entered into the main database during the past 24 hours. The program can be designed to do the update in a way that does not interrupt the transaction processor, and as it can be hard-wired into the system it is anyway an efficient operation. Also, since it is only written to in the daily update, it can have more optimistic locking rules so that one complex enquiry does not necessarily lock out other enquiries. This second database holds the same data but it does not necessarily hold it in the same form. As there is no need to record transactions nor to retrieve large numbers of simple queries, it is possible to optimise the table structure for analytical processing.

Once you have the idea of a separate database devoted to analytical processing it is an obvious step to add to it data from other databases which contain potentially useful information. Hence a **data warehouse** is established.

Another obvious step is to add specialised processing capacity to the database engine so that it can be hard-coded to carry out complex operations, for example to compare a value in the current row with values in previously output rows.

The main database schema used for a data warehouse is the Star Schema. There is a table (a Fact table), which is largely a collection of foreign keys linking all the other tables (often called Dimension tables). A simple example might be a Fact table that contains required facts (for example the date and amount of a sale), with key references to the tables that contain the dimensions for the analysis (for example the products, shops, suppliers, etc. tables). In some cases, for example in the data warehouses used by regulatory authorities, the Fact table may consist only of links to Dimension tables (for example your tax reference, National Insurance [Social Security] number, your student loan number, your .... who knows what else?).

© Copyright the University of Oxford and contributors.

Early data warehousing systems were dedicated applications, but recent versions of Oracle include much of the hard-coding needed for data mining and warehouse maintenance. Future systems may well be based on XML, any new data imported into the warehouse being converted into an extended mark-up language. This would make data warehouses inter-operable and would raise even more worries about the privacy and security of personal data.

Pause for thought

Can you recall the eight principles of the UK Data Protection legislation?

Answer

- fairly and lawfully processed
- processed for limited purposes
- adequate, relevant and not excessive
- accurate
- not kept for longer than is necessary
- processed in line with the user's rights
- secure
- not transferred to other countries without adequate protection

Data mining

One of Asimov's short stories (Asimov 1956) is about a future time when computers are so powerful, reliable and self-programmable that the only difficult part of computing is knowing the questions to ask. In that future only 'Grand Masters' can work with the computer and their job is to get the computer to come up with new information. We are some way off that future, but there is a hint of it in the amount and complexity of the data held by organisations. We interrogate the database to find out how many people bought a product, how many live in Oxford, or how many paid by credit card, and increasingly we are finding ways of combining these queries to produce collated data. But what we really want to know is something much more like information than data. The questions we would most like answered are 'Which of our customers are most likely to respond to a particular kind of mailshot?' or 'Which of our customers are the best credit risks?'

Data mining is the umbrella word for a range of systems that undertake this kind of research. In the terabytes[1] (some say even petabytes[2]) of data held in the warehouse there are nuggets of information waiting to be mined. We just need the right tools. This is a wide and complex area on which we can only touch in this unit, but some flavour of it can be gained from the following broad summaries of three approaches to data mining.

© Copyright the University of Oxford and contributors.

Rule-based systems

This is a simple concept, though often far from simple in its application. Business rules decided by the organisation, or known characteristics of relationships between data that have been observed by the organisation, are programmed into the system, usually using a declarative or list processing paradigm. Many such systems are in fact based on the Prolog or Lisp programming languages. It is a sort of top-down approach. We know what governs the outcomes we want and we use the rule-based system to mine it for us.

Fuzzy logic systems

Fuzzy systems are similar to rule-based systems but they allow some degree of fuzziness in the way decisions are reached. A system of logic is used (Bayesian logic) in which the expression if(bigSpender AND highEarner) can return not only true or false, but also 'maybe'. The propagation of these fuzzy outcomes through the system can produce more accurate end-results. The economist John Maynard Keynes is reputed to have said 'It is better to be roughly right than to be precisely wrong' and this is the principle of Bayesian logic systems. In fact the systems are far more complex than suggested here. The 'maybe' involves many levels of statistical probability and they are combined in very sophisticated ways. In operation, however, Bayesian logic systems are characterised by their ability to combine fuzzy inputs to produce a highly probable output.

Bayesian logic is currently being used to develop anti-spam programs. This goes well beyond the subject matter of this course, but if you are interested an intelligent search on the web will bring up lots of information of the subject.

Artificial neural networks (ANNs)

Neural networks take the processing of fuzziness to new heights. They are modelled on an approximation (not altogether shared by neurologists) of the way the human brain works.

Inputs are connected to a layer of cells, which pass messages to one or more inner layers of cells before connecting to an output cell or cells. They have many uses, including image and handwriting recognition, but they are often used to discover information in large and diverse data-sets. Here the approach is bottom-up. We do not know how the data govern the outcomes, so we do not have the knowledge to programme Boolean or Bayesian logic rules. We have to 'train' the network to discover these rules.

One way of training a neural net is to set the input and output cells repeatedly to values of a known situation. On each training event, the weights of the cells are adjusted to provide the best overall match between the input values and the output value. For example, you might input details of the age, earnings, job type, etc. of a sample of people you have investigated together with their historical creditworthiness. The system adjusts the weights in the cells to achieve, overall, the best match between inputs and outputs. Thereafter you can input similar details of all customers and observe from the outputs which ones are the good (or bad) credit risks.

If you use input values based on industry type, income level, savings pattern, gross profit rate, etc., and if you use output values based on a random audit, you could use a neural network to select which individuals or companies would be the best targets for a Revenue or other regulatory body audit. It may also be possible to determine people who are poor health risks without knowing the precise influence any individual characteristic contributes to the overall decision.

This is the unique (and to many chilling) feature of the neural network. Rule-based and fuzzy systems may be complex but in the final analysis we can deduce and explain how they arrive at their conclusions. We believe the neural net because experience tells us it works, but we cannot explain how it reaches its conclusions.

© Copyright the University of Oxford and contributors.

Discussion

Is this technology a blessing or a curse for mankind?

How can all this take place when we have data protection laws?

This is an important topic on which there are no easy answers and we would value your views. Please post them to your tutor group forum.

References

Asimov, Isaac, 1956 'Jokester' (short story), in Asimov, Isaac, 1987 *Robot Dreams*, ACE Books, New York, ISBN: 0441731538.

[1] terabyte: 1,024 gigabytes (approximately one trillion bytes)

[2] petabyte: 1,024 terabytes (a lot of bytes)

6.4. Networks and the migration of intelligence

This section is a brief overview of the impact of networking on business structure and organisation. It is not particularly about databases but is important because it describes the effects of, and the motivation for, the technology discussed in the rest of this topic.

Metcalfe's Law and network effects

Robert Metcalfe is the founder of 3Com and the designer of the Ethernet networking system. He recognised a universal feature of networks, one that had been apparent in areas like the railways and the telephone system, that a network required a critical mass to take off, and after that point it increased in value at a much higher rate. Indeed Metcalfe's Law, as it has come to be known, states that beyond the critical mass the value or utility of a network equals the square of the number of users. As with many of these 'laws', the apparent mathematical precision is misleading, but it is certainly true that beyond a certain point the added value of additional network nodes is non-linear.

A network promotes modularity and the distribution of intelligence within a system. Instead of being concentrated (typically at the centre of an organisation), intelligence can remain where it is developed or gravitate to where it is most easily maintained. This migration occurs within organisations but it is increasingly occurring between organisations.

The nature of the firm

When the economist Ronald Coase was awarded the Nobel Prize for Economics in 1991, one of the citations justifying the prize was his paper published in 1937 entitled 'The Nature of the Firm'. Coase, then a socialist, had been studying in the USA to determine why private enterprise had evolved monolithic corporations that carried out the full range of market functions from market research through manufacturing to support. He concluded that the nature of firms was not accidental or political but followed an inexorable economic law. His conclusion

© Copyright the University of Oxford and contributors.

was that economic activity is carried out in the way that minimises the cost of transactions. If these costs are minimised by a firm doing something in-house, then that is where it will be done. A good example is procurement. If every office clerk bought his own paper and pens from an external stockholder, there would be a duplication of the research, ordering and accounting costs, so it is better to bring the stockholder in house. In a similar (though more complicated) way, the minimisation of transaction costs produced the vertical enterprises that characterised the greater part of the twentieth century. Coase identified the following main costs:

- Search costs – buyers and sellers need to find each other in what may be a disorganised and dispersed market.
- Information costs – researching the appropriateness, quality and price of the goods and services available.
- Contract costs – negotiating terms, exchanging letters, holding meetings, etc.
- Decision costs – evaluating offers, taking legal advice, evaluating longer-term implications of the transaction.
- Policing costs – ensuring that the goods and services are of the required quality, delivered on time, etc.
- Enforcement costs – coping with failures in the contracting, payment or delivery process.

In the social and technological environment of the time, these costs were minimised by bringing everything in house. It was the only way to ensure continuity of business without excessive costs.

In a **netcentric** business environment, some of these transaction costs disappear and many more are very much reduced. It is no longer necessary to bring everything in house; indeed it becomes uneconomic to do so because the in-house transaction costs (management, maintaining expertise, etc.) are greater than the costs of external transactions.

Linking the implications of this law with the law of network effects, it would appear that the nature of the firm is likely to change. There are indeed already signs of this change: companies are increasingly outsourcing non-core operations. It is already possible to identify well-managed value chains among cooperating independent companies that are more efficient than the internal chains of vertically integrated firms.

Unbundling the corporation

The theme outlined in the last paragraph is developed by Hagel and Singer (2000), whose analysis is centred on the three main types of business that a typical organisation runs. It will have a customer relationship business, a production (of goods or services) business and an infrastructure business. These three businesses have been traditionally welded together as core functions and are rarely outsourced. There is tension, however, between the strategic and cultural positions of the three functions. Hagel and Singer summarise these differences as:

	Customer	**Product Innovation**	**Infrastructure**
Economics	Making the most of customer relationships implies diversifying the offering.	Time to market is crucial in achieving premium price and market share. Flexibility is important.	High fixed costs imply economies of scale to produce low unit costs. Flexibility undermines this.

© Copyright the University of Oxford and contributors.

Competition	Centred on customer loyalties and inertia.	Centred on talent. Low barriers to entry enable small players to thrive.	Centred on scale. Consolidation reduces unit costs.
Culture	Service oriented. Investing in customer relationships.	Employee oriented. Attracting and nurturing talent.	Cost oriented. Standardisation and efficiency.

(Source: adapted from Hagel and Singer, 2000)

Outsourcing is common in some support areas, for example IT and Human Resources, but increasingly this fragmentation goes right to the heart of the business. Most people think of Yahoo as a search engine, but in fact it outsources the search engine business and it is a pure customer relationship business. Already the majority of credit card transactions in the United States are processed by a third party (for example First Data) and not by the company facing the customer. Again, it is worth remembering the opposite but complementary trend, of independent firms working so closely together that they become a virtual firm. (For example, if you buy a Cisco router the chances are that none of the firms that make, supply, deliver or support the product will be owned by Cisco.)

In the remaining sections of this topic we look at some of the technologies that give rise to the distribution of intelligence which is driving these changes in the structure and operations of organisations.

References

Coase, R. H., 1937. *The Nature of the Firm*.

Reprinted in, Coase, R. H., 1990. *The Firm, the Market, and the Law* (Paperback). USA:University of Chicago Press. ISBN:0226111016

Hagel, John and Singer, Marc., 2000 'Unbundling the Corporation, *McKinsey Quarterly*, no. 3, pp 148–61, also available online, http://www.mckinsey.com/business-functions/strategy-and-corporate-finance/our-insights/unbundling-the-corporation, accessed 8 January 2017.

6.5. Distributed database management systems (DDBMSs)

One of the fundamental characteristics of the database approach is the centralisation of data, but this is incompatible with, and seriously threatened by, the power of networks to migrate intelligence. New paradigms for handling distributed data are emerging and we look briefly at some of them in the rest of this topic. In this section, however, we look at the way a single relational database can be distributed between locations in an organisation. This may be the result of splitting a single database, or of linking existing relational databases into a single networked database.

Note that distribution of data over networks is not the same thing as accessing data over networks. We saw a very simple method in Topic 5, where anyone could have access to our Gigs database by simply using a browser. In the same topic we saw how a connection can be made to a database using JDBC, and how the example we

© Copyright the University of Oxford and contributors.

used could be changed (using very little extra code) to access a remote database. In this section, however, we are not concerned with *accessing* a database over a network, but with the *distribution* of the database itself over the network. There are three approaches to the distribution of a relational database:

- the fragmented, or partitioned, database
- the replicated database
- the hybrid system

Fragmented, or partitioned database

We start with a very simple example. A large company has branches in several cities and each branch has very high data access demands (in particular for recording business transactions, which typically require several accesses per transaction). Despite the speed of modern networks and the power of modern RDBMSs, performance is limited because of the large amount of communication and message management needed for all cities to communicate with the central database. The answer is to split the database tables horizontally. Each branch has the same set of tables, with the same attributes and keys, but the rows of the tables in each location are restricted. For example, the Paris branch has tables that contain only the rows where the value of the city attribute is Paris. Each of the branch databases has its own schema, recording what is held locally, but it also has the main schema (containing metadata about the whole database) and a distribution schema (showing where data is located). When data access is requested, the local DBMS uses the schemas to locate the data and to access it in the appropriate way. Most of the time the data will be read from or written to the local database, but occasionally it may be necessary to access the database at another location. The organisation of the database is transparent to the user who uses the same commands and expressions, and gets the same response, wherever the data is located. This kind of distributed database is called a **horizontally fragmented database**. It is the most common type. But sometimes there is a need to fragment using column headings: for example, the payroll department may need employeeNumber, salary and tax code, whilst the HR department needs employeeNumber, name, address and dateOfBirth. Provided the only duplicated fields are those comprising the primary key, the database can be fragmented by columns producing a vertically fragmented database.

Replicated database

This approach involves maintaining a complete copy of the database at each site. All database access is on the local copy. Where the tables are used for long periods as read-only, this can be a very satisfactory solution, with the replicated databases being updated at intervals using the same *snapshot* made of the data on the central copy.

If write access is light, it is still possible to use a replicated database but any change in a local copy of the database must be propagated throughout the system immediately. There is a performance overhead since each write operation has to be carried out many times. Nevertheless, where read access is heavy and write access light, this approach may be a more efficient solution than using a central database.

Hybrid system

This is simply a combination of methods to achieve the optimum result and is in practice the most common system. Some of the data is stored centrally, some is fragmented and stored locally, and some is replicated throughout the network. It is a system that seeks to maximise the advantages and minimise the disadvantages associated with the networking of data.

© Copyright the University of Oxford and contributors.

Homogeneous and heterogeneous distributed databases

Where an existing database is fragmented or replicated, it usually results in a **homogeneous** distributed database. All parts of it are based on the same data model and run on the same type of DBMS. A system that is an aggregation of existing databases is more likely to involve different data models and DBMSs, producing a **heterogeneous** distributed database.

All distributed databases require some sort of central management, but this is clearly a major issue with a heterogeneous DDBMS. One approach to managing this kind of system is to employ **gateways** that translate the commands sent to one database into the format required by the other. This is, however, an incomplete solution because it does not cope with major differences in the underlying model and does not provide concurrency control. A more ambitious, and more effective, approach is the **multibase** (MDBMS) system. This involves a separate DBMS that sits on top of the existing systems. Queries are made to the MDBMS, which carries out the necessary queries in the underlying databases, and then transforms and aggregates the results. The MDBMS also provides concurrency control for transactions that involve more than one of the underlying databases.

Concurrency and transaction processing

We saw, in Topic 4 how databases use locks in order to preserve the serialisation of processes, and ensure the atomicity of transactions by using rollback and commit. Where the database is replicated, the local DBMS is unable, on its own, to provide concurrency control or deal with SQL transactions. Even where there is no replication, if the database is fragmented it is still necessary to introduce another layer of processing in order to cope with SQL transactions.

Any system that seeks to commit everything at once includes some risk of leaving the database in an inconsistent state, but in a single system the risk is very small. In a distributed database, where the transaction to be committed involves action on several systems, the risks are much greater. The essence of the system normally used to provide concurrency and transaction control in a DDBMS is a **two-phase commit** process. Where an SQL statement, or a transaction of related SQL statements, involves a distributed database, none of the commands is committed until there is agreement between the affected DBMSs. This process is controlled by a multibase system (MDBMS), which monitors the distribution of the query or transaction. When the originating location completes the series of statements that form the transaction, it sends a **commit request** to the MDBMS, which polls all affected systems to check whether they are **ready to commit**. The local DBMSs reply with a **ready to commit** or **abort** message, or they may fail to reply within a set period. Depending on their replies, the MDBMS sends a **global commit** or **global abort** to all participants. On receipt of a global commit, each resource manager writes the data from its log to the database tables.

Any system for preserving the serialisation of transactions presents recursive opportunities for failure. However fine the granularity of the transaction, the final committal of data can still go wrong. In practice, if the concurrency arrangements do ultimately fail, it is at the final write to table stage, and the affected DBMS will usually be aware of the failure and able to rectify the position using its transaction log.

Distributed database transparencies

This sub-section contains nothing new, but summarises the levels of transparency exhibited by DDBMSs that are described or implicit in the processes outlined above.

© Copyright the University of Oxford and contributors.

Fragmentation transparency

The user does not need to know anything about the fragmentation, and may be unaware that the database is fragmented. The user is aware only of the global schema and uses the distributed database as if it were a single local database. So a list of all employees would require

```
SELECT * from employees
```

even though the data will in fact be retrieved from a number of tables.

Replication transparency

The user does not need to take account of (or even know about) the replication of the data being used.

Location transparency

This is implied by *fragmentation* and *replication* transparency, but may also be a characteristic of systems that do not exhibit them. In such systems the user needs to know *how* the data is fragmented but does not need to know its *location*. Here the list of employees would also be obtained by a single SQL command but this time it would need to include the names of the fragmented tables:

```
SELECT * from Region_1_Employees
UNION
SELECT * from Region_2_Employees
```

Naming transparency

In a distributed system, problems would arise if the same name were used for different objects, or for objects located in different places. This is avoided either by using a central **name server** which controls the allocation of new names to objects, or by using a unique prefix in each of the databases (for example the Region n prefix in the above example). The extent to which naming transparency is exhibited varies and depends on the way the data is distributed and accessed.

Transaction transparency

Transaction transparency ensures that all transactions, wherever they originate and whatever parts of the system they involve, maintain the integrity and consistency of the database.

6.6. Web services

The technological foundation for contemporary netcentric computing is the new technology called web services, which are based on the use of the Extensible Mark-Up Language (XML). Many of the ingredients of web services technology are not new, but they are combined and empowered in a new way to produce new functionality.

Web services technology enables computers to talk to each other, to exchange semantic messages, and to do so even though they are incompatible technologies in terms of their hardware and software.

Web services provide an application communication service. The applications might be under the immediate control of human input (for example, running in a window on your desktop) or might be running without immediate human supervision. For example, the communication could be between accounting or enterprise

© Copyright the University of Oxford and contributors.

resource planning (ERP) systems. Web services include the equivalent of a 'Yellow Pages', in which an application can discover what services are provided by a remote system, and the protocols it needs to use to access them. Under this system it does not matter where data is located, or what kind of data model or DBMS is used. The messages in both directions are in common XML-based languages.

A system 'wanting' a service (it is hard not to be anthropomorphic when discussing this topic) would use web services to find out where the service it requires is available. Using the same system, it would interrogate a number of the sources it found to obtain the information it needed to make a decision. It might then use the same procedure to enter into a contract with one of those remote systems.

Web services also enable users to make use of 'contentless' software: for example web services can not only allow a system to interrogate an existing, remote stock database, but it can also ask a software provider for the minimum functionality needed to open and use a new stock database. The local system gets the ability to add, remove, update, etc. using the web service, without needing to know what sort of machine or what sort of software is being used by the provider.

It is not yet a pervasive technology, but it is gaining ground and it is the future. The key features that will drive its widespread adoption are:

- It honours diversity in existing systems. There is no need to rip out old systems and replace them with something entirely new. The existing investment is not lost since web services can be implemented as overlays on existing systems.
- It is itself modular. The software needed for implementation is mostly available as ready-made modules which can be readily adapted, leading to faster and relatively inexpensive rollout of new features.
- It is loosely coupled. Unlike existing integration technologies, it is not tied into existing systems and so can be implemented on any platform without worrying about the platforms with which the system needs to communicate.
- It can be incremented gradually. In contrast with legacy technology, there is no need for an all-or-nothing decision. It is possible to test the water by having web services enable part of a system.

We cannot in this unit spend much time on the technology of web services, but we hope the following examples will provide some insight into the way they promote flexible networking and therefore promote modularity in applications and the migration of intelligence.

Web services are based on the Extensible Mark-Up Language (XML), which is used to define other languages and has many characteristics and uses. The one we focus on here is its capacity to carry meaning. Hypertext mark up language (HTML) provides only a means of displaying information. The displayed information may mean something to the human reader but it means nothing to the machine. XML is able to record and transmit information from which *meaning* can be derived by both human *and* machine.

A simple XML file contains information between <tags> that define the information. In its simplest form an XML file looks like this:

© Copyright the University of Oxford and contributors.

```
<?xml version="1.0" encoding="ISO8859-1"?>
<CarHire>
  <car>
    <make>Ford</make>
    <model>Mondeo</model>
    <cost>100</cost>
  </car>
   <car>
    <make>Ford</make>
    <model>Fiesta</model>
    <cost>75</cost>
  </car>
</CarHire>
```

The tags <CarHire> and </CarHire> define some data about car hire. The car tags define each car, etc. It is far from being in plain text but it is capable of being understood by a human. It can also be 'understood' by a machine.

The simplest way of using XML is to display it in a web page. If the above XML file had as its second line:

```
<?xml-stylesheet type="text/xsl" href="cars.xsl"?>
```

and it were retrieved by a browser, the browser would look at the stylesheet referred to (cars.xsl) and would display the data as:

Make	Model	Cost
Ford	Mondeo	100
Ford	Fiesta	75

It is a relatively simple matter to arrange for the stylesheet to differ according to the platform on which it is received. The relevant stylesheet would be loaded so that the same XML file would be displayed differently on a PC, a PDA, a mobile phone or even as a spoken telephone message.

Alternatively, the XML file might be retrieved by an application that had been programmed to extract data from the file. One way it might do this is to locate on the Internet a Document Type Definition (DTD) or Schema file, which again might be based on a reference in the XML file. For example, the XML file might contain (among other related lines):

```
xsi:schemaLocation="http://www.zzcars.org/CarHire.xsd">
```

From this **schema**, the application would 'know' what data fields to expect and which of them were mandatory.

Web services technology builds on these simple concepts to produce even more flexible and feature-rich alternatives to traditional models of distributed applications. The functions and protocols of web services technology are quite complex but we can gain some insight into them from considering a simple example.

© Copyright the University of Oxford and contributors.

Imagine a university in which lecturers frequently order books. The established process involves passing requests through the library which deals with them according to business rules (here very simplified):

- If Amazon charges more than $30, research alternative sources.
- If not available within 24 hours from Amazon, research alternative sources.
- Otherwise, order immediately from Amazon.

The manual system might involve a library assistant searching the Amazon site to find the details then approving requests or referring them for further investigation. Since Amazon publishes details of its web services, it is possible to program an application to obtain an XML file containing details of selected books. Here is a very simplified *extract* from a typical Amazon XML file (which in practice would contain a large amount of detail).

```
<Product>
     <Asin>0764516426</Asin>
     <ProductName>eBay for Dummies</ProductName>
      <Authors>
           <Author>Marsh Collier</Author>
      </Authors>
       <Price> 15>39 </Price>
       <Availablity> Usually ships within 24 hours</Availability>
</Products>
```

There are ready-made software objects that can parse XML files and extract data from them, and the programmer could very easily adapt an application to extract from this XML file a table of data like this:

University of Oldshire Automatic Ordering system

ASIN (ISBN)	Price	Availability
0764516426	$15.39	Usually ships within 24 hours
0471236403	$35.00	Usually ships within 24 hours
0471266523	$27.97	Usually ships within 24 hours
0471432571	$24.50	Usually ships within 24 hours

Note this is no longer a page on the Amazon system but data within the local software application. It is displayed here so that we can discuss it, but it would in practice be an object, for example an array, in the local application. Based on this data the application can identify orders needing manual review and make automatic orders for the remainder.

There are, of course, issues of trust, contract and payment involved in more ambitious web services, and the technology to deal with these things is not yet standard. But it is currently possible, and will eventually be commonplace, to find and use commercial web services. It does not matter where the data are: the machine will find them, negotiate their use and obtain them.

© Copyright the University of Oxford and contributors.

Our main concern here is with accessing data, but web services do far more than simply access data. A web service can provide a service. Though the following is a trivial example, it gives the flavour of the power of web services. If I were writing an application to produce a business plan, I could simply program the input and output and use third party web services to do the relevant calculations. I would have no idea what sort of platform or programming language was being used at the other end but would simply use Hypertext Transfer Protocol (HTTP) over the World Wide Web to send the raw figures in XML and in the same way get back the ratios in XML.

6.7. Tuple spaces

So far we have looked at the distribution of data using a variation of the tightly defined and complex system of an RDBMS, and the more flexible but still quite complex web services systems. There have been a number of research projects over the years into even more loosely defined and simply operated databases using something similar to the declarative programming of languages like PROLOG. The generic term for these data management systems is tuple spaces. The most successful early implementation of tuple spaces was a system called Linda developed in the 1980s. It was, indeed still is, highly regarded in academic circles but it was little used in commercial applications.

In this context a tuple is a list of fields and values, for example (course='Java', points=30, startDate=30/1/06.) In many implementations the tuples contain only the values, for example ('Java', 30/1/06), with the users and the system following a common protocol for the field names.

A **tuple space** is a public repository that contains tuples. It works on the basis of association. Tuples in the tuple space can be accessed by matching some or all of the elements in a template tuple with tuples in the space. If we query the tuple space with the template:

```
('Java', 30, 1/1/06)
```

we will retrieve either nothing (if there is no match) or a tuple, which is identical to the template. If, however, we query the tuple space using the template:

```
(?, 30, 1/1/06)
```

we may get several tuples, for example:

```
('Java', 30, 1/1/06)
```

```
('Databases', 30, 1/1/06)
```

```
('Bookkeeping', 30, 1/1/06)
```

because the tuple space is associative. The target tuples are identified by matching them with the attributes provided in the template.

There may be more than one match, even if the template contains values for all of the fields. Unlike the relational database, which is based on the Set (which cannot contain duplicates), tuple spaces are based on the Bag (which can). This is an important feature of tuple spaces.

© Copyright the University of Oxford and contributors.

A key feature of these systems is that they are *purely* associative. You can add a tuple, read a tuple and remove a tuple, but you cannot operate on a tuple whilst it is in the space. If you want to amend a tuple you have to remove it, and write to the space a new tuple containing the amended information.

Think of a simple online transaction system in terms of tuple spaces. A seller would write to the space a tuple containing relevant attributes, like product category, make, model number, etc., together with an asking price and a seller reference. A buyer would ask to read tuples that matched a template which showed the make and model number, and might then select one of the tuples and ask for it to be removed from the space. The removed tuple would be amended to show a buyer reference and then rewritten to the space. The seller would check at frequent intervals for tuples holding his seller reference and a not-null value for the buyer reference. Of course this is not the complete application. Many questions arise about security and the way the matches are converted into transactions, and tuple spaces are not in practice as simple as suggested here. But the core concept is simple and potentially very powerful.

JavaSpaces

Using the object-oriented paradigm, it is possible to envisage tuple spaces, which allow users to store and exchange executable content. Software objects can be stored in tuples and retrieved in (essentially) the same way as data. Whilst the software objects are in the tuple space, they are passive but once they are retrieved the user can access their public attributes and methods. There is an implementation of this kind of tuple space system, called JavaSpaces, which allows users to access Java objects which can be used in a local application. This system does much more than provide an associative data store: it provides a framework in which distributed applications can be created and run. You can have thin clients which themselves contain little functionality but which are able to perform a variety of tasks by obtaining objects as they are needed from a tuple space. This is a mature technology and it is used in commercial applications, but it remains too tightly coupled to meet the ideal of tuple space technology. The spaces application and the user applications must be only running in the Java Virtual Machine and must be built using special JavaSpace library files.

XML spaces

XML spaces bring together:

- the basic Internet protocols, for example HTTP and DNS
- the tuple space paradigm
- web services technologies
- the Public Key Infrastructure (PKI *)

(* We assume you will have studied PKI in another course or unit. If, however, it is new to you, you should first read this background paper on security of transactions [Link not available in this medium, please view it in the online course.]).

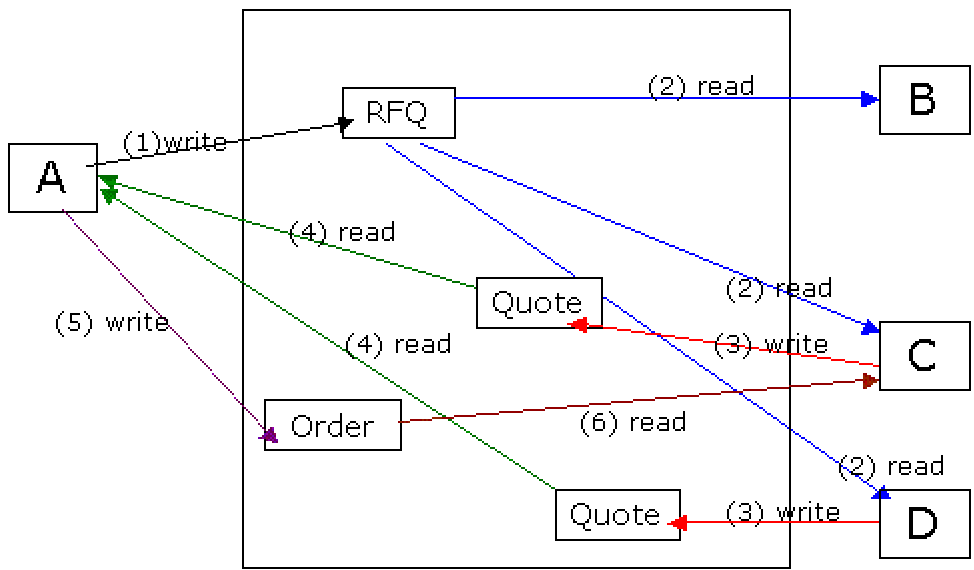
XML spaces are tuple spaces that contain XML documents and which are accessed using web services technology. The spaces may be located:

- on an intranet, for use within an organisation
- in the demilitarised zone of a server for use by partners of an organisation
- on the Internet as a service for use by a community of users (or even any user)

© Copyright the University of Oxford and contributors.

The use of XML spaces makes it possible to share XML documents between diverse platforms and applications in the same way that the World Wide Web enables browsers to display pages on a wide range of platforms. An application, whether running on a mainframe or an intelligent microwave, can access the XML documents in the space.

The key feature of XML spaces, however, is the way they foster flexible interaction between users. Here is a simplified view of the way the technology could be used for a tendering process. The companies B, C and D might be partner firms or firms previously unconnected with company A.



The request for quote (RFQ) document would be accessible to many (possibly all) users of the system. It would be read by interested parties, some of whom would write to the space their quote for the contract. They would ensure the quote was confidential by using A’s public key, and they would authenticate the quotation by using their own digital signature. A would award the contract by placing an order to the space, protected by using C’s public key and authenticated using A’s digital signature.

This may seem a long way from databases, but as this technology develops, the amount of material in these tuple spaces will increase and they will to some extent perform the role currently served by the conventional database. Indeed, the ideas we have touched on in this section are at the root of the technology which is forecast to produce the biggest and best database ever – the Semantic Web – which we will look at in the final section of this topic.

6.8. The semantic web

In his seminal paper proposing the document management system which was to become the World Wide Web, Tim Berners-Lee wrote ‘The aim would be to allow a place to be found for any information or reference which one felt was important, and a way of finding it afterwards’ (Berners-Lee 1989).

Many years later, he wrote ‘For the documents in our lives, everything is simple and smooth.’ ….. ‘But for data, we are still pre-Web’ (Berners-Lee 2001).

© Copyright the University of Oxford and contributors.

Most of us use some kind of intelligent agent to handle the data in our lives. It is not real intelligence of course, but our calendars can 'understand' that it is time for an appointment; our e-mail client can respond to a request for a receipt, and our operating system 'knows' which application to use to run a selected file.

The semantic web is an idea – a dream some would say – of a web of intelligent agents able to 'learn' from each other and to grow in their 'understanding' of the meaning of data. If I have an urgent call to attend a meeting, my agent should be able to use local data and data retrieved from a range of other sources, to arrange for transport and accommodation, renegotiate any existing appointments for that time, and inform me that, as I will need to stay in Little Puddlehampton overnight I might like to go to the Pig and Whistle where there is my kind of music. The data that will underpin these outcomes will be new, higher-level data synthesised from existing data.

In organising my trip to Little Puddlehampton my agent did some remarkable things. It used data in its own domain to 'know' that I like modern jazz. It found a source (which other sources said was reliable) with data about live music in the area. It even 'discovered' that the otherwise meaningless ABC Group which was all the Pig and Whistle agent provided, was in fact a new modern jazz trio that had recently been featured on Radio 3. There will have been many exchanges between agents, including messages like '"How current is the information?" "How reliable is the information?" and even "What do you mean by that?"'.

The technology of the semantic web

The technology of the semantic web is still evolving, and there are some disputes about the best way to go. But the approach we consider here is the one which seems most likely to become the de facto standard. Its hallmark is simplicity. It depends on three concepts and technologies, two of which we already use:

- Universal Resource Identifiers (URI)
- Hypertext Transfer Protocol (HTTP)
- Resource Description Framework (RDF)

Universal Resource Identifier (URI)

A URI is a text string that uniquely identifies something. The most common URI at the moment is the Universal Resource Locator (URL) which, as well as identifying the object, also locates it. There are other URIs which identify but do not locate: for example, a document may have a URI that is a Universally Unique Identifier (UUID), which is generally a number generated from the time, date and Ethercard number. An agent might use a UUID to reference local data whilst using URLs to access remote data. There is also a type of URI that is constructed using hashes and cryptography so that the URI itself contains a digital signature. A URI can point to a particular part of a document, for example to a particular element in an XML document.

URIs are the raw material of the semantic web. They contain the primary data and may also contain metadata, for example data about the semantics of the primary data.

Reputation, security and value transfers

An intelligent agent needs to find a large number of resources in order to make a decision. Some of those resources will be public, but others will be private to a group or to an individual. They will be protected using the public key infrastructure, in much the same way as it is used in XML spaces. Similarly, there will be arrangements for transfers of value, for example payments for viewing documents, built into the system.

© Copyright the University of Oxford and contributors.

Agents will have access not only to billions of primary resources but also to a large number of resources providing ontological information about them. Anyone can set up semantic web resources and there will often be conflicting evidence about the meaning of resources. The way this will be resolved is through the properties attached to resources. These properties, many of them authenticated using PKI, provide a **web of trust**, which will enable people and inference engines to sort the wheat from the chaff. I will prefer resources authenticated by people I trust, or failing that by people who in turn are authenticated by people I trust. Core and subject-related ontologies will emerge through natural selection to form the de facto semantics of the web, but there will always be a wide range of other ontologies connected with organisations and subject areas adding finer detail to the core ontologies.

Ubiquitous computing

This is a term used in several areas of computer science, encompassing the effects of radio networks, third generation mobile technology and the prospect of ordinary household devices becoming network aware. The implementation of Version 6 of the Internet Protocol (Ipv6) will enable each of us to have hundreds of IP numbers, and we will be surrounded by electronic devices that are, potentially, in touch and working together for (we hope) our good. Developments like e-paper (based on a kind of polymer LED technology) and projected micro displays suggest we will eventually *wear* our computers, which will be constantly scanning the electronic neighbourhood for other devices.

The existing arrangements for cooperative working, for example the Microsoft plug-and-play protocols and the 802 wireless networking standards are prescriptive and generally need some kind of intervention by the user. In the future it will be necessary to have protocols that allow all devices to discover services available in the neighbourhood and to negotiate their use. The ultimate objective is 'serendipitous interoperability'. Devices never intended for each other and meeting for the first time should be able to determine what each does and, if appropriate, to negotiate some cooperative activity. If a friend gets a call on his new mobile phone, my hi-fi system will be informed of the call and will reduce the volume until the call is over. (This is a trivial and trite example, but perhaps you could think of something more interesting and exciting. If so, please share your thoughts on the group forum.)

References

Berners-Lee Tim, Hendler Janes and Lassila Ora (2001), 'The Semantic Web, *Scientific American*, May, also available online at https://kask.eti.pg.gda.pl/redmine/projects/sova/repository/revisions/master/entry/doc/Master%20Thesis%20(In%20Polish)/materials/10.1.1.115.9584.pdf, accessed 8 January 2017.

Updegrove Andrew (2005)'The Semantic Web – An Interview with Tim Berners-Lee June, 2005 available online at https://kask.eti.pg.gda.pl/redmine/projects/sova/repository/revisions/master/entry/doc/Master%20Thesis%20(In%20Polish)/materials/10.1.1.115.9584.pdf, accessed 8 January 2017.

Berners-Lee, Tim, 1989 'Information Management: A Proposal', available online at http://www.funet.fi/index/FUNET/history/internet/w3c/proposal.html, accessed 8 January 2017.

Berners-Lee, Tim, 2001 'Business Model for the Semantic Web', available online at http://www.scientificamerican.com/article.cfm?id=the-semantic-web, accessed 8 January 2017.

© Copyright the University of Oxford and contributors.