Chapter 1

Information Systems

Corresponding to every physical system for some aspect of organised human activity there is an *information system*. The information system is a *model* of the physical system. As such, the information system does not contain people, products, places, etc: these are what the physical system contains. The information system contains *representations* of people, products, places, etc.

Models can be used to reason about the thing being modelled. The reasoning we do with information systems has the aims of facilitating control and coördination of the activities of the physical system in achieving some overall objectives.

An information system has three concerns:

- flow of information: the sending and receiving of information can trigger activities;
- transformation of information: information must be processed to convert it into new and useful forms (e.g. summarising, calculating a total, etc.)
- storage of information: information must be kept so that it is available when it is needed.

Obviously in Computer Science we are most interested in the case where the information system is of sufficient complexity to warrant computer support: the model is computerised. In this course we further focus on the third requirement above, namely the need to store information. This implies a repository to hold the information. We will look primarily at databases as a way of providing such a repository.

(Note: Some people draw a distinction between data and information. Information, they say, is "interpreted data", i.e. data that has been made sense of. It follows that computers take in, store, process and output *data*: from a computer's point of view the data has no meaning so it cannot be information.)

1.1 Traditional Computer-Based Information Systems

Prior to the development of the approach which we cover in this course, computerisation of an information system proceeded by a process known as *application-centered development*. Sub-areas of the business would be computerised one at a time in succession, e.g. first accounts, then sales order processing, then inventory, etc. For each sub-area a 'stand-alone' system would be built:

- the system would have its own application programs;
- the system would be based around *files* of data. The programs mentioned above would make use of a number of these files; the files *might* be shared between several programs but more usually they would contain data used by only one application and would be structured in a way that was convenient to that application.
- each program would contain a description of the files it made use of so that it would know how to access the file and what data to expect to get from them.

There are problems both with application-centered development as a methodology and with 'stand-alone' systems as a technical solution.

1.1.1 Example

Consider computerising the Computer Science department of a provincial university in an application-centered way.

Perhaps a decision is made to deal with student registration first. We devise a file of student details and we write a program that inserts new students, modifies details about existing students, and deletes departing students:

Student File

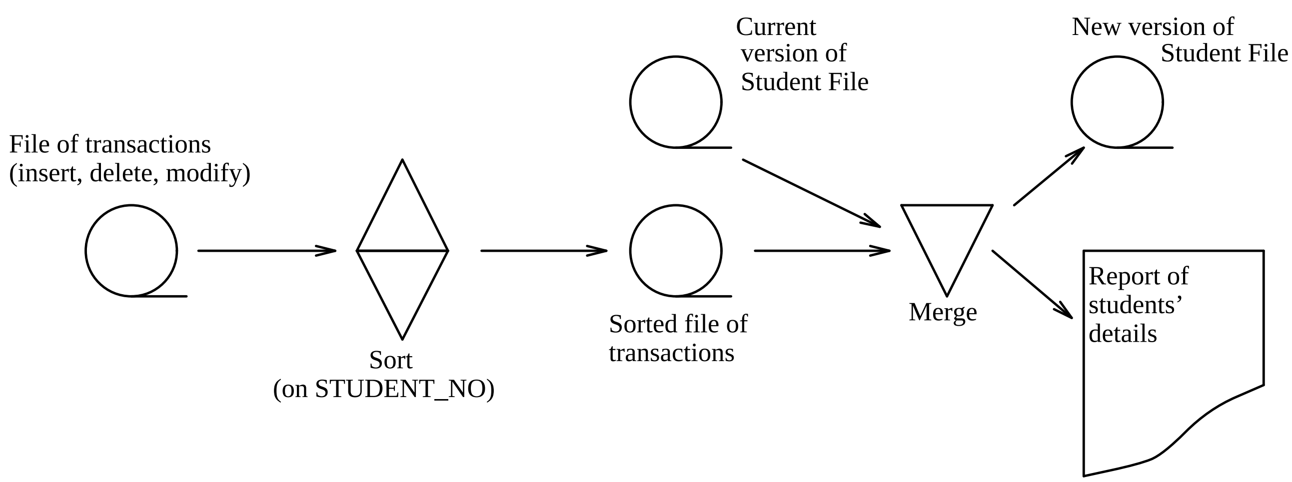
STUDENT RECORD

STUDENT_NO
NAME
ADDRESS

(Sorted on STUDENT_NO)

Updating program

-run every 6 months
-adds new students, deletes departing students, changes names and/or addresses



Some time later it is decided to computerise course enrollment too (i.e. which students take which course units, what exam marks they get for those units, etc.). A separate system is developed with its own file and its own program:

Enrollment File

ENROLLMENT RECORD

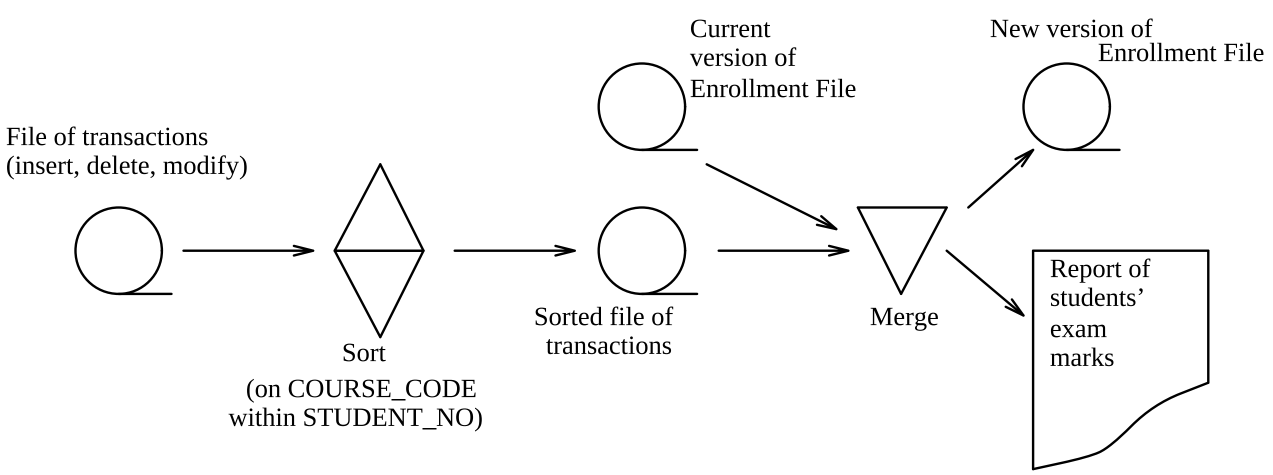
STUDENT_NO
NAME
ADDRESS
COURSE_CODE
EXAM_MARK

(Sorted on COURSE_CODE within STUDENT_NO)

Updating program

-run every term

-adds new enrollments and exam marks, changes names and/or addresses (to allow marks to be sent to students)



1.1.2 Problems that Arise

The following problem arise through using applications-centered development and stand-alone systems (based on files).

Inconsistency

The data in files might get 'out of step', e.g. the files might record different addresses for a student. This is caused by uncontrolled redundancy of data ('duplication'[1]). You can write programs or instigate manual checks

[1] Duplication of data is bad partly because it wastes spaces but more especially because of the possibility of inconsistency. However, it is possible to have controlled duplication that avoids consistency problems.

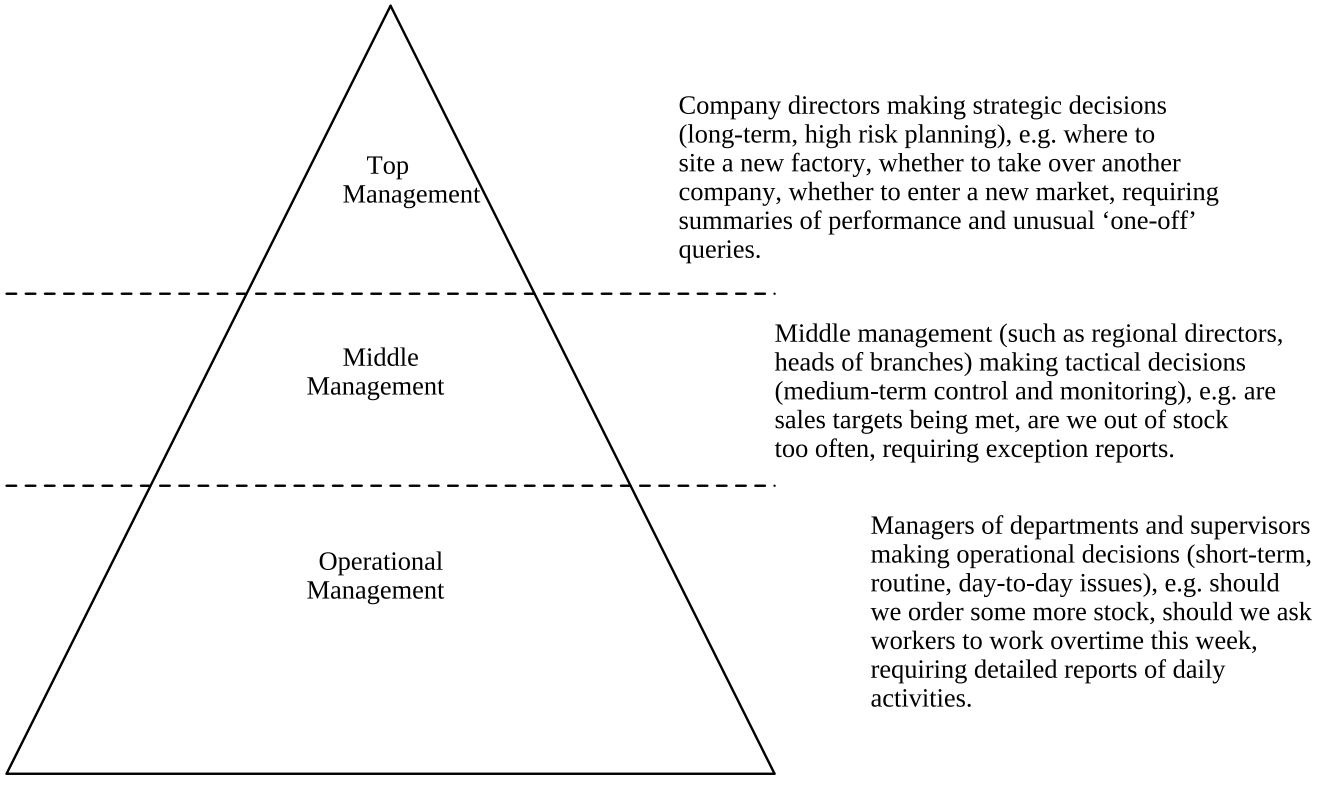
to ensure consistency. However, this becomes increasingly costly, unreliable and complicated as more applications are added.

Inflexibility

Each system is difficult to change. For example, to add a field to STUDENT RECORD entails recompiling all programs which use STUDENT FILE (as each has its own description of the file built in). As another example, to respond to a new requirement requires a lot of work (changes to files, changes to programs, new files, new programs).

Failure to meet business needs

This is a consequence of the inflexibility. Here are the management information needs of a typical business:



All but the lowest level are ignored in traditional computer systems. The higher levels require more unusual processing (even ad-hoc, one-off summaries, etc.) for which programs must be written specially (by which time the information is probably too late).

Other problems...

The approach is also characterised by:

- low productivity (limited sharing, high maintenance, therefore applications backlog);
- poor data security (due to lack of control);
- poor enforcement of standards (absence of a plan).

Advances in three areas have made solutions possible to all the above problems:

- hardware (networks, direct access storage devices);
- software (database management systems, fourth-generation languages (4GLs));
- techniques (data modelling, database theory).

1.2 Database Systems

1.2.1 What is a Database System?

A database system is basically a computerised record-keeping system. (*THIS IS NOT AN ADEQUATE DEFINITION*) Operations are provided to add, modify and delete file definitions (= define the structure of the database), and to insert data into, modify the data within and delete data from existing files, and to retrieve data from existing files. But, all of these are available in a simple file system. What is special about a database system? It is more than just a collection of files. There are notions of *integration* and *centralised control*:

> A database system is a repository of integrated data kept under centralised control.

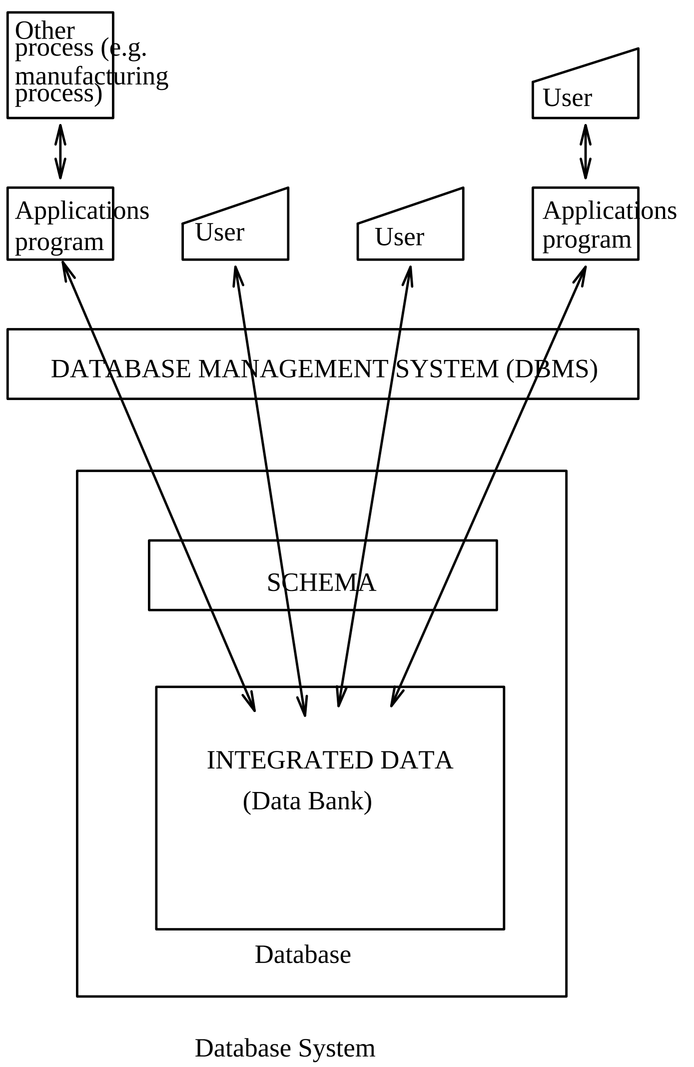
(*THIS MIGHT BE A DEFINITION*) We must examine the key concepts of integration and control.

Integration: This is more than just the lumping of the files together. There is also some imposition of structure. Integration on its own gives us a data bank; we need control to get a database system.

Centralised control: Note that it is control that is centralised, not necessarily the storage of the data. (A distributed database stores, and even replicates, data at different nodes in a network, but there is still some notion of centralised control.) Control is provided by:

- **Schema:** a description of the data kept in the system, kept separate from the application programs;
- **Database Management System (DBMS):** software which intercepts all database accesses. In principle, no access may be made to the database except through the DBMS.

Here's the above in pictorial fashion:



1.2.2 What are the advantages of database systems?

The supposed advantages of a database system are that we can overcome all of the problems with traditional, stand-alone, file-based systems:

- We can achieve consistency:

  - integration reduces redundancy (i.e. the unneeded duplication) of data thus saving space but, more importantly, making inconsistency less likely;
  - note that there may still be duplication but now it is controlled by the DBMS;
  - in fact, in principle, all sorts of integrity checks may be put into the schema and enforced by the DBMS.

- We can achieve flexibility:

  - having a schema means that application programs do not need to contain descriptions of the data: a change to the schema might not affect an application program (data independence —see later).

- We can more easily meet business needs:

  - modern database systems provide high-level query languages to allow ad-hoc access.

- We can solve some of the other problems associated with traditional systems:

  - data can be shared (thus the needs of new applications may be satisfied without the need for new files);
  - security restrictions can be applied;
  - standards can be enforced;
  - conflicting requirements can be balanced (we can work towards the 'best' database structure for the organisation as a whole).

The use of databases has also had a major spin-off (arguably more important than database technology itself). This spin-off is the emergence of a new philosophy for systems development. We have moved form an application-centered viewpoint to a data-centered viewpoint. Organisations now recognise that data is a valuable corporate *resource*. Therefore, it is crucial to take an engineering approach to database development in which we must plan for maintainability, reliability and understandability. This new philosophy is that of data modelling or data analysis.

1.2.3 But all is not rosy...

Along with the advantages, there are problems with database systems, e.g.:

- Cost:

  - Costly, specialised personnel may be required;
  - A DBMS is a large, complex piece of software:
    * While cheap PC DBMSs (£200) can be bought, professional products can cost between £30,000 and £100,000 and are therefore very expensive to buy.
    * There are usually recurrent maintenance costs (e.g. paying $>$ £10,000 in some cases for licences that entitle you to upgrades and software support).
    * Training costs can be high for everyone from programmers to end-users.
    * You may require better (more expensive) hardware, e.g. a machine with more memory or a faster processor.

- Sharing data may actually increase security and privacy problems (contrary to what I said in the Advantages section).
- Sharing data may increase integrity and reliability problems (contrary to what I said in the Advantages section).

...and there can be problems with the data modelling philosophy:

- Data modelling is lengthy and difficult:
  - It requires expensive trained personnel.
  - The working system can be a long time in the making.
- Data modelling may have over-redressed the balance. Originally the emphasis was too much on the processing needs of individual applications. Now the emphasis may be too much on the data. (Object-orientation is being proposed as a solution.)

WARNING: Some pieces of software marketed as PC database systems are little more than file systems. They possess few of the advantages of real databases, but attempt to gain a marketing foothold by using the word "database". (But, in fairness, I should also say that many PC database systems are the real McCoy, and very good bits of software.)

1.2.4 Components of a Database System

To get an overview of the facilities that a typical modern database system offers, there is a simple on-line tutorial you can run. It purports to give an overview of the particular database system which our Department owns, namely R:Base.

INSTRUCTIONS: At the login prompt (where you normally type `orbit`), type `login`. Enter your user id and the password which I have issued to you when prompted. This logs you in to Student2[2]. Note that the operating system is now DOS, not UNIX.

To run the tutorial, change directory:

```
cd \apps\rbase\tutorial
```

and invoke the tutorial:

```
tutorial
```

After some nonsense about testing the cursor keys and entering your name, a menu is displayed. Highlight lesson 1 (A Quick Tour of R:BASE) and press **Enter**. Despite what the next screen says, it should only take 10 minutes or so to work through it. I'm afraid it's a bit patronising, but it provides a good overview.

The main ideas to gather from tutorial lesson 1 are that the DBMS provides facilities that allow a variety of ways of inserting, modifying and deleting data, and a variety of ways of retrieving and displaying data. (DBMSs offer many other facilities too, which we will look at later.)

Note that there is no need to run lessons 2-11 of the tutorial at this stage. I will indicate when it is worth using them (much later in the term). (Obviously, if you can't wait, then go ahead: they're pretty much self-explanatory.)

1.3 Exercises

EXAM QUESTION: Question 7, Part (i) only, 1992

In the Computer Science Department of a provincial university, it is necessary for timetabling and exam registration purposes to keep track of which students are taking which courses. The present system is run by Dennis Brick (timetabling officer) and Danny Burgess-Hart (exams officer), and uses various text files and Pascal programs as follows:

Dennis Brick uses a text editor to create a file of the following format (shown with some sample data):

[2]Note: please do not abuse this access to Student2. If anyone does, then *all* of your Student2 accounts will be deleted immediately and without warning.

```
Student Name    Degree          List of codes for courses
                                taken by the student
---------------------------------------------------------

ABLE, John      Comp Sci        CS21 CS22 CS23 CS24

BRIGHT, Clare   Comp Sci/Maths  CS21 CS24
```

Dennis uses electronic mail to send a copy of this file to Danny Burgess-Hart, and also runs a Pascal program which provides him with timetabling information (e.g. number of students per course, courses which can be timetabled concurrently, etc.).

Danny Burgess-Hart runs a Pascal program which takes in the file he has been sent by Dennis and outputs a file containing exactly the same data but which inserts asterisks as separators. For example, for the file above it produces:

```
ABLE*John*Comp Sci*CS21*CS22*CS23*CS24

BRIGHT*Clare*Comp Sci/Maths*CS21*CS24
```

This is now in the correct format to be used by another of Danny's Pascal programs, which produces a file of the following format:

```
Exam Code     Student No     Student Name     Degree
------------------------------------------------------------

EX52          1              ABLE John        Comp Sci

EX52          2              BRIGHT Clare     Comp Sci/Maths

EX53          1              ABLE John        Comp Sci

  .             .                  .              .
  .             .                  .              .
  .             .                  .              .
```

i.e. this program has assigned every student a unique identifier, has translated course codes into exam paper codes (e.g. from CS21 to EX52) and then ordered this by students per exam.

In the meantime, Dennis Brick will have used electronic mail to send Danny messages about any students who Dennis knows to have revised their choice of courses. Danny must update his files using a text editor.

Finally, Danny sends the output of his program to the administrative centre of the university, Helplesston Hall, where exam details are coördinated.

[12 marks] Describe in detail the problems that can arise using the approach described above.

Postscript: Dennis Brick has now taken over as exams officer and has rationalised these systems.

Chapter 2

Information System Development

We are now going to look at the stages of design required to create an information system. Obviously, we concentrate on the design of *computerised* information systems and we will specifically be aiming for the design of *a database system*. However, the first stage of the methodology that we consider (namely, conceptual data modelling) is useful whether the information system is computerised or not, and the second stage (logical data modelling) is useful for all sorts of computerised information systems (e.g. databases and stand-alone file-based systems). Only when we reach the third stage (implementation) do we need to specifically consider whether we're to use a database or a file-based system, and, in the case of a database, what type of database it is to be.

Obviously, we need a methodology. A methodology is a coherent collection of methods and techniques with supporting tools. Using a methodology should improve your productivity and the quality of the finished product: it can help you to achieve consistency, completeness, clarity and control by telling you what to do next and giving you standards and management procedures for each step of the process.

The disadvantages of methodologies include that not all the stages are applicable to every project (how do you know which are applicable to your project?), there may seem to be too much admin., and methodologies can seem like overkill. People who reject methodologies using these arguments are missing the point —using a methodology may be costly but it does make it more likely that you will meet your requirements; on the other hand, using a more ad hoc approach may seem cheap, but if you fail to meet your requirements then even greater costs may ensue.

2.1 Structured Systems Analysis and Design Methodology

The methodology we shall look at briefly is the Structured Systems Analysis and Design Methodology (SSADM). This was developed by a UK Government department and is now a Government standard. Many companies now use it.

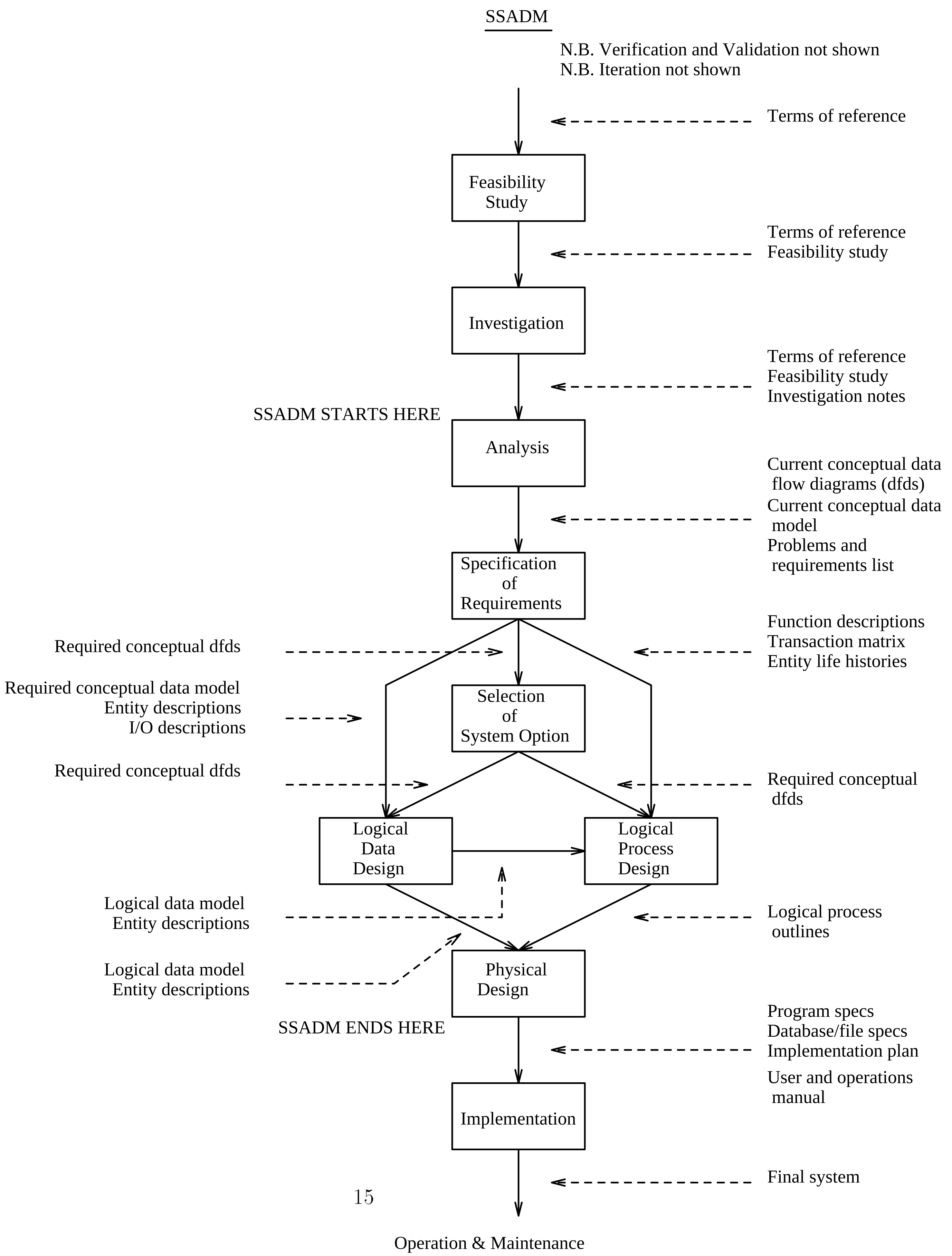
It specifies a *life-cycle* for systems analysis and design, i.e it specifies an ordered series of stages, each stage transforming its inputs into a set of outputs, e.g. one stage might transform a requirements specification into a design specification. Crucially, each stage ends with a verification and validation step.

Verification: given the specification produced at the ends of a stage, verification checks the internal consistency of the spec. Thus this is a kind of 'syntax' check. The designer is asking "Have I applied the transformation right?', 'Is the spec. a legal spec.?', 'Is the spec. right?'

Validation: given the specification produced at the end of a stage, validation checks the consistency of the spec. against the requirements. Thus this is a kind of 'semantic' check. The designer is asking 'Does this make sense?' 'Does it achieve my goals' 'Is it consistent with my goals?', 'Is this the right spec.?'

Iteration of a stage or stages is necessary in SSADM until verification and validation give the OK.

The diagram shows the main stages in SSADM.


SSADM
N.B. Verification and Validation not shown
N.B. Iteration not shown
Terms of reference
Feasibility Study
Terms of reference
Feasibility study
Investigation
Terms of reference
Feasibility study
Investigation notes
SSADM STARTS HERE
Analysis
Current conceptual data flow diagrams (dfds)
Current conceptual data model
Problems and requirements list
Specification of Requirements
Required conceptual dfds
Function descriptions
Transaction matrix
Entity life histories
Required conceptual data model
Entity descriptions
I/O descriptions
Selection of System Option
Required conceptual dfds
Required conceptual dfds
Logical Data Design
Logical Process Design
Logical data model
Entity descriptions
Logical process outlines
Logical data model
Entity descriptions
Physical Design
SSADM ENDS HERE
Program specs
Database/file specs
Implementation plan
User and operations manual
Implementation
Final system
Operation & Maintenance

Here are some brief words about each stage.

Feasibility study: are the objectives realistic (whether they be technical, financial or social)? This is a mini-SSADM project in itself involving a rough investigation, analysis, requirements specification and system selection.

Investigation: detailed study of existing system.

(SSADM begins now...)

Analysis: construct conceptual data model and conceptual process model of *current* system; document problems with the current system and requirements for the new system.

Specification of requirements: construct conceptual data and process models of the *required* system.

Selection of system option: select from a shortlist a new implementation (a hardware, software, database and peopleware configuration) which will be developed further, based on costs, timescales and degree of fit with requirements.

Logical data design: complete a detailed logical data design specification.

Logical process design: complete a set of detailed logical process designs.

Physical design: translate logical data design into file or database specifications, and translate logical process design into program specifications.

(...SSADM stops here.)

Implementation: translate specifications into an operational system (i.e. do the programming and the database/file definitions).

Operation: use the system, review it, and modify it as necessary.

SSADM goes on to break these steps down into smaller steps and shows how to produce documentation for each step. SSADM is 'data analysis'-inspired: many of its workproducts (entity models, transaction matrices, etc.) and its techniques (e.g. normalisation) come from ideas about database system development.

Rather than pursue SSADM any further, we shall return to looking at the development of the data repository in isolation.

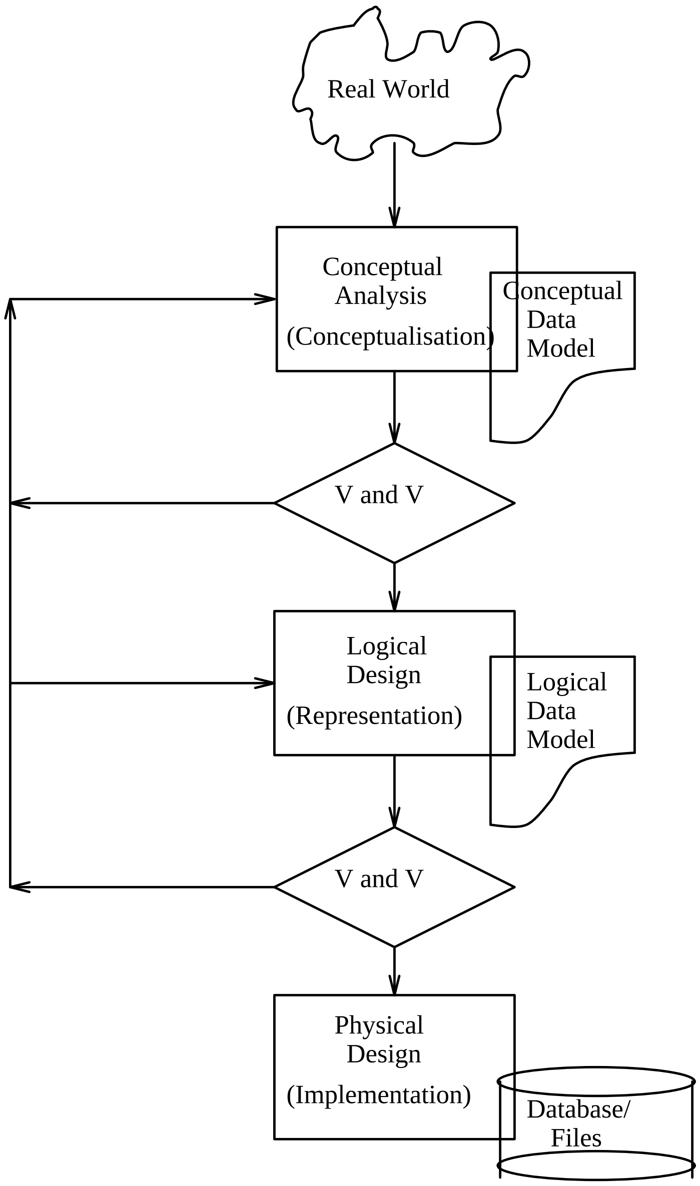
The basic approach we will follow is based on Whittington, chapters 8-13, which is probably the most comprehensive, accessible description to be found in the literature. It is worth consulting Whittington for two reasons (a) he gives more guidance on points of detail, and (b) he gives additional examples. (However, watch out for the differences between his presentation and mine.)

Bear in mind the following two points about my and Whittington's descriptions of data analysis and design:

- we do not show the interactions of data analysis and design with process modelling that we have been discussing in the context of SSADM. In this course, we treat the design of the storage facility of an information system as if it could happen in isolation. Clearly this is a simplification. After all, what you store is partly determined by who needs the information and what processing you want to do with it. (Indeed, to some extent, striking the balance between data design and process design is still an open issue: object-oriented design, for example, has recently gained in popularity as a way of getting this balance right, but its details have yet to be spelled out.)
- we will not be drawing a distinction between the *current* system and the *required* system. We will assume that we want an information system that faithfully captures the current system.

2.2 Data Analysis and Design

We proceed through a succession of models, until we are finally ready to implement the system.



Conceptual Analysis: the process of conceptualisation; also known as conceptual data modelling (sometimes when people talk simply about data modelling they mean conceptual data modelling), and also known as data analysis. From an examination of the 'real world' (the so-called universe of discourse), we produce a conceptual (data) model. This is a high-level model showing entities and relationships, unconstrained by any particular thoughts about computerisation. We will use entity-relationship (ER) diagrams for this.

Note: **this is a description of reality, not of the data.**

Logical Design: the process of representation (or, perhaps, formalisation). This is the first design step. The conceptual model is mapped to a

collection of implementation-independent data structures We will use a set of relations for this —see later; this does not commit us to using a relational database management system —see later. This model remains abstract, but is at least more amenable to a computational treatment.
Note: **we have now switched to a description of the data but we are not constrained by implementational choices.**

Physical Design: the process of implementation. Whittington splits this into three phases which he calls structural design, physical design and implementation. In this course, we treat them all as one stage. At this point we decide whether to use a database or a file-based system. And, if we are to use a database, we decide what sort of database management system to use (relational, network, hierarchical, etc.), and which particular commercial database management system to use. We then map the logical data model onto the various structures that our chosen implementation platform supports, e.g. issues of distributed databases, storage structures, and typing the whole thing in and getting it going. In this course, we concentrate on implementing a relational database.
Note: **by now we're taking into account all implementational constraints of the hardware and software that we are using.**

2.3 Rapid Prototyping

Prototyping tools allow the rapid construction of a database and some of its associated programs. The prototype so-constructed is a rough mock-up that allows one to demonstrate what the final system could be like.

Many database systems now come with 4th Generation Languages, report generators, form generators and application generators that facilitate the construction of prototypes that can be refined (e.g. extra code can be written to perform data validation checks, etc. that were left out of the prototype) once the user has expressed satisfaction with the look and feel of the prototype. Showing the user a quick-and-dirty solution is supposed to allow the early incorporation of their comments into analysis and design to ensure that the final system meets their needs.

Some people take this idea to extremes and argue for *rapid prototyping* as a methodology in itself, i.e. you repeatedly construct or refine prototypes until you get one the user is happy with. This one becomes the operational system.

For small projects, this might be fine but speed of development must be weighed against the risk that the approach's lack of rigour leads to other problems, e.g. you might miss requirements, the final prototype might not be robust or efficient enough to be your operational system, you will have to do a lot of testing of the final prototype because this is your only guarantee that you've got what you need, and you will probably have no documentation to accompany your system.

For more important projects, producing a conceptual data model is an absolute 'must'. Prototyping still has a role to play though as it can be useful as a verification and validation technique. If you use it in this way you retain the control afforded by a life-cycle approach but you also incorporate a high degree of user feedback. However, the exact relationship between prototyping and database development is an open issue.

Chapter 3

Conceptual Data Modelling

Conceptual data modelling is the process of building an abstract model of the universe of discourse. It is supposed to be completely uninfluenced by all thoughts of computerisation. We are building a model of the real world.

The conceptual data model can be used for a variety of purposes, e.g.:

- general discussion, consideration of organisational goals;
- suggesting changes to clerical procedures;
- the beginnings of the design of a traditional file-based system;
- the beginnings of the design of a database system, whether it be a navigational (network or hierarchical) database, an object-oriented database or a relational database.

The main virtue of most conceptual data models, and certainly of the conceptual data models that we will consider, is that they are pictorial. They are thus an excellent communication tool.

3.1 ERA

ERA is the label for the approach we will use. The initials can stand for a number of things, but the approaches differ only superficially:

- Entity Relationship Attribute (Whittington)
- Entity Relationship Approach (Chen)
- Extended Relational Analysis (Addis and, as a young man, Bridge)

The idea is to model the world in terms of

- entities and entity types,
- relationships and relationship types, and
- properties and property types.

Note: I use "property" where others use "attribute". We will encounter another use of the term "attribute" later, and I like to keep these two pieces of terminology distinct. Other writers are not so particular. I won't impose penalties if you use them interchangeably.

3.2 Entity Types

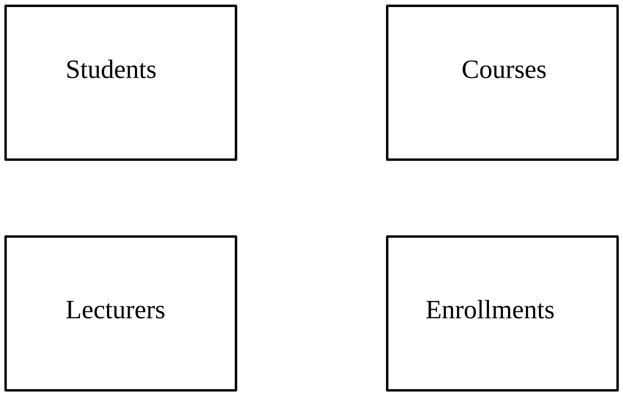
An *entity* is

- a distinguishable thing of interest, and/or
- a significant object of thought.

Entities are classified into *entity types*. (Try to avoid sloppy interchange of the terms "entity" and "entity type".) For example, the object called E.White is an entity of type Students, the object called DAD is an entity of type Courses, the object called D.G.Bridge is an entity of type Lecturers, and the object that we can refer to as E.White/DAD is an entity of type Enrollments.

(The last example gives us a clue as to how to spot an entity: it is anything with properties. Hence, the object referred to as E.White/DAD is an entity and not a relationship because it has examination grade as one of its properties.)

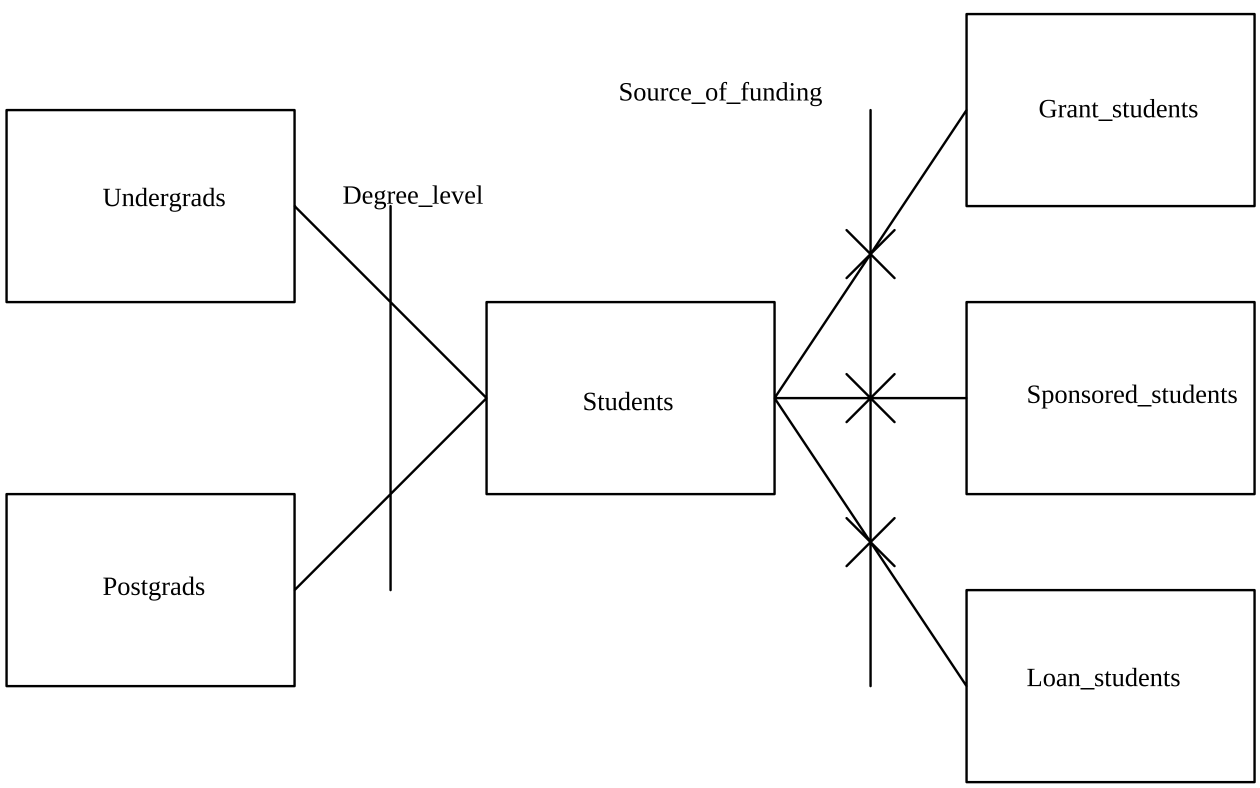
Diagrammatically, we use a labelled box for entity types.



Entity types can have *subtypes*, e.g. Undergrads are a subtype of Students; Postgrads are another subtype of Students. Whether a particular student entity is a member of the subtype Undergrads or the subtype Postgrads will depend on a property (see later for what these are) of the student, namely his/her Degree_level. We shall assume that this subcategorisation is *exclusive*, i.e. if you are an Undergrad, you cannot be a Postgrad and vice versa, i.e. Undergrads $\cap$ Postgrads = $\{\}$.

Another example of subtypes of Students depends on the property Source_of_funding, which subcategorises Students into Grant_students, Sponsored_students and Loan_students. In this case, the three subtypes are *non-exclusive*.

Diagrammatically, boxes are used for subtypes with arcs between a subtype and its supertype. A bar with a property shows the basis for the subdivision. The subtypes will be assumed to be exclusive unless starred:



In the interests of not cluttering the diagram, other information should be noted separately, e.g. in a margin somewhere. Examples might be that Undergrad $\cup$ Postgrads = Students, but Grant_students $\cup$ Sponsored_students $\cup$ Loan_students $\neq$ Students (because some students have other sources of funds, or none at all). Another example is that Postgrads $\cap$ Loan_students = $\{\}$ (because Postgrads cannot participate in the Government's loans scheme).

3.3 Relationships

A relationship is an association between entities. Some examples:

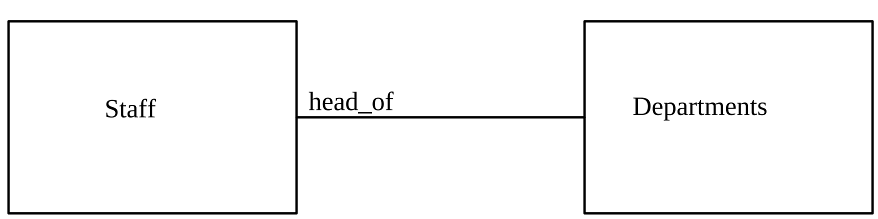
- there is a relationship between Staff and Departments which associates some (but not all) staff with the Departments entity of which they are the head;
- there is another relationship between Staff and Departments which associates Staff entities with their departments;
- there is a relationship between Lecturers and Students which associates Lecturer entities with the Students they teach.

We classify relationship types between entity types by their maximum cardinality:

- *one-to-one* (1:1) (as with the first example above)
- *one-to-many* (1:m) (second example)
- *many-to-many* (m:n) (third example)

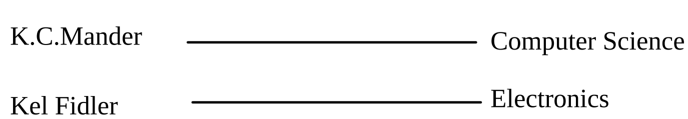
In the above, "one" can mean one or none; and "many" can mean many, one or none, e.g. if entity type A is related to entity type B by a one-to-many relationship type, this means that an entity of type A is related to zero, one or several entities of type B, and an entity of type B is related to one or zero entities of type A. This is why I said we were characterising *maximum* cardinalities.

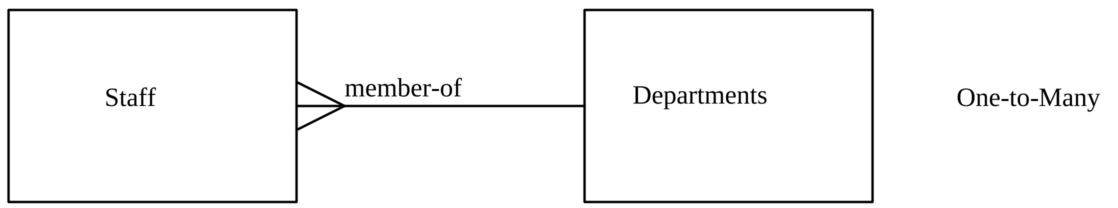
Diagrammatically, we use arcs between entity type boxes for relationship types. Labelling the arcs can make things clearer, particularly when there is more than one relationship between entity types.



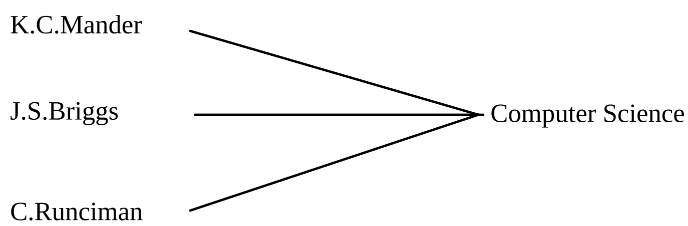
One-to-One

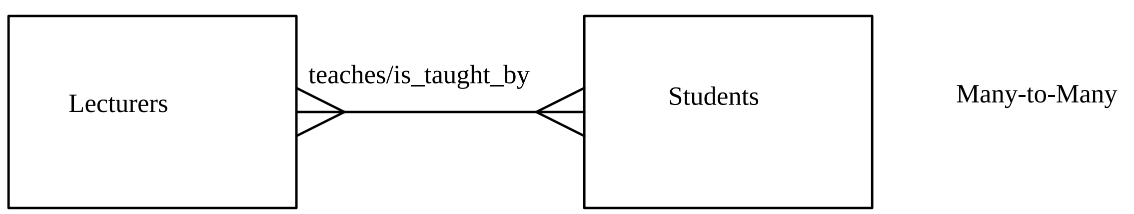
i.e.



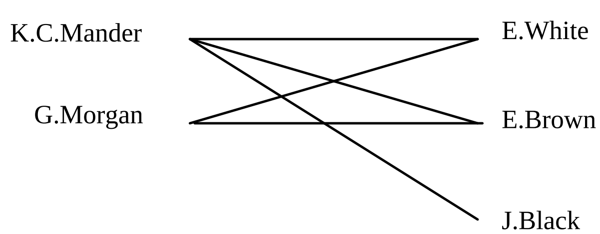


i.e.





i.e.



In all cases there may be other information to record. For example, minimum cardinality (e.g. must every member of staff be related to a Department? must a Department have members of staff in it?), even more precise cardinality (e.g. one tutor is related to between 0 and 4 first year tutees), whether being related to an entity through one relationship excludes you from being related by another, or whether being related to an entity through one relationship obliges you to being related by another (e.g. head_of relates a staff entity to a Department, and that same entity must also be related to that department by member_of).

3.4 Properties

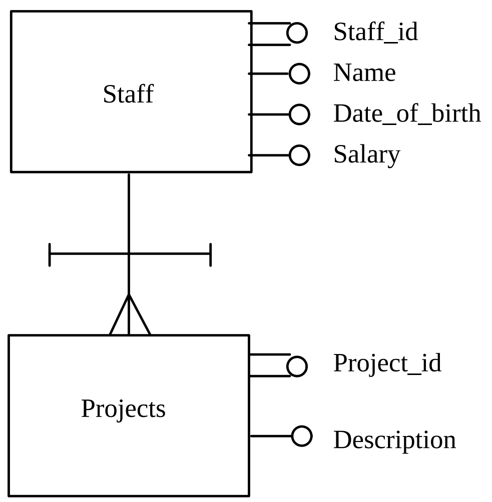
A *property type* is a characteristic or descriptor of an entity type. A property for a particular entity has a *value*, which is drawn from a *value set*. Examples of properties of the Staff entity type are Staff_no, Name, Date_of_birth; an example for Enrollments is Exam_grade.

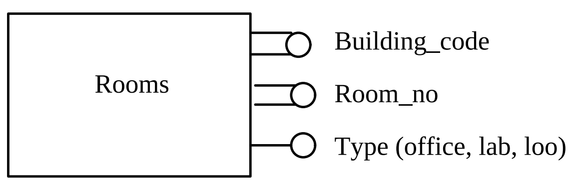
You have to be careful in distinguishing properties from related entity types. For example, Staff have departments. Is Department a property of the Staff entity type or is it another entity type associated with Staff by a relationship? As a guideline, to distinguish these cases, ask whether the suspect object (department in this case) has properties or relationships of its own, e.g. if we are interested in the head, size and budget of departments, then Department is likely to be an entity type. But if we were not interested in them, then Department_name could be a property of Staff.

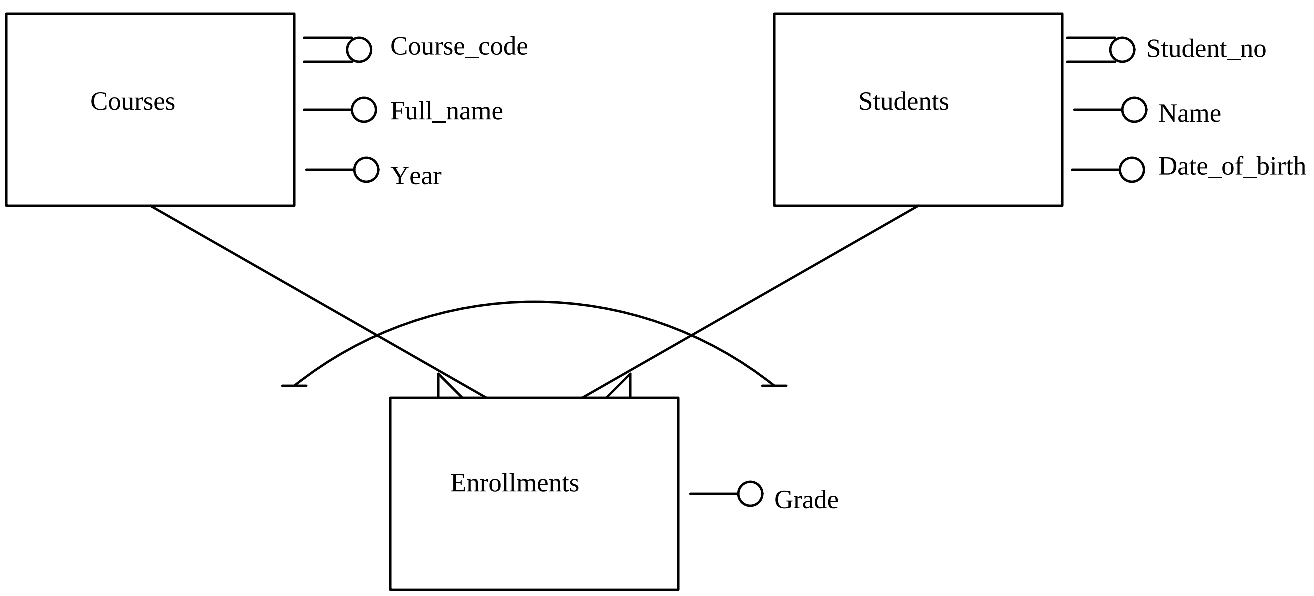
Since entities are individually distinguishable, some property (or combination of properties) of an entity type will take on values which uniquely identify an entity of that type: this property or set of properties is the *identification key*. Examples: for Staff it is Staff_no; for Courses it is Course_code; for Rooms it is Building_code and Room_no in combination (e.g. <XE, 012>, <PD, 017>); for Projects it is Staff_no and Project_no in combination (e.g. <dgb, 1>, <dgb, 2>, <jsb, 1>, ...); for Enrollments it is Student_no and Course_code in combination (e.g. <a1, itp>, <a2, dad> ...).

There is something to note about these last two examples (Projects and Enrollments): identification of entities can be *dependent* on identifying entities of another type. This is called *identification dependence*.

Diagrammatically, we add little attachments for properties. The identification key properties have a double line attachment. Identification dependence is signalled by a bar on the relationship arc connecting the dependent with what it depends on:







3.5 Conceptual Analysis: Some Guidelines

These guidelines are *not* to be followed slavishly.

1. Identify entity types and subtypes

This is very difficult; inevitably you will iterate back to this a lot. Look for things which have identifiers, have properties which we are interested in, engage in relationships with other things, are talked about a lot or are stored in the existing filing system (be it manual or computerised).

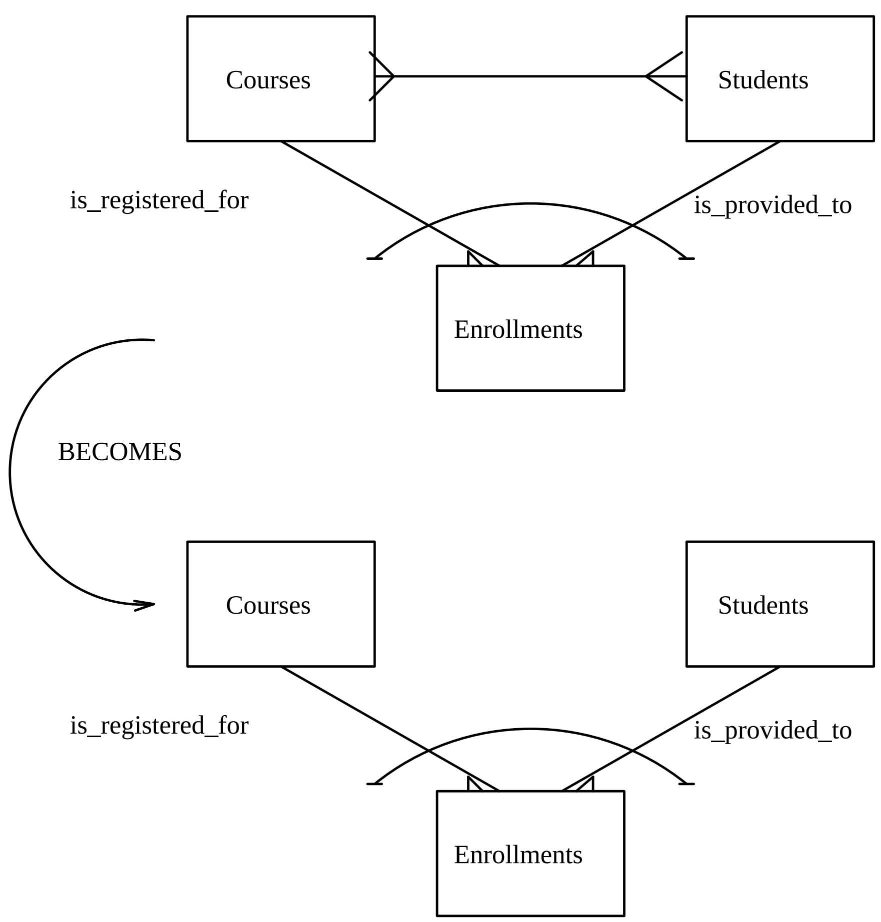
2. Identify associations between these entity types

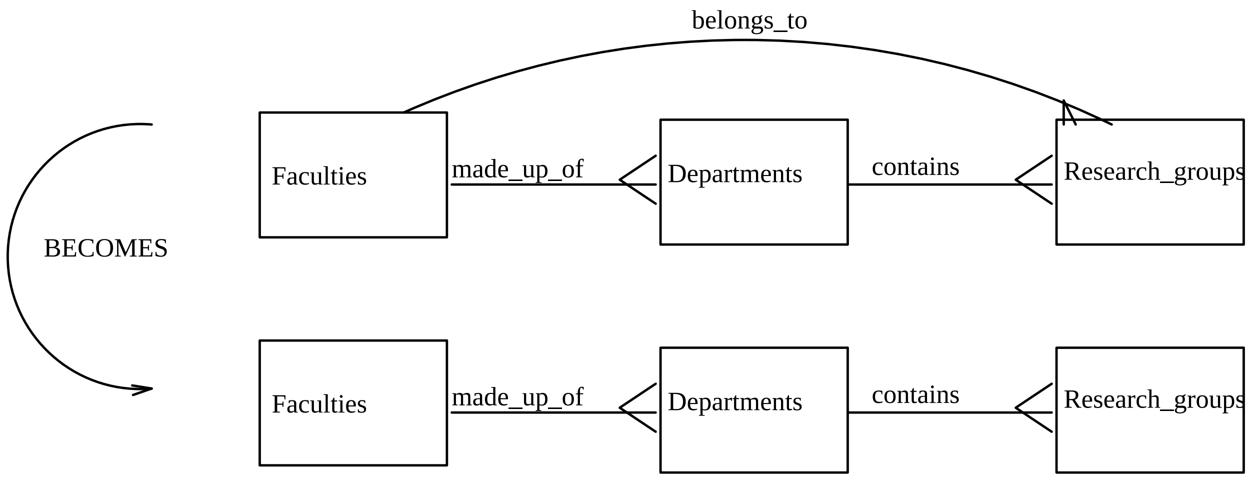
Try to determine which entity types are related. Draw entity types as boxes and interconnect them. If the relationship is subtype-supertype, group the boxes according to the category which determines the subdivision and show whether it is exclusive or not. If the relationship is a straightforward association, mark the cardinality, the identification dependencies and relationship names.

3. Rationalise the diagram

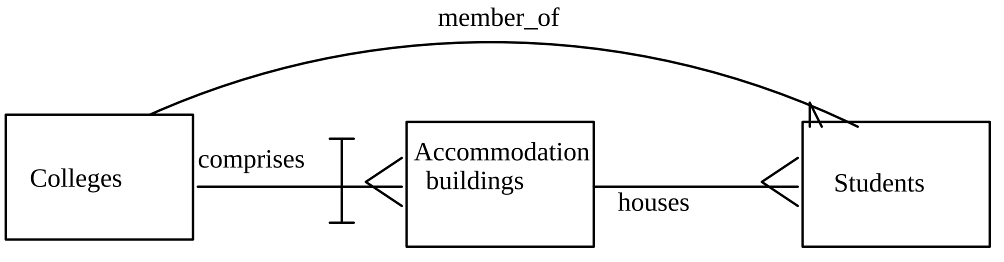
This is split into three tasks.

(a) Remove redundant relationships: Convince yourself that a relationship really *is* redundant before you remove it.





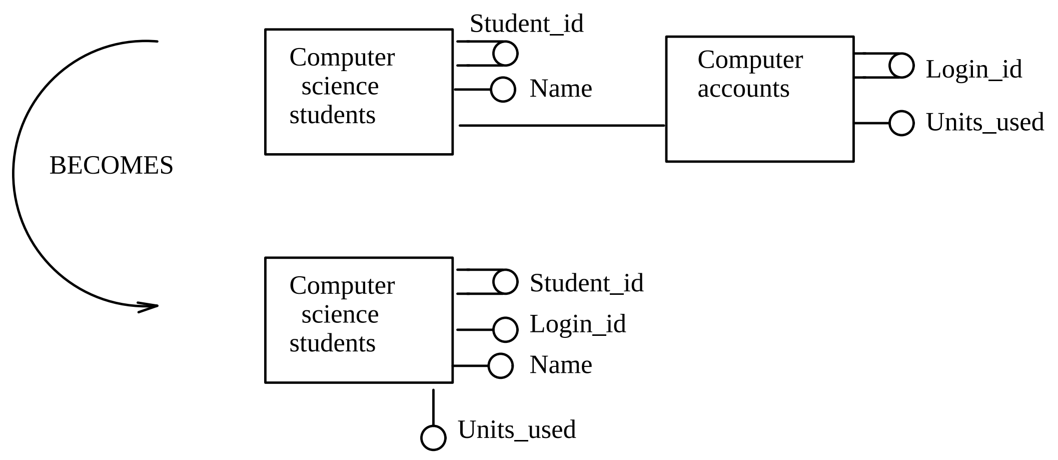
But. . .



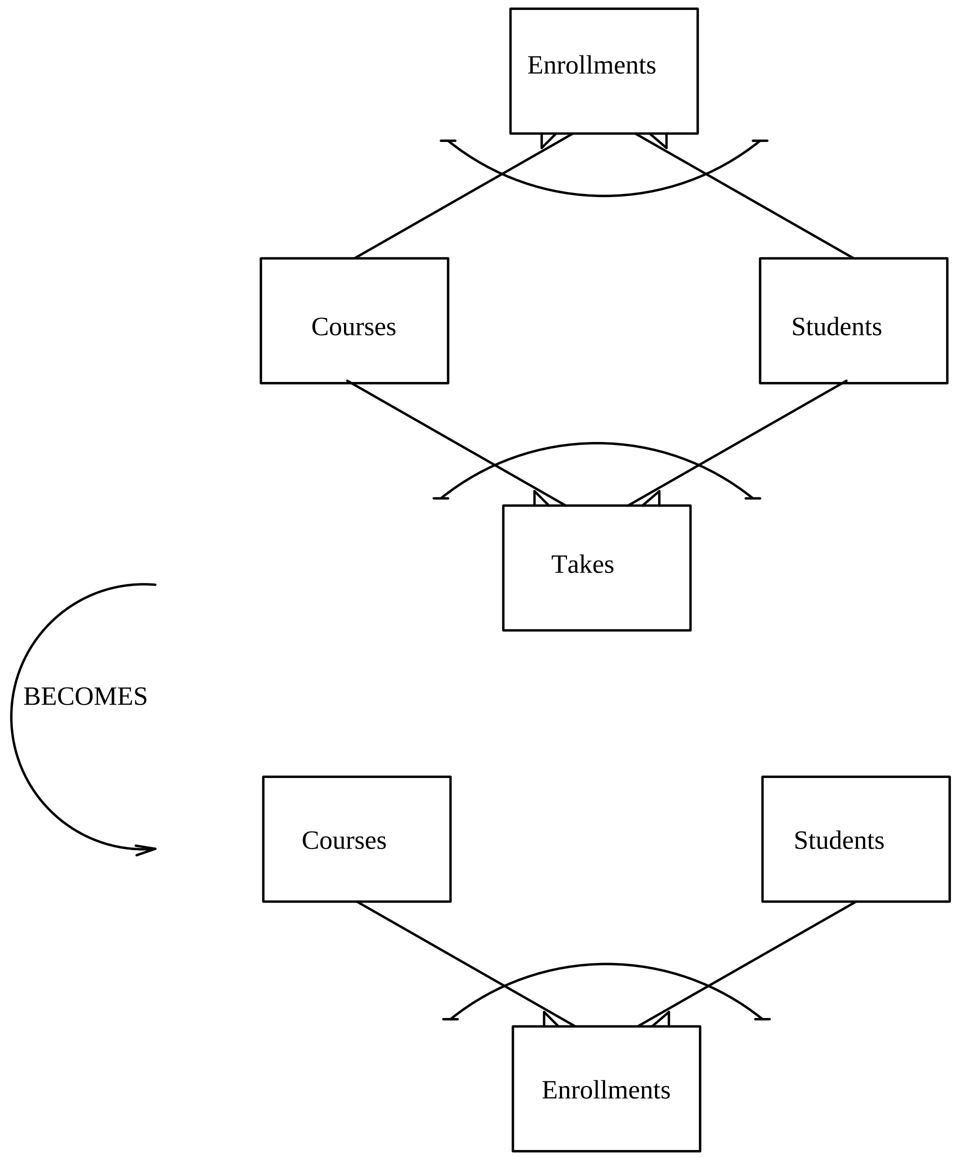
. . . this cannot be simplified since, e.g., E.White is housed in Alcuin 'C' block but (exceptionally) belongs to Goodricke, and J.Black 'lives out' but belongs to Derwent.

(b) Remove redundant entity types.

For example, if two entity types are related one-to-one, you might be able to merge them (if they seem to be the 'same' thing).

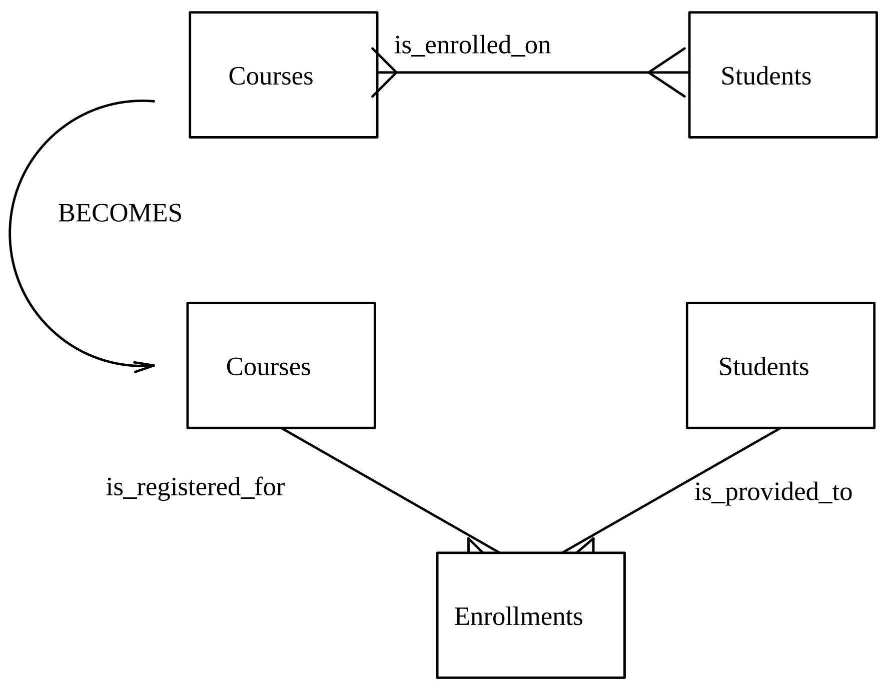


Another example is where an entity type that links two other entity types is repeated:



(c) Remove many-to-many relationships

We introduce a new entity type in place of these relationships. The reason we do this is that it replaces one m:n link with two 1:m links which are conceptually simpler and also, by introducing a new entity type, we tend to get a more robust analysis: properties can easily be added to the entity type in the future. (I have also seen people get their analysis completely wrong when they fail to do this.)



4. Identify properties of entity types

Try to determine the properties of interest and associate them with their rightful entity types.

5. Consider unique identification of entities

Distinguish the identification keys, check that subtypes have the same keys as their supertypes, and check that the identification dependencies identified earlier *do* supply sufficient identification.

6. Verify

Check, e.g., entity type names are all different, all entity types have identification keys and/or identification dependence, there are no many-to-many relationship types.

7. Validate

Discuss the diagram with people who understand the universe of discourse. (Here lies the usefulness of a diagrammatic technique.) Is it complete? correct? natural? simple?

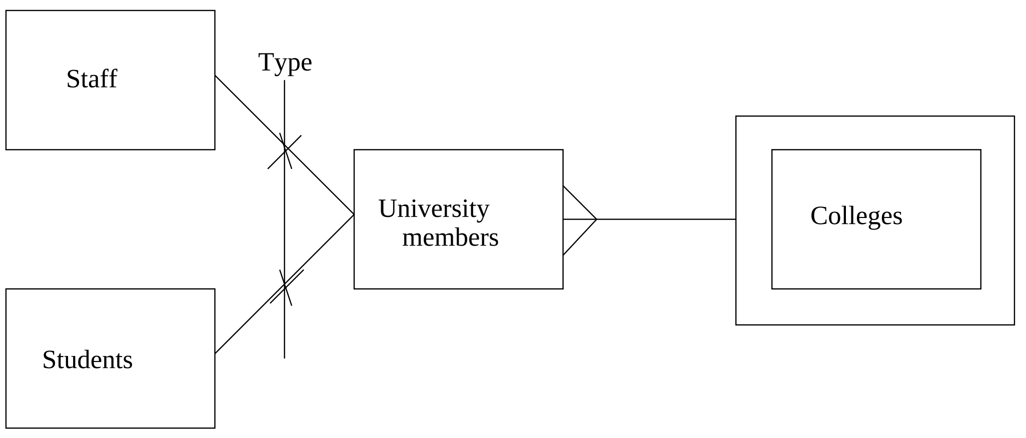
8. Iterate

. . . until your conceptual model is 'signed off'.

3.6 Local Models

We have been proceeding as if we are developing a single *global data model*. But, in practice, different parts of the model might be the purlieu of different people. To help us to break down the analysis task, we could develop a set of *local data models* (one per subarea) and then merge them.

In a local model, we would still want to show connections to other local models. We use a double-edged box to represent a *set* of entity types belonging to an area whose detail is not relevant to the local model.



Here, college details (e.g. conference bookings, bar opening times) are deemed to be at the wrong level of detail for the rest of the diagram.

3.7 Exercises

A) EXAM QUESTION: Question 9, Part (i) only, 1989. (Set by my predecessor, based on an Open University example)

A travel company arranges holidays for its customers.

The company offers holidays in many countries. In each country it arranges holidays in several cities, and in each city it does business in several hotels.

The company deals with several airlines, arranging charter flights to/from the cities where it uses hotels. Each flight flies directly to the city.

The company has several agents.

When a customer wants to book a trip, he goes to an agent.

The customer may have several holidays booked at one time.

Two days after a customer makes his/her contract a confirmation is posted to him/her telling him/her the airline name, the airline code, the flight numbers and times, the airport's names, the name of the city he/she flies from, and the name, address, and telephone number of the hotel.

After the holiday is over, the travel company records which customers have been to which agents, so that the agent can send out mailing lists in future years.

[12 marks] Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

B) EXAM QUESTION: Question 8, Part (i) only, 1990

A video hire shop has a database of its stock and loans.

Films are identified by their titles. Customers are also interested in a film's length and its rating (U, PG, X). The shop may stock several copies of a film on different (physical) video cassettes. Only one film, however, is stored on each cassette.

The shop is supplied by a number of distributors: a particular film is supplied by a single distributor.

The shop's customers are issued with plastic hire cards showing their identification numbers, names, addresses and dates of birth (to prevent X-rated films being loaned to under-18s, etc.). A customer may have several videos on hire at any one time. If a video is on loan, we need to know the date it is due for return. If it is not on loan, the last date it was borrowed is held.

A customer may place reservations on films. S/he will receive the first video that is returned to the shop that holds the reserved film. Since several customers may have the same film on reservation at any one time, a record is kept of the time and date a reservation is made so that customers' reservations may be dealt with in order of reservation.

[10 marks] Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

C) EXAM QUESTION: Question 10, Part (i) only, 1992

KopyRight is a company which supplies photocopiers on a national basis. To deal with customer problems, KopyRight has a set of Help Desks, as described in detail below.

A customer will have a number of sites, each being identified by a site number and having a telephone number and postal address. On a site, a customer will have a number of photocopiers.

The country is divided into geographic regions, e.g. the South-East, the South-West, the Rest. Each customer site falls into one geographic region. Each region has a Help Desk to serve it. A Help Desk has a single telephone number which customers who are experiencing problems with their photocopiers can ring.

Each Help Desk has a number of staff, some of whom are Help Desk Operators, others of whom are Photocopier Experts. (A person can play both roles.)

It will be one of the Desk's Operators who answers the 'phone. S/he will record details of the call such as the site number, the identification number of the photocopier being complained about, the nature of the complaint (in the form of a short piece of text) and the date and time of the call. S/he becomes the 'primary owner' of that call: as such, it is his/her responsibility to make sure that the customer's problem eventually gets resolved.

Sometimes the Help Desk Operator will be able to resolve the problem himself/herself immediately over the 'phone. In this case, s/he records the status of the call as 'resolved'. Often, however, s/he will need to take a so-called referral action.

In referral, the Operator passes the call on to one of the Photocopier Experts who works for the Help Desk. This makes the Expert a 'secondary owner' of the call. The Expert will try to fix the problem either over the 'phone or by visiting the customer's site. If s/he solves the problem, the Expert informs the primary owner of the call that its status is now 'resolved'. The Expert now ceases to be the secondary owner. If, however, the Expert cannot resolve the problem, s/he informs the primary owner that the call is still 'unresolved'. Again s/he ceases to be the secondary owner. The primary owner might now try to pass the call on to another Expert, i.e. to make someone else the secondary owner. This goes on until the call is 'resolved'. (An Expert might even assume the role of secondary owner several times if s/he is invited to have repeated stabs at solving the problem.) It is important to record who the secondary owners of a call have been, along with the date and time the call was passed to them and the date and time they passed it back to the primary owner.

[12 marks] Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

D) NOT AN EXAM QUESTION: Based on Open University material

A country bus company owns a number of buses. Each bus is allocated to a particular route, although some routes may have several buses. Each route passes through a number of towns of a given population. One or more drivers and conductors are allocated to each stage of a route, which corresponds to a journey through some or all of the towns on a route.

Drivers have an employee number, a name, address and a telephone number.

Some of the towns have a garage where buses are kept and each of the buses has one 'home' garage. Buses are identified by their registration number and can carry different numbers of passengers, since the vehicles vary in make and size and can be single or double-decked. Each route is identified by a route number and information is available on the average number of passengers carried per day for each route.

Each garage has a name, address and telephone number. Although each bus has a home garage, that garage need not necessarily maintain the vehicle. In fact, only certain models of bus can be serviced at each garage. Different models have different mileage intervals between services. Records must be kept of each bus's mileage at their last service in order to determine whether they are due for their next service.

A staff of maintenance engineers works at a particular garage, each engineer having an employee number, name, address and telephone number.

Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

E) EXAM QUESTION: Question 10, Part (i) only, 1993

Poopers was established in 1905. The company originally laundered nappies and linen for hospitals. On the death of its founding director, Old Mr. Pooper, his grand-niece, Sally Pooper, took over the company.

Sally knows that a forest the size of Cumbria needs to be set aside every year to keep British babies in disposable nappies. She therefore plans to offer a door-to-door cotton nappy collection, washing and delivery service to environmentally-conscious parents in the Sudbury area. Such a service would need the support of a computerised information system.

In the expanded company, Poopers will own a fleet of Fort Transit vans, the Poopmobiles, each identified by its registration number. To keep the vans properly serviced, Sally will need to know the date of their last service and the number of miles on the clock at last service.

Each van will have a 'main driver' associated with it. This is the employee who has responsibility for that van. A driver will not be the main driver for more than one van. This, Sally believes, will foster pride in drivers for their vans.

Sally is hoping for a much expanded pool of clients (each to be identified by an identification number). Different clients will receive different treatment. For households, Sally will want to hold the names of the parents and the number of nappies to be delivered. For each baby in the household, she will want to record its name and (an inspired touch this) its date of birth so that birthday cards can be delivered.

Other clients will be 'orgnisations' rather than households, such as hospitals and nurseries. Sally will again need the address and the number of clean nappies to deliver. Additionally, she will need the name of a contact person at the organisation.

The information system must be able to print out routes for drivers. A route will visit several clients, and will take place on a particular day of the week. There must be an indication of the order in which clients on a route are to be visited. Some clients, by virtue of being heavy nappy users, will want to appear on more than one route, usually for different days.

Each driver will be solely responsible for a number of routes, usually one route per day. By making a driver solely responsible for certain routes, Sally hopes to be able to provide a friendly, familiar service.

[12 marks] Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

F) EXAM QUESTION: Question 2, 1994

[20 marks]

Sudbury's recently closed colliery was bought up by a cartel of southern entrepreneurs who converted it into a theme park called simply 'Up North'.

These entrepreneurs overlooked the need for an information system, so you have been called in to provide one.

Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description below. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

Note that there are many constraints contained in the description that cannot be shown on your ER diagram. State in English all such constraints.

Bookings for 'Up North' must be made and paid (by cash, cheque or credit card) in advance of their start date and are for an integral number of day's stay. Only group bookings are allowed; a group consists of one or more adults and any number of children (possibly zero), as long as there are at least two individuals in a group. The names of the individuals in a group must all be recorded for insurance purposes. It is hoped that satisfied groups will visit 'Up North' time and time again.

The theme park site contains three terraces of miners' cottages that have been converted into three hotels. Further terraces may be converted into further hotels once the remaining miners have been evicted. Each of the hotels has a number of bedrooms, and bedrooms might be twin-bedded or contain a double-bed. Thus each bedroom can cater for up to two individuals.

Bookings can be made with or without on-site hotel accommodation. In the case of those bookings that include on-site accommodation, individuals in the group are associated with rooms of one or more hotels at the time of booking. (Clearly, if a group makes another booking, then the individuals could end up in different rooms on their next visit.) The number of nights a group stays can be one less than, the same as, or one greater than the number of days it stays.

Hotels provide no meals. However, lard sandwiches and other Northern delicacies can be purchased from the many Grub Shops on site.

In a special offer, a group that is also booking on-site accommodation may purchase a number of Grub Tokens. Each token is redeemable for one meal in a Grub Shop. A group taking up this special offer can purchase as many of these tokens as it wants, but they are non-refundable. Of course, all visitors can purchase meals in the Grub Shops with cash, but Grub Tokens work out cheaper per meal.

For each member of a group, the initial booking price (whatever type of booking it is) includes a fixed number of Ride Tokens (currently 3 per person). These are redeemable for entrance to the many rides on site. Most rides 'cost' one token, but presently the Slag-'Eap-Slide 'costs' two tokens. It is conceivable that new, more exciting rides will 'cost' yet more tokens. All that can be concluded at the moment is that rides 'cost' an integral number of tokens. (Should a group run out of Ride Tokens, extra rides can be paid for in cash; each ride having a price related to the number of tokens it 'costs'.)

On any one day a ride or Grub Shop may take a certain amount of cash and a certain number of tokens in exchange for rides and meals supplied to visitors.

Large numbers of ex-miners staff the theme park. Staff form distinct workforces. There are those who work in the hotels, each connected to a particular hotel; those who work in the Grub Shops, each connected to a particular Grub Shop; those who sell Eee-By-Bubble-Gum and other novelties from the souvenir stalls; and those that operate the rides, each connected to a particular ride. The technicians form a further distinct workforce. Some technicians carry out general repair tasks on the hotels, shops, etc.; others carry out repairs to the rides, in which case they each specialise in a particular ride. Some but not all technicians do both. The final workforce is the set of entertainment staff, who dress up in whippet and pit pony costumes to entertain the kiddies.

There is an hierarchical organisation to the staff, i.e. a member of the staff has one immediate supervisor, and that supervisor has one immediate supervisor, etc.

Chapter 4

Logical Data Modelling and the Relational Data Model

Logical data modelling (logical design) is the process whereby a conceptual data model is transformed into a logical data model. The result of this is a 'first-cut' set of data structures. We refine these later (in the implementation stage) after performance analysis. If we have produced a good conceptual data model, then the process of converting it to a logical data model should be straightforward.

We are going to use the *relational data model* for expressing our logical data design. This model was developed through the 1970s and has stood the test of time. Its inventor was Ted Codd. Codd's mathematical formulation has the following properties:

- it is simple: it is based wholly on 'relations';
- it has an underlying mathematical foundation: the mathematics of relations;
- normal forms and normalisation have been defined: these specify a validation procedure for checking that the relations are 'natural' and are not susceptible to certain update anomalies;
- there are high-level manipulation operators: these allow the manipulation of sets of objects rather than manipulation of only one object at a time.
- the model gives a storage-independent view: the relations can be implemented however you like to give an efficient underlying storage structure; they do not, contrary to some opinion, have to be implemented as serial files.

This data model has proved itself so much that there are now many commercial database management systems based on it (so-called relational database management systems). Even those systems that are not relational often come with relational features these days. However, do not be misled. Two points need to be made. (1) The relational data model is a nice, clean model for the logical design stage. While it may then be an obvious step to implement it using a relational database management system, it is just as suitable as a starting point for implementation using some other type of database system or for a traditional file-based system. (2) Relational database management systems often fail to accurately capture the full power of Codd's relational data model (because industry always goes for the quick buck).

We begin with an informal presentation of the model's building blocks, and then consider how to map from the conceptual data model to the logical data model.

4.1 Domains

The smallest unit of data in the relational model is an individual data value. These data values are taken to be *atomic* (i.e. cannot be broken into smaller units).

A *domain* is a set of values, usually of the same type. Ideally, they should be any user-definable set of atomic values, e.g. the domain of student identifiers, the domain of course codes, etc.

The significance of domains is that if two attributes (see later) have the same domain, then operations involving those attributes (e.g. comparison between them) probably make sense.

4.2 Relations

Mathematically, a relation R defined on domains $D_1, \ldots D_n$ is a subset of the Cartesian product, $D_1 \times \ldots \times D_n$. But, in the relational model, things are more complicated because a relation has an intension (header) as well as an extension (body), and the extension is allowed to vary over time.

Here's an example of the STAFF relation:

STAFF		
Staff_no:	Name	Department_code
115-a	R.Hall	MA
137-a	D.Bridge	CS
064-a	N.Allinson	EL
532-b	A.Burns	CS

This is made up of two parts. There is the *intension*:

STAFF		
Staff_no:	Name	Department_code

And there is the *extension* (at this moment in time):

115-a	R.Hall	MA
137-a	D.Bridge	CS
064-a	N.Allinson	EL
532-b	A.Burns	CS

4.2.1 Intensions

Informally, the intension is the 'header' of the relation. It is the permanent part of the relation. It consists of:

- a name (e.g. STUDENTS);
- a fixed set of attributes (e.g. Student_no, Name, Department_code, Date_of_birth, Start_date);
- a set of constraints between attributes.

A relation intension is written:

> STUDENTS(Student_no, Name, Department_code, Date_of_birth, Start_date)

Attributes are simply names (column headers):

- each is associated with a domain, e.g. Student_no with the domain of student numbers, Name with strings of up to 40 characters;

- the domains need not be distinct, e.g. Date_of_birth and Start_date are both from the domain of dates;
- an attribute's values will be drawn from the attribute's domain.

The notation to uniquely pick out an attribute is, e.g.:

STUDENTS.Name

The number of attributes in a relation is called the *degree*, e.g. STUDENTS has degree 5, and STAFF has degree 3.

4.2.2 Extensions

Informally, the *extension* is the body of the relation.

If there are n attributes, A_1, $A_2, \ldots$, A_n associated with domains D_1, $D_2, \ldots$, D_n respectively (D_1, $D_2, \ldots$, D_n not necessarily distinct), then the body will at any time be a subset of the Cartesian product of the domains $D_1 \times D_2 \times \ldots \times D_n$.

Alternatively, the body is a set of *tuples*, $<d_1, d_2, \ldots, d_n>$ such that $d_1 \in D_1$, $d_2 \in D_2, \ldots$, $d_n \in D_n$, e.g. <825-e, J.Smith, PS, 15/02/55, 17/05/82>

The number of tuples in a relation at a time is the *cardinality* of the relation. For example, STAFF *currently* has cardinality 4.

NOTE — THE EXTENSION IS TIME-VARYING: the actual subset of the Cartesian product (the actual set of tuples) constituting the body is not fixed. Updates change the extension. (Mathematical relations do not vary over time. An 'update' on R would produce a new relation R′. In the relational model, we assume that an update on R produces a new extension for R.)

4.2.3 Some properties of relations

- The extension is a *set* of tuples. Since it is a set: (a) there will be no duplicate tuples, and (b) tuples are unordered in the set. (Unfortunately, writing a relation down gives an ordering. Remember that this ordering is not significant to the model.)
- The intension has a *set* of attributes. Since it is a set: (a) there will be no duplicate attributes within a relation (i.e. attribute names will be unique within a relation, but their domains need not be distinct, and two relations might have an attribute name in common hence the

notation STUDENTS.Name, LECTURERS.Name, etc.), and (b) the attributes should not be taken to have a significant order (strictly you should refer to "the Name attribute of STUDENTS", not "the second attribute of STUDENTS").

- Domains are sets of atomic values: a value cannot be a repeating group of values (or, if it is, it will be treated as an indivisible unit).

4.3 Null values

There is some debate about *null values*. This is what supporters of null values might say:

- each domain is allowed to have a special value, a null value;
- strictly this is not the same as spaces or zero (although some commercial systems use spaces as a null value);
- a null value in a tuple will be a 'place-holder' for some actual, non-null value;
- null values are needed because we only ever get incomplete, partial data about the real world.

However, the problem with nulls is that we can have different types of null values with different meanings. People have not always drawn these distinctions. Here's an example with three nulls but each might have a different meaning:

STUDENTS		
Student_no	Name	Tel_no
a1	E.White	null
a2	E.Brown	null
a3	J.Black	null

Value unknown: we know E.White has a telephone but we don't know what the number is;

Value inapplicable: we know E.Brown does *not* have a telephone;

Status unknown: we do not know whether J.Black has a telephone or not.

Nulls are not really needed for the second example (value inapplicable). A different set of relations can handle the data without resorting to nulls:

STUDENTS(Student_no, Name)
STUDENTS_ON_PHONE(Student_no, Tel_no)

Then STUDENTS_NOT_ON_PHONE can be computed as the difference between STUDENTS and STUDENTS_ON_PHONE.

The third case (status unknown) is really beyond the scope of conventional systems. This is because the conventional interpretation of the result of an interrogation of a computer system uses the *closed world assumption*, i.e. roughly, the user assumes that the system tells us *all* truths about the world, i.e. it contains all the true facts: if we cannot find a fact in the system, then it must be false. Since this interpretation is so ingrained, we need to move to deductive databases to remove it.

Returning now to the first use of nulls (value unknown). The supposed uses for these nulls are:

- when inserting a tuple if you don't know a value;
- when adding a new attribute to an existing relation, fill it in with nulls;
- for empty answers to queries, e.g. Get all students whose grade > 85%. (But what's wrong with an empty relation for this?)

The argument against these nulls is that they look and behave just like a value, but they aren't ordinary values; they're variables. This causes implementors and users problems, e.g.:

- comparison: is null = null?
- arithmetic: is null - null = 0?
- counting: how many distinct telephone numbers are there in this relation: 2 or 3?

STUDENTS	
Name	Tel_no
E.White	null
E.Brown	2722
J.Black	2756

4.4 Keys

Keys in the relational model are not to be confused with identification keys in the conceptual data model or with access keys/indexes in the internal level of database management systems (see Chapter 9), although they are related to both.

A key of a relation will be one or more of the relation's attributes. The key attributes' values will (in combination) uniquely identify a tuple of the relation.

We know that every relation will have a key because the extension is a *set* of tuples so there are no duplicates. In the degenerate case, *all* attributes will form the key.

YOU CANNOT DETERMINE THE KEY OF A RELATION BY LOOKING AT ITS EXTENSION AT ONE MOMENT IN TIME. From the extension of a relation you may be able to tell what *cannot* be the key and what *might* be the key. But it is really for the client (the universe of discourse expert) to say what the key actually is. This is because key is an intensional concept; it is a user-defined *constraint* on possible extensions (for all time).

It is usual to insist that keys are minimal, e.g. suppose that the client says that given the values of a Staff_no and a Department_code in combination we would be able to uniquely identify a tuple of a STAFF relation. We would want to check with the client that this is a minimal combination: if Staff_no on its own identifies tuples then that would be the key.

There may be more than one minimal combination of attributes that identifies a tuple, e.g. for Computer Science lecturers at York, Login_id, Office_no, Telephone_no, NHI_no. Each is referred to as a *candidate key*.

A more formal summary:

> Let R be a relation with attributes A_1, A_2,...,A_n. The set of attributes K, where K = (A_i, A_j,...,A_k) of R is said to be a *candidate key of R* if and only if it has the following properties:
> (1) uniqueness: At all times, no two distinct tuples of R have the same values for all attributes in K.
> (2) minimality: None of the attributes of K can be discarded without destroying the uniqueness property.

From the candidate keys, we choose a *primary key* (often referred to simply as the key). (The remaining candidate keys are sometimes referred to as *alternate keys*.) Attributes not part of the primary key we will call *non-key attributes*. Non-key attributes which are not part of a foreign key (see later) are *own attributes*.

The notation for primary keys is to put them first and preceding a colon. (Use of italics, underlining or bold also helps to highlight them.)

STAFF(**Staff_no:** Name, Department_code)

(We have no special notation for showing the other candidate keys, the alternate keys. If they need to be recorded they are noted in a margin.)

Primary keys, being constraints, enable us to define an important rule for controlling the *integrity* of a relational data model.

Integrity rules for primary keys

1. Values of the primary key attributes of a relation must be unique within the relation.

2. Entity integrity: No attribute participating in the primary key of a relation is allowed to accept null values.

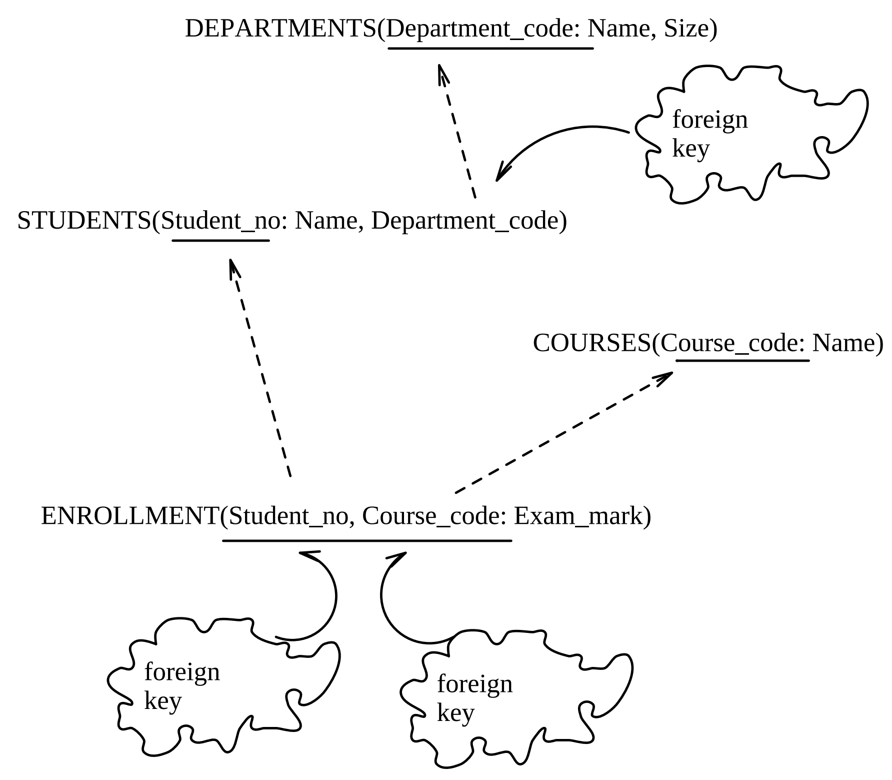
The first of these is obvious; the second may be less so and so is explained further:

Informally, a tuple is going to represent some real-world entity (e.g. a student, course, etc.). Therefore, while we might allow the use of null values in non-key attributes (to signify that a person's address, say, is unknown), we cannot allow null values in the primary key of a relation. If we did, we would have no way of identifying that entity as distinct from all others, e.g. we could not tell apart two students both called "John Smith", both belonging to Computer Science, and both having unknown student identification numbers.

4.5 Foreign Keys

Foreign keys define cross-references between relations. A foreign key is an attribute (or set of attributes) (either key or non-key) of a relation which is required to match the primary key of another relation. Foreign keys are also defined by the client since they too specify constraints.

In this course, we use the convention that if two attributes of different relations have the same name and one is a primary key, then the other is a foreign key. However, in cases where this could be misleading, the foreign keys and the primary keys they correspond to must be explicitly stated, e.g. in English in a margin somewhere.



In the above, Name is not a foreign key since it is not a primary key anywhere (and, intuitively, it sets up no cross-references anyway).

Foreign keys enable us to define a second important rule for controlling the integrity of a relational data model.

Referential Integrity

> If relation R2 includes a foreign key F which coreferences the primary key P of a relation R1, then every value of F in R2 must either:
> (a) be equal to the value of P in some tuple of R1; or
> (b) be wholly null (i.e. each attribute of F in this tuple of R2 must be null).

Informally, if one entity is related to another (i.e. if one tuple is to cross-reference another), then that entity must exist (i.e. that foreign key value must appear in the other relation). For example, an enrollment entity must refer to a student entity and a course entity that we know about. (The part of the definition that allows the foreign key to be null enables an optional relationship. Obviously this only applies if the foreign key in R2 is not part of R2's primary key, otherwise entity integrity would be violated.)

Examples:

COLLEGES(**College_name:** Size, No_of_TV_rooms)
STAFF(**Staff_no:** Name, College_name)
STUDENTS(**Student_no:** Name, College_name)
COURSES(**Course_code:** Name)
ENROLLMENT(**Student_no, Course_code:** Exam_mark)

- ENROLLMENT.Student_no is a foreign key that is referentially dependent on STUDENTS.Student_no. Clearly, the foreign key values must all appear in the STUDENTS relation, and they cannot be null because they are part of the key of ENROLLMENT.
- STAFF.College_name and STUDENTS.College_name are both referentially dependent on COLLEGES.College_name. Their values must appear in COLLEGES, but it is reasonable for them to be null, i.e. for someone not to be a member of a college[1]. In fact, at York we would allow STAFF.College_name to be null but not STUDENTS.College_name. This requires us to write in the margin that STUDENTS.College_name cannot be null (on the assumption that the default is that it can be).

In the relational model, if one tries to insert, e.g., an enrollment for a student but for a course code which does not appear in COURSES, the constraint is violated and the insertion is disallowed. (Commercial systems might offer the opportunity to propagate the insertion.)

We have now covered all there is to the structural aspects of the relational data model. It is easily possible to show that this model can capture all sorts of real world structures, including trees and graphs.

4.6 Logical Design

You will recall that logical design is the process of mapping a conceptual data model, which in our case is an ERA diagram, to a logical data model, which in our case is a set of relations, still uninfluenced by considerations of the final implementation such as efficiency.

Here are *guidelines* for producing a logical data model.

1. Create first structures

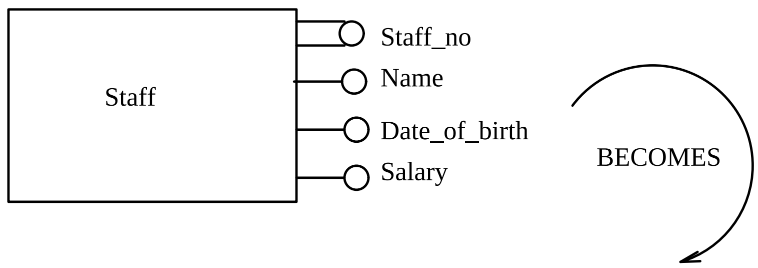
For each entity type in the conceptual data model, define a relation in which:

- the relation name is the same as the entity type name;

[1] Note how this is the second use of nulls, value inapplicable.

- properties of the entity type become the attributes of the relation;
- value sets of the properties become domains of the attributes.

For example:



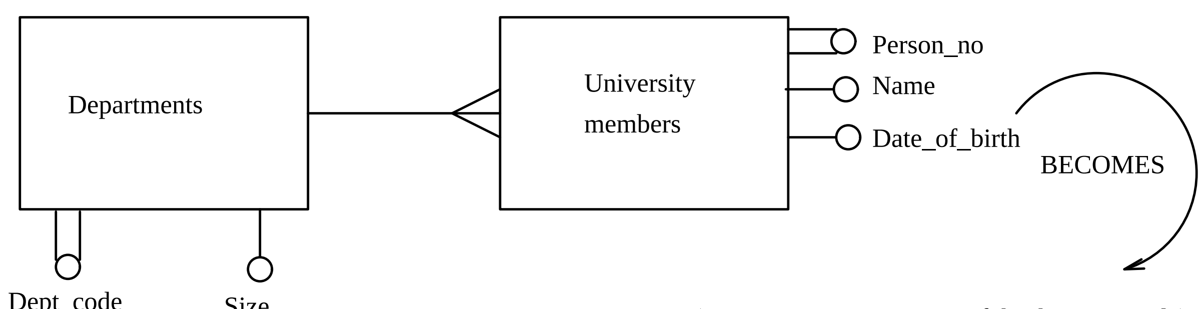
STAFF(Staff_no, Name, Date_of_birth, Salary)

Domains

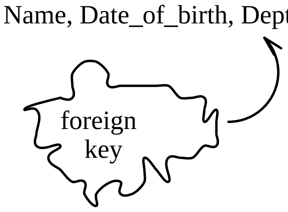
Staff_no:	Staff_nos
Name:	String(1..40)
Date_of_birth:	Dates
Salary:	Integer 0..Pittance

2. Establish foreign keys

For each one-to-many relationship, e.g. one A has many Bs, include the primary key of A in B as a foreign key. For example:



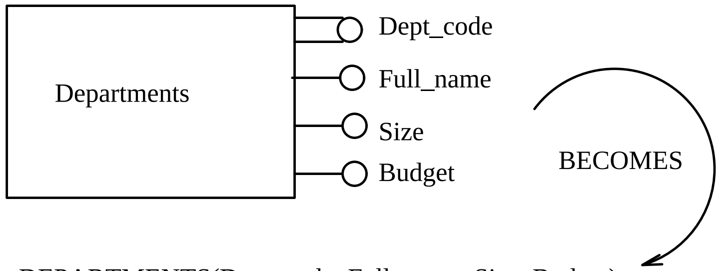
UNIVERSITY_MEMBERS(Person_no: Name, Date_of_birth, Dept_code)



3. Establish primary keys

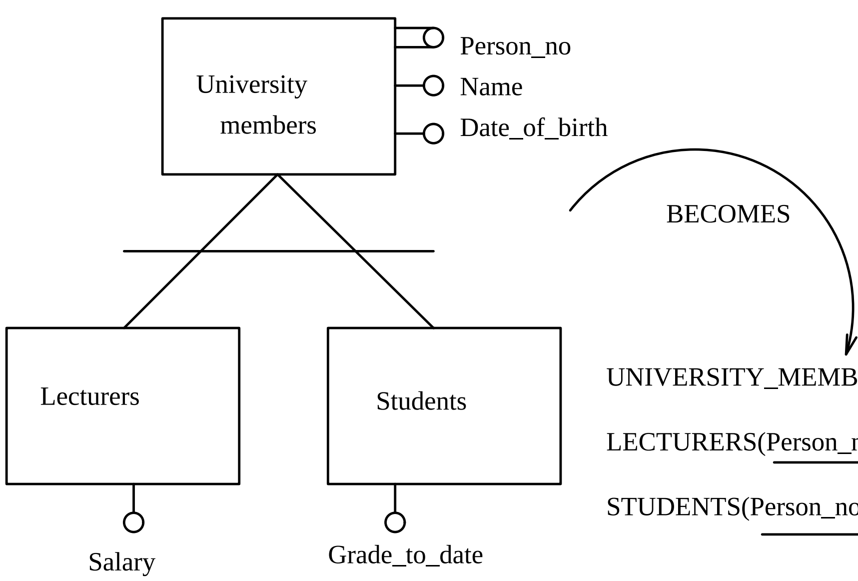
There are three cases to consider.

- the original entity type is not identification dependent on other entity types, nor is it a subtype: in this case, trivially, identification key properties become the primary key attributes:



DEPARTMENTS(Dept_code: Full_name, Size, Budget)

- the original entity type was a subtype of another: in this case, the primary key attributes are the same as the primary key attributes of the supertype (note that step 4 of the methodology would also apply to this structure):

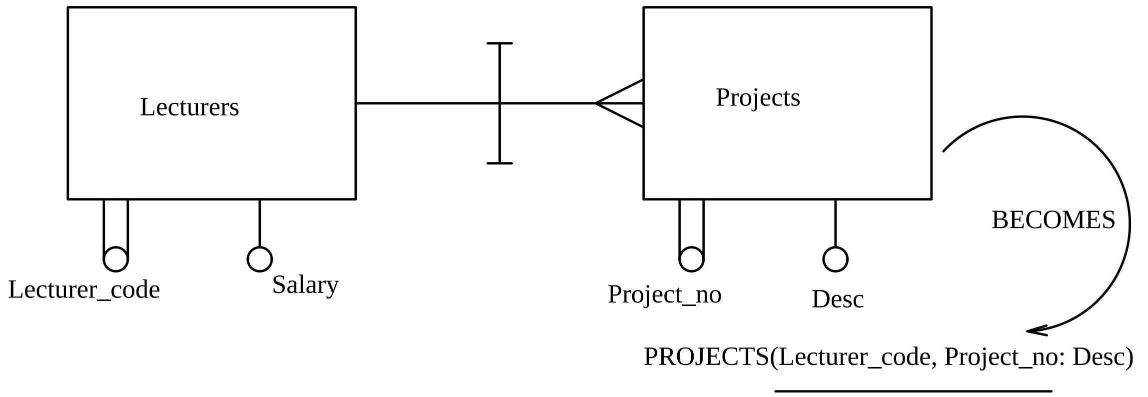


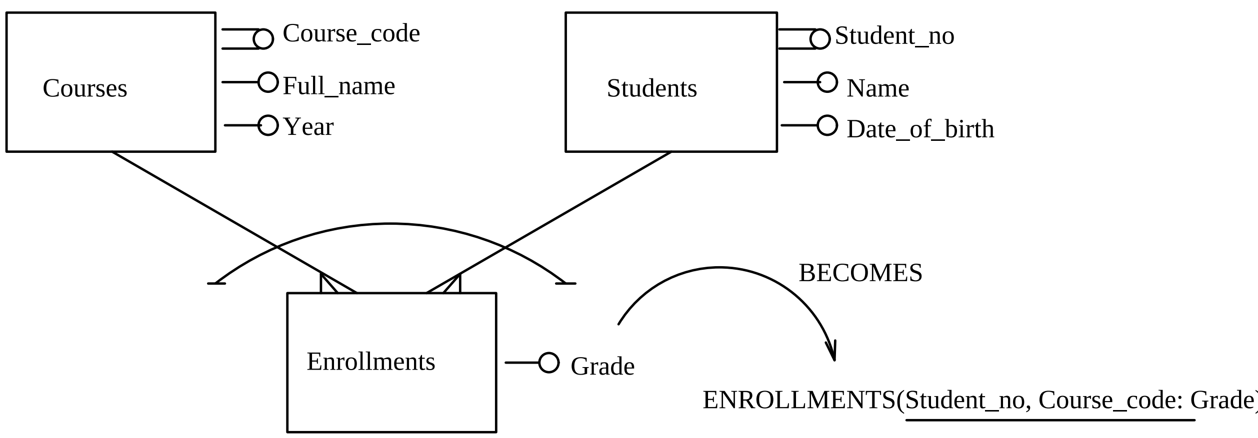
UNIVERSITY_MEMBERS(Person_no: Name, Date_of_birth)

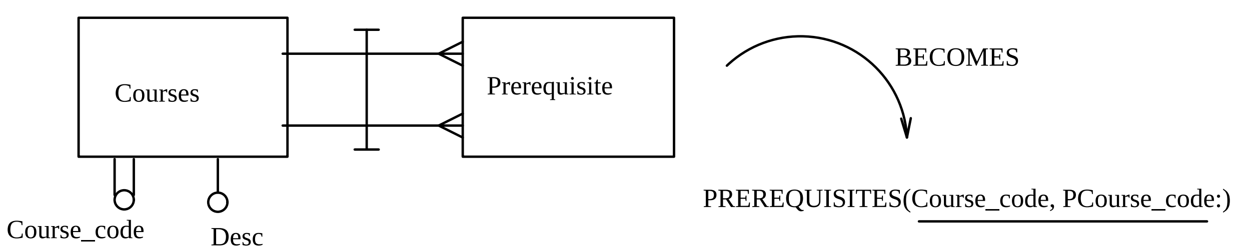
LECTURERS(Person_no: Salary)

STUDENTS(Person_no: Grade_to_date)

- the original entity type was identification dependent on one or more other entity types: in this case the primary key attributes are any identification key properties plus all identification properties of entity types it depends on:





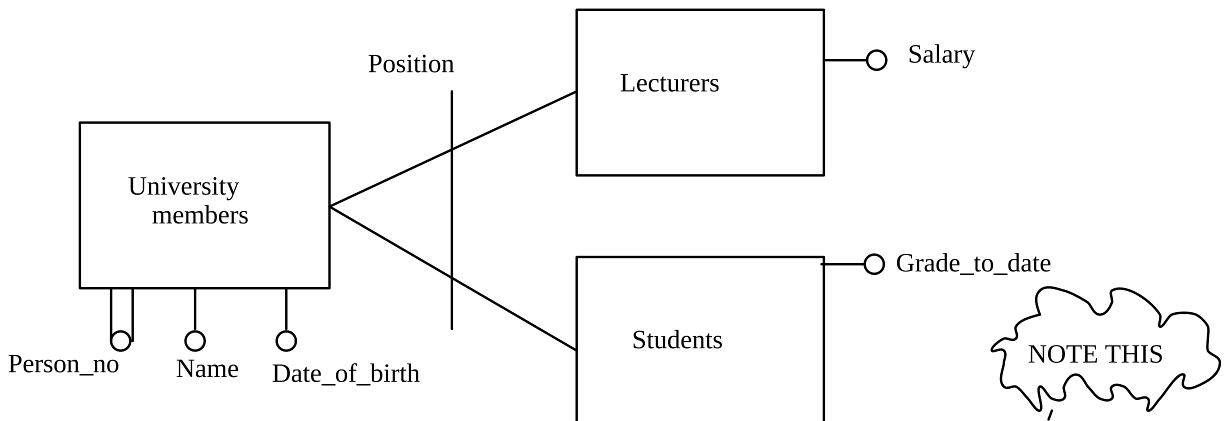


[Note: the observant amongst you will have noticed that step 2 seems to assume that step 3 has already been done, and step 3 seems to assume that step 2 has already been done. In practice, by working 'in' from the 'extremities' of the diagram, carrying out the two steps in the order given here as you go, will not cause a problem.]

4. Establish classification attributes for subtypes

For a subtype-supertype relationship, we need an attribute which says which subtype a supertype belongs to. The solution differs according to whether the subtypes are exclusive or not.

- if they are exclusive, add another non-key attribute to the supertype (corresponding to the classification property):

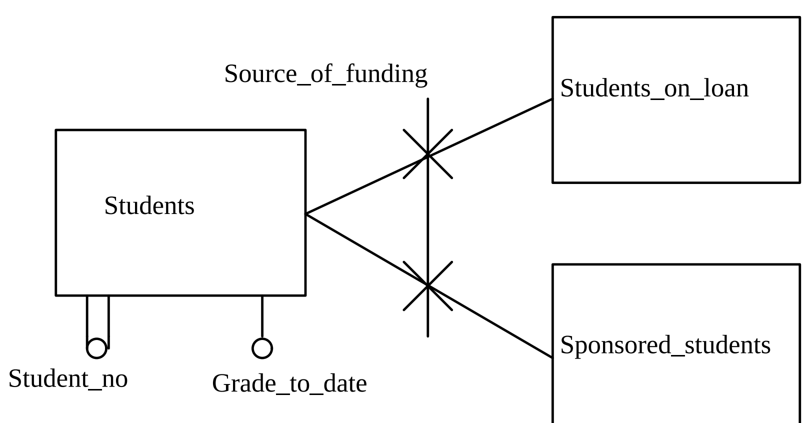


BECOMES

UNIVERSITY_MEMBERS(Person_no: Name, Date_of_birth, Position)

Position has domain {Lecturers, Students} telling you where to look for more information.

- if not exclusive, define a new all key relation:



BECOMES

STUDENTS(Student_no: Grade_to_date)

SOURCE_OF_STUDENTS_FUNDS(Student_no, Source_of_funding:)

Domain of Source_of_funding is {Loan, Sponsor}

5. Refinement

Check for any redundant relations, e.g. an all-key relation *might* be redundant (but be careful: all-key relations often express important relationships —consider the relation PREREQUISITES).

6. Verify

Check, e.g., relation names are all different, there are no duplicate attribute names within a relation, all relations have a primary key.

7. Validate

See below.

8. Iterate

... until 'sign off'.

4.7 Validation of a logical data model

The existence of a good validation method is one of the factors which makes the relational data model so attractive as a logical data model.

If we have produced a sensible conceptual data model, and followed the transformation rules, then the relations we have in the logical data model should be natural and not contain redundancies that might otherwise cause update anomalies.

To check that this is so, we can see whether the relations are in one of a number of *normal forms*. A relation is said to be in a particular normal form if it satisfies a certain specified set of rules. If a relation does *not* conform to a normal form, then it should be amended so that it does. However, this usually implies that there was a problem with the conceptual data model or with the way you carried out the mapping between the two models. So, rather than just altering the relational data model, go back and check!

4.7.1 Functional Dependencies

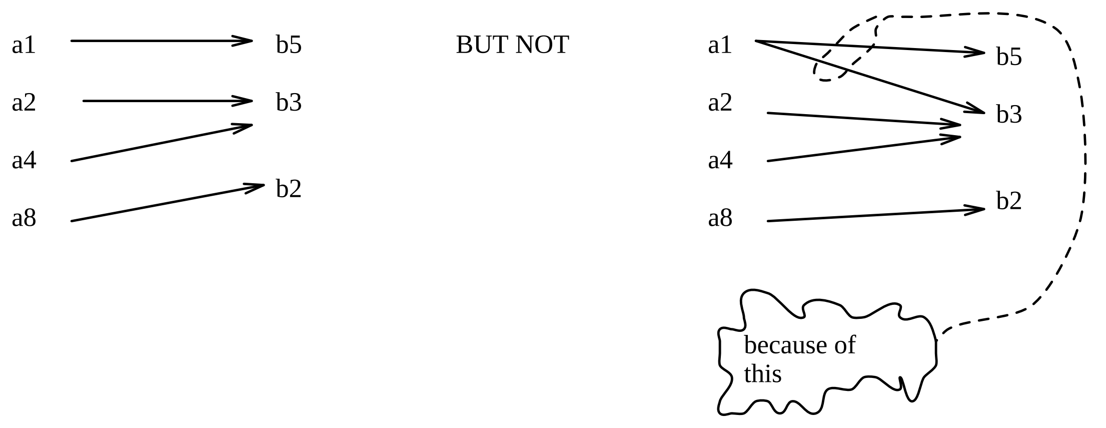
To define the normal forms, we need to define the notion of *functional dependency*.

> For a relation R, one attribute, B, is *functionally dependent* on another attribute, A, if, at each point in time, knowing the value of A uniquely determines the value of B.

(Each value of A will have only one value of B associated with it, at any one time.) Our notation for this will be:

$R.A \rightarrow R.B$

i.e.



Note that the definition includes the phrase "at each point in time" — functional dependencies are not determined by inspecting a relation's contents. They are statements of constraints.

Some examples of fds:

- Person_no → Name
  (This does not rule out two people having the same name, but it does rule out two names being associated with the same person, i.e. it is m:1.)
- Person_no ↔ NHI_no
  (i.e. Person_no → NHI_no and NHI_no → Person_no, which gives a 1:1 link.)
- <Course_code, Student_no>→ Exam_mark
- <Lecturer_code, Project_no>→ Proj_desc

You might notice that there are fds between the keys of the relations we have defined up to now and their non-key attributes. This is no accident. It shows we did get our analysis right: we want non-key attributes to be functionally dependent on their key attributes. Normalisation is the process that helps us to check that we do have this desirable property.

Normalisation uses fds to check that:

> "non-key attributes provide a fact about the key, the whole key and nothing but the key."

We check that our relations conform to first, second and third normal forms. There are other normal forms (Boyce/Codd, 4th, 5th, project/join, domain/key) but these are beyond the scope of this course and rarely arise in the real world.

We will be able to show that if a relation is not in a certain normal form, then there will be problems in updating the relation.

4.8 First Normal Form

> A relation is in *first normal form* (1NF) iff all of its domains contain atomic values only.

In other words, there should be no 'repeating groups'.

The following is *not* in 1NF:

STUDENTS(**Student_no:** Name, Course_codes)

STUDENTS		
Student_no	Name	Course_codes
123-a	E.White	{dad, fss, ops}
234-a	J.Black	{dad, fun, ops}
452-b	M.Brown	{fun}

The Course_codes attribute contains the set or repeating group of courses the student takes. This is a bad structure: the relational model insists that we treat data values as atomic, hence with this structure we cannot access the individual courses taken.

Avoiding this is a constraint we saw that has been placed on the relational model itself. Other normal forms are not enshrined in such constraints: they are merely good design guides.

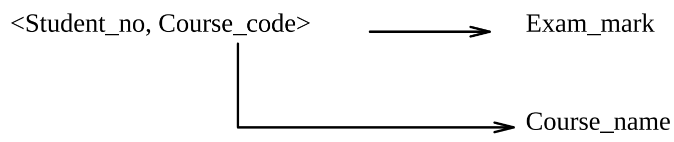
4.9 Second Normal Form

> A relation is in *second normal form* iff it is in 1NF and every non-key attribute is fully dependent on the primary key.

In other words, non-key attributes are dependent on the *whole* of the primary key.

The following is not in 2NF:

ENROLLMENTS(**Student_no, Course_code:** Exam_mark, Course_name)



...because Course_name depends only on part of the primary key.

If the relation has the following extension:

ENROLLMENTS			
Student_no	**Course_code:**	Exam_mark	Course_name
234-a	dad	60	Data Analysis and Design
235-a	dad	70	Data Analysis and Design
458-b	fun	55	Functional Programming
458-b	dad	75	Data Analysis and Design
399-a	ops	0	Operating Systems
456-a	fun	55	Functional Programming

...then we can illustrate the following update anomalies:

- if this is the only place where we record which Course_code has which Course_name, then
  - we cannot insert the new course, whose code is 'ist' and whose name is 'Intelligent Systems', unless someone is taking it;
  - if we delete student 399-a, we lose the fact that course code 'ops' has course name Operating Systems;
  - if we want to modify the course name of 'dad' to 'Death and Destruction', we need to change 3 tuples.
- this is therefore clearly a bad place to record which Course_code has which Course_name.

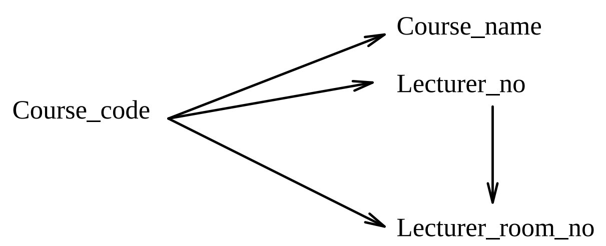
4.10 Third Normal Form

A relation is in *third normal form* (3NF) iff it is in 2NF and every non-key attribute is non-transitively dependent on the primary key.

In other words, a non-key attribute should be directly dependent on the key and only the key, not dependent on the key indirectly through some other dependency.

The following relation is not in 3NF:

COURSES(**Course_code:** Course_name, Lecturer_code, Lecturer_room_no)



...because Lecturer_room_nos are determined by Lecturer_code and therefore only indirectly (transitively) determined by Course_no.

If the relation has the following extension:

COURSES			
Course_code:	Course_name	Lecturer_code	Lecturer_room_no
dad	Data Analysis and Design	dgb	V/X034
mom	Models of Machines	ajf	X/D107d
log	Logic programming	cgm	V/X036
flt	Formal Language Theory	ajf	X/D107d

...then we can illustrate the following update anomalies:

- if this is the only place where we record which lecturer has which room, then
  - we cannot insert a new lecturer's room no. (e.g. Marvin Minsky, with Lecturer_code min, and room no. X/E007) until we know what he teaches;
  - if we delete 'log' from the curriculum, we lose the fact that V/X036 is cgm's office;
  - if we want to modify the office of ajf by moving him to X/D108, we need to change 2 tuples.

- this is therefore clearly a bad place to record which lecturer has which room.

An alternative formulation of the 3NF definition is:

> A relation R is in third normal form (3NF) iff, for all time, each tuple of R consists of non-key attributes which are mutually independent and functionally dependent on the whole of the primary key.

So, the validation stage of logical design should include a check that each of the relations is in 3NF. 3NF is a form which is free of certain update anomalies and also seems to capture 'entityhood'.

If any of your relations is not in 3NF then something has gone wrong in conceptual data modelling or in logical data modelling: locate this problem, correct it and propagate through any changes.

4.11 An Alternative Methodology

In days gone by, before the days of entity-relationship diagrams and conceptual data models in general, logical data model design was done using *normalisation*.

To use normalisation as a logical data model design technique, there is no need to draw up a conceptual data model first. Simply do the following:

- draw up some rough relations which hold all the data you wish to represent;
- if the relations are not in a particular normal form, decompose them into smaller relations which do satisfy that form.

Here's an example. Suppose I find the following report in the Computer Science Office:

*** STUDENTS LIST, 10/10/94 *** PAGE 1 OF 10

Student_no	Name	Course_code	Course_name	Lecturer_code	Lecturer_room_no	Grade
123-a	E.White	dad	Database Analysis & Design	dgb	V/X034	20%
		fun	Functional Programming	cr	X/D114c	40%
		ops	Operating Systems	adne	X/E009	25%
345-b	J.Black	fun	Functional Programming	cr	X/D114c	60%

This seems to show all the data I anticipate storing in my new system, so I will begin with the following relation as a rough-cut logical model:

> STUDENTS(Student_no, Name, Course_code, Course_name, Lecturer_code, Lecturer_room_no, Grade)

The attributes seem to be about students and Student_no identifies students, so we can indicate the key:

> STUDENTS(**Student_no:** Name, Course_code, Course_name, Lecturer_code, Lecturer_room_no, Grade)

But, this relation has a repeating group (Course_code and other attributes describing the courses a student takes), so it is not in 1NF. We will decompose it into two relations:

> STUDENTS(**Student_no:** Name)
> ENROLLMENTS(**Student_no, Course_code:** Course_name, Lecturer_code, Lecturer_room_no, Grade)

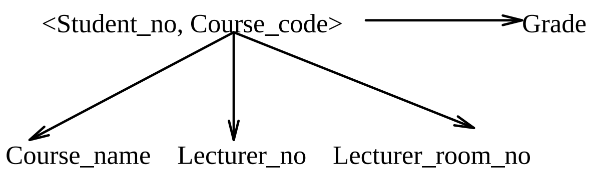
(Note that Student_no is also put into the new relation as a foreign key to retain the information about which courses a student takes.)

Now I look at whether attributes are dependent on the whole key. For STUDENTS:

Student_no ⟶ Name

This is OK.

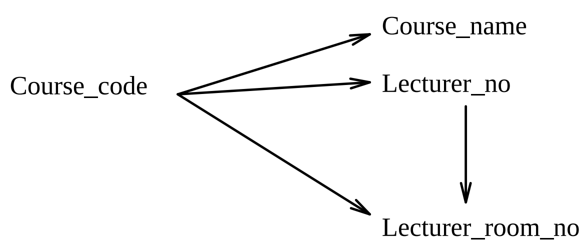
For ENROLLMENTS:



This is not in 2NF. We decompose ENROLLMENTS as follows:

ENROLLMENTS(**Student_no, Course_code:** Grade)
COURSES(**Course_code:** Course_name, Lecturer_code, Lecturer_room_no)

Now I check that attributes are non-transitively dependent on the key. For COURSES:



This is not in 3NF. We decompose COURSES into two relations as follows:

COURSES(**Course_code:** Course_name, Lecturer_code)
LECTURERS(**Lecturer_code:** Lecturer_room_no)

This completes the normalisation: all relations are now in 3NF. The final relations are:

STUDENTS(**Student_no:** Name)
ENROLLMENTS(**Student_no, Course_code:** Grade)
COURSES(**Course_code:** Course_name, Lecturer_code)
LECTURERS(**Lecturer_code:** Lecturer_room_no)

This logical data modelling technique (i.e. normalisation) was originally thought to have great promise: surely given a set of initial relations and a set of fds a program could do the decomposition for us?

This may be true, but (a) getting the initial relations is difficult (particularly in making sure we leave nothing out), and (b) getting the fds is difficult.

Thus the more top-down approach, i.e. conceptual modelling using entity relationship diagrams, seems to have won the day. Normalisation is now just a validation technique within this top-down process.

4.12 Exercises

A) EXAM QUESTIONS: Several years' worth

For those of you who submitted answers to Chapter 3's conceptual modelling questions: using the model answer conceptual data models, produce *logical data models* in the form of collections of normalised relations.

This can be done for the airlines question (1989, worth 8 marks), the video hire shop (1990, worth 10 marks, but to get the full 10 marks, you were also asked to list the *referential integrity constraints* that would need to be maintained, so do this too now), the Help Desk question (1992, worth 8 marks), the buses question (never set as an exam question), and the nappies question (1993, worth 8 marks). Don't bother to do it for the 'Up North' Theme Park exercise.

B) EXAM QUESTION: Question 9, Part (iii) only, 1990

The Queen wants a simple database to keep track of the works of art she has been given on her travels, and to show to which museums she has loaned these items.

The following relation definitions are her first attempt to model the situation:

ITEMS(**Item_no:** Description, Country, Continent)
LOANS(**Item_no, Museum_id:** Museum_name, Museum_address, Date_loaned)

(a) By giving examples, explain to Her Majesty some of the problems of this representation. (5 marks)

(b) Apply the technique of *normalisation* to the above set of relations to produce a set of relations which conform to third normal form. In your answer, include brief explanations of each step of the normalisation. (5 marks)

C) EXAM QUESTION: Question 10, 1991

Aubrey Boomley-Smythe is manager of the Grand Theatre, Sudbury. Over a cup of coffee and a French Fancy, Aubrey describes to you a small part of his business as follows:

The auditorium is split into a number of "parts of the house", e.g. the Stalls, the Dress Circle, etc. Each part of the house contains a number of seats. Each seat has a unique number.

The price of a ticket for a seat depends only on which part of the house the seat is in. So, for example, all seats in the Stalls are one price, all seats in the Dress Circle are another price, and so on.

Obviously, on any particular date, many of the seats may be sold and a seat may be sold on any number of dates. When a seat is sold on a particular day, Aubrey likes to know whether the ticket was bought using cash, credit card or cheque.

(i) [7 marks] Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the above description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

(ii) [4 marks] Using the results of (i), produce a *logical data model* in the form of a collection of third normal form relations.

(iii) [6 marks] Rather sheepishly, Aubrey shows you the design he had originally produced for this database, comprising a single relation:

SALES(**Date:** Seat_no, Part_of_house, Price, Method_of_payment)

Apply the technique of *normalisation* to this relation to produce a set of relations which conform to third normal form. In your answer, include brief explanations of each step of the normalisation.

(iv) [3 marks] With reference to your answers to parts (i), (ii) and (iii), discuss the role of normalisation in database analysis and design.

D) NOT AN EXAM QUESTION: Based on Open University material

Here's an extract from a report which needs to be produced at your local doctors' surgery:

Pat No.	Pat Name	GP No.	GP Name	Appoint. Date	Cons No.	Cons Name	Consultant Address	Sample required
01027	Moon	919	McGrope	3.9.89	315	Fleece	15 Acacia Ave	Blood
				20.12.89	315	Fleece	15 Acacia Ave	None
				1.2.90	420	Cozenage	9 Beech Rd	Urine
08023	Tewes	818	Fumble	3.9.89	315	Fleece	15 Acacia Ave	Semen
				3.9.89	437	von Ripov	31 Fir St	Urine
				3.9.89	420	Cozenage	9 Beech Rd	Faeces
191146	Woden	717	Grabble	4.10.89	437	von Ripov	31 Fir St	Blood
001239	Thor	818	Fumble	5.6.89	437	von Ripov	31 Fir St	Sputum
007249	Frigg	717	Grabble	9.11.89	420	Cozenage	9 Beech Rd	None
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.

It shows which patients (Pats) of which GPs had to be referred to which consultants (Cons).

Use normalisation to produce a database that could be used to produce this report.

E) EXAM QUESTION: Question 7, 1993

"Victorian Values" is a political pressure group that embraces causes such as the banning of BBC2 and the return of the death penalty.

It has a simple file-based membership system, which runs on a PC and is used by Colonel Alfred Duffer. The system consists of just one file of the following format (shown with some sample data):

```
Member  Name          House       Town    MP's Member-  Campaign  Campaign Donation
 No.                                     Party ship Paid  No.      Name

------------------------------------------------------------------------------------------
 1  Brian Brisket 15 Acacia Ave  Sudbury  Con.   Yes      1    Anti-porn   $12
                                                          4   Pro-hanging  $25
                                                          4   Pro-hanging  $15
 2  Jane Topside  72 Fir Lane    Ilford   Lib.    No
 3   Arnold Rump   6 Ash Street  Sudbury  Con.   Yes      2    Anti-perv  $500
```

(Take it to be the case that towns have unique names and that each town has only one MP, but an MP can serve many towns.)

A simple Pascal program is run once a year to request payment of membership fees. During the year, Alfred uses an editor to keep the file up to date (e.g. to change addresses, insert donations, etc.).

(i) [10 marks]

(a) Using the example data, explain to Alfred some of the problems of this representation. [3 marks]

(b) Apply the technique of *normalisation* to the file structure to produce a set of relations which conform to third normal form. In your answer, include brief explanations of each step of the normalisation. [7 marks]

(ii) [10 marks] Mrs Betty Simpson runs the campaigning system. This system runs on a separate PC. Therefore, every 12 months, Betty takes a copy of Alfred's file, which she then uses to form a mailing list for campaign letters for the next 12 months of campaigning.

The program that prints these letters was written in BASIC by Betty's son. It requires its data to be in a diffrent format from the format used in Alfred's file. Specifically, it expects each record to consist of the following fields in the given order:

```
Member No.   Surname   Title   Initials   House   Town
```

It also expects data fields to be delimited by colons. Here is some example data:

```
1:Brisket:Mr:B:15 Acacia Avenue:Sudbury
2:Topside:Ms:J:72 Fir Lane:Ilford
3:Rump:Mr:A:6 Ash Street:Sudbury
```

Betty converts Alfred's file into a file of this format manually using an editor.

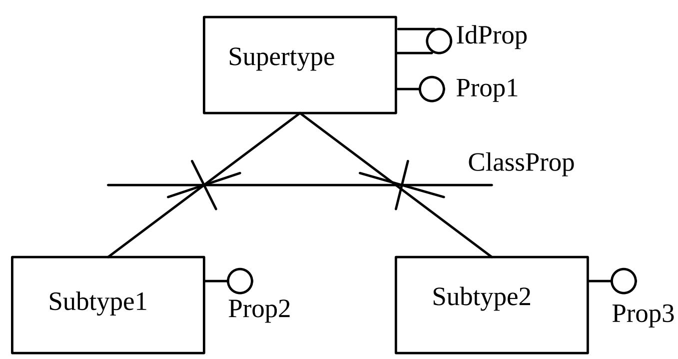
During the year, Alfred occasionally passes hand-written notes to Betty mentioning people who have died, changed address, etc. so that Betty can update her file.

Describe *in detail* the problems that can arise using the membership and campaign systems given above.

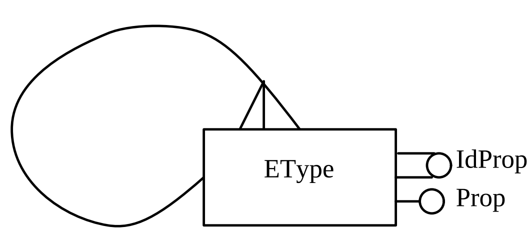
F) EXAM QUESTION: Question 3, 1994

(i) [6 marks] Produce logical data models in the form of collections of normalised relations for the following three ER diagrams. Give *brief* justifications of your answers.

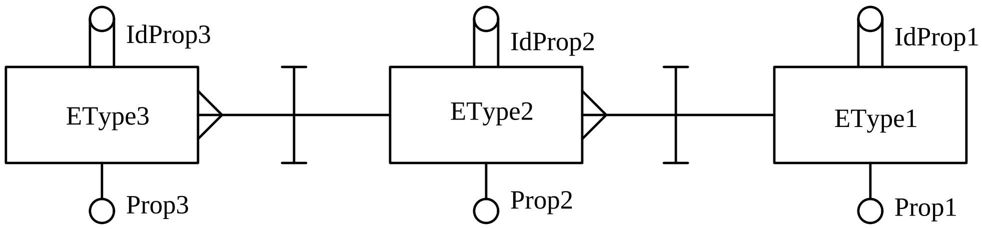
(a) Non-exclusive subtypes



(b) A unary relationship

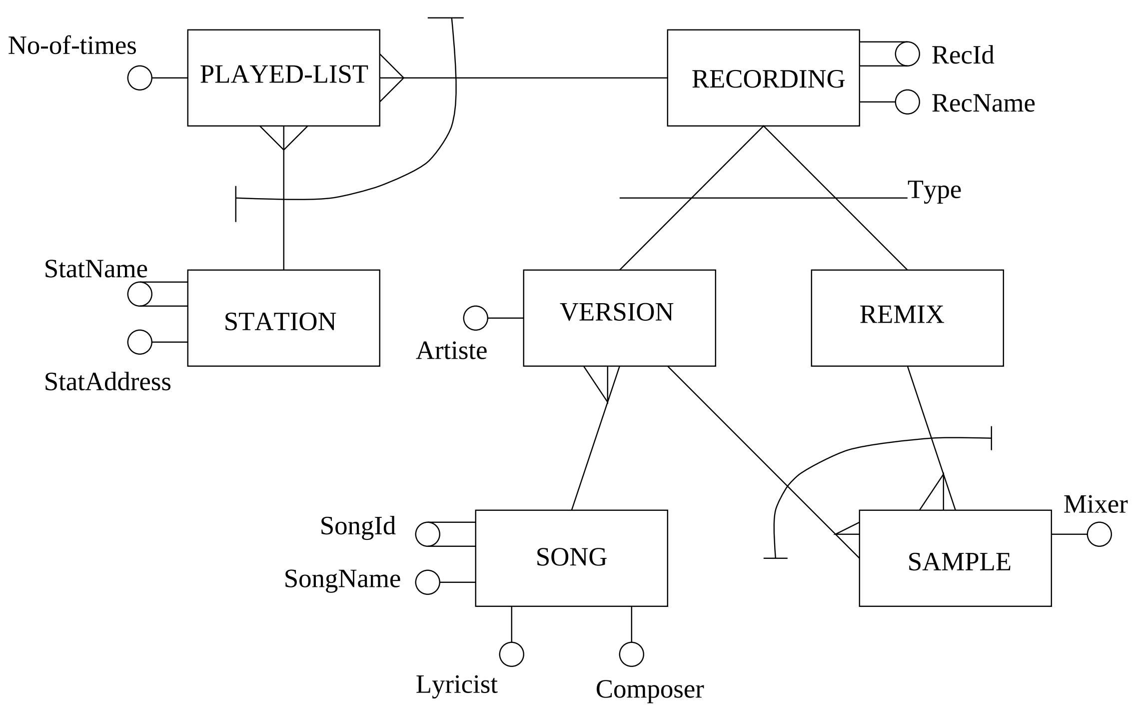


(c) Identification dependences



(ii) [7 marks] Whenever a record is played on the radio, royalties have to be paid by the radio station to the lyricist, the composer and the performing artiste. This process is mediated by the Performing Rights Society (PRS). The PRS receives lists of songs played by radio stations, computes how much the radio station should pay and passes the money on to the creative people. This is complicated due to re-recordings and the use of sampling.

You are given the following ER diagram by Baz Boomsteer, the PRS' systems analyst:



Using the ER diagram above (which you can take to be a correct diagram), produce a *logical data model* in the form of a collection of normalised relations.

List the *referential integrity constraints* that would need to be maintained.

(iii) [7 marks] You are now told by Baz that

(a) the Artiste property of VERSION is not in fact a single-valued property, but a set-valued property (e.g. to allow for duets), and

(b) the Mixer property, which says which DJ carried out the sampling to create the remix, is not a property of SAMPLE but of REMIX.

Therefore, the VERSION and SAMPLE relations you produced in (ii) above are not in third normal form.

- State which normal forms they violate and why.
- By giving some example data for the SAMPLE relation, exemplify the update anomalies that arise from problem (b).
- Describe the steps you would take to correct these problems. (You are not being asked to carry out the steps.)

Chapter 5

Implementing Relational Databases

Let's recap briefly. We analysed our organisation and produced a conceptual data model (ER diagram). The diagram is useful for all sorts of thing (e.g. discussion about whether the business is well-organised, etc.). Perhaps the most obvious use though is as a lead in to some form of computerisation. If that is the route you want to take, then you map the conceptual data model to a logical data model (a set of third normal form relations). You are then free to map the logical data model to any kind of implementation that you like (e.g. a traditional file-based system or some form of database system). In this chapter, we assume that you want to map it to a relational database system.

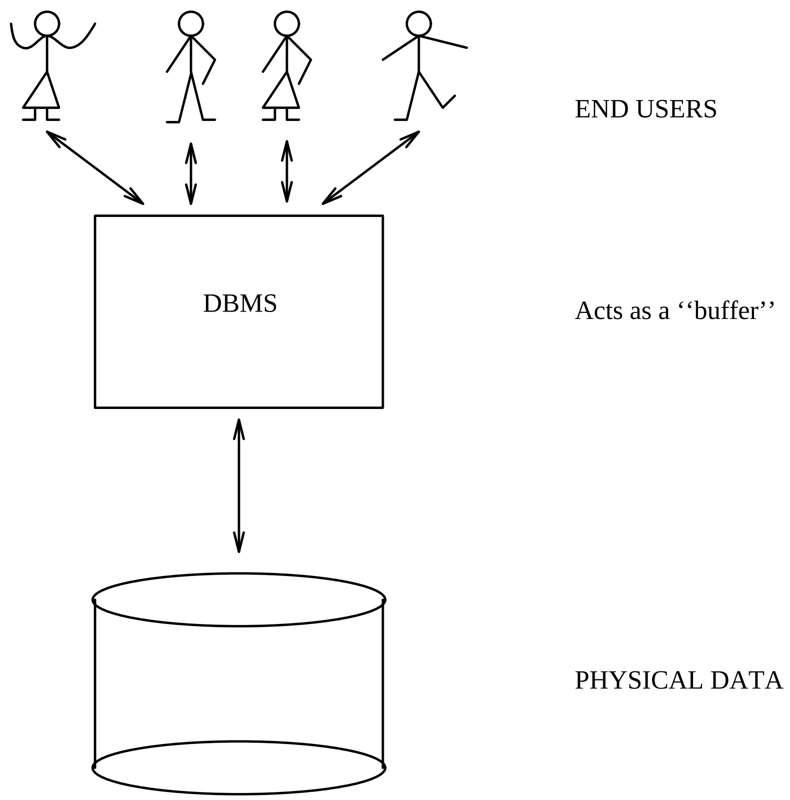
First, though, we will give ourselves some more terminology for talking about databases in general.

5.1 The Architecture of a Database System

You might recall from our definition of a database system (Chapter 1) that database systems afford us control over integrated data by virtue of there being a database management system (DBMS) and one or more schemas.

5.1.1 The DBMS

Physically, the data recorded in a database system is stored in one or more files on secondary storage. The DBMS sits between the users of the data and the physical database. Ideally, it controls *all* access to the data.



The DBMS is a *generalised* data manager: it intercepts user requests, interprets them, determines how to meet them (by inspecting the schema), initiates necessary actions, and returns results to the user.

A comprehensive DBMS provides at least 8 major functions:

1. Schema definition and maintenance services (i.e. updating the data dictionary/catalog —see later);
2. Data manipulation services (retrieval and update);
3. Integrity services (constraint enforcement, data validation);
4. Recovery services (transactions, backup, logging, rollback);
5. Concurrency control (locking, deadlock);
6. Security mechanisms (passwords, views);
7. Data communications interface services;
8. Performance monitoring services (statistics for tuning).

5.1.2 The Schema

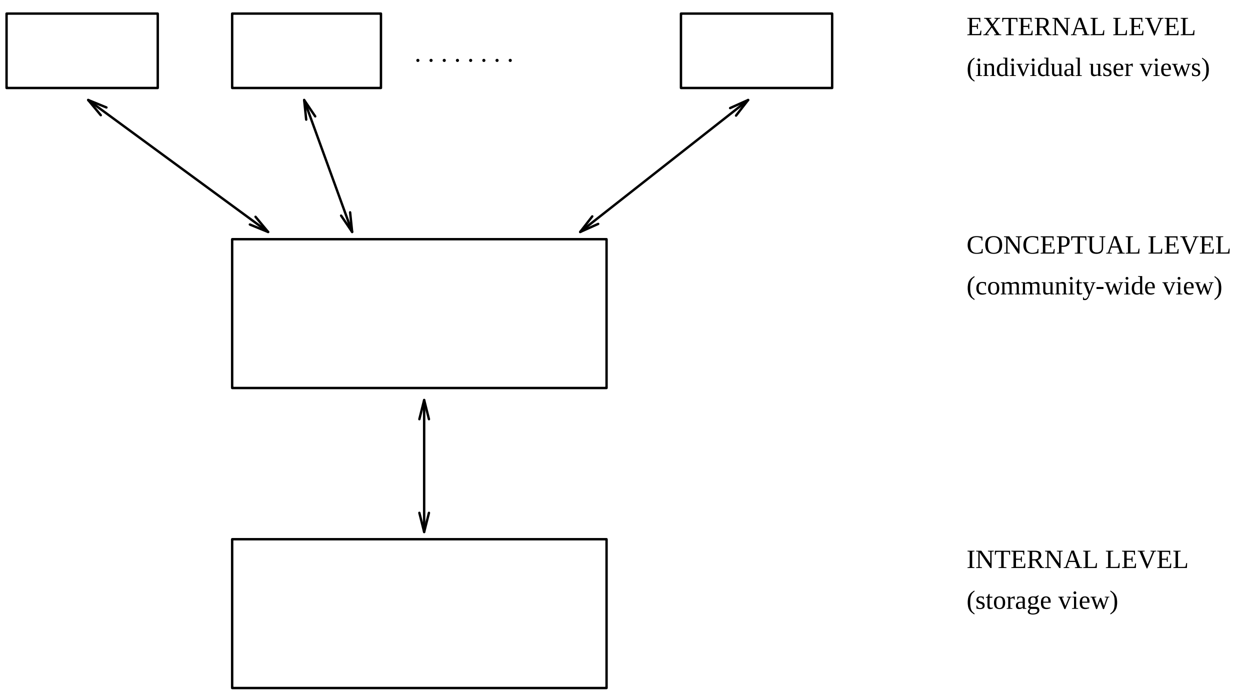
The DBMS determines how to satisfy a user request with reference to a definition of the structure of the stored data: the schema. There are three levels at which such a descriptions of the structure of the data could be given:

The Internal level: the physical storage level, describing how the data is actually stored (its low-level format, etc.);

The Conceptual level: the community-wide level, describing the data in a way that gives the view that the whole enterprise has, and reconciling the other two levels. (NB: This use of the word "conceptual" has no significant relationship to our use of that word in Chapter 3);

The External level: the user level, describing how end-users perceive the data.

Schematically, we have the following possible levels of description:



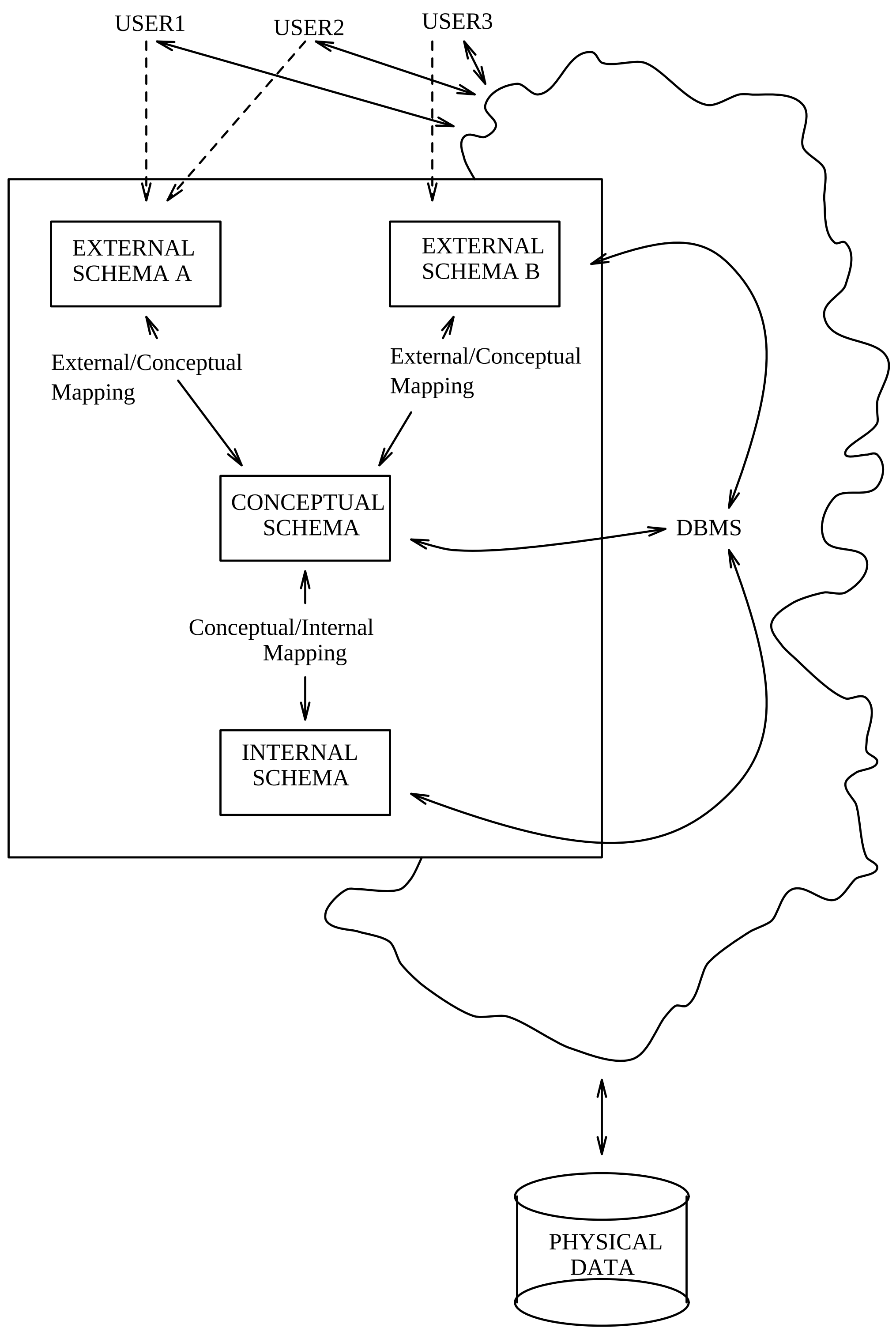
Given that there are several levels at which we could describe the data, which level should we choose as the level most suitable for our schema to capture?

This was investigated by the American National Standards Institute/Systems Planning and Requirements Committee (ANSI/SPARC) Study Group on Data Base Management Systems.

The ANSI/SPARC Reference Model

The Committee proposed an idealised architecture that would reflect all the levels. Their model can be used as a benchmark against which we can 'measure' commercial systems.

They proposed schemas for each level. Hence, there is an *internal schema*, a *conceptual schema* and numerous *external schemas*. In a system that has all three of these types of schema, the relationships between the different schemas must be defined and recorded. As the diagram shows, you must define and record a number of External/Conceptual Mappings and a Conceptual/Internal Mapping.


USER1
USER2
USER3
EXTERNAL SCHEMA A
EXTERNAL SCHEMA B
External/Conceptual Mapping
External/Conceptual Mapping
CONCEPTUAL SCHEMA
DBMS
Conceptual/Internal Mapping
INTERNAL SCHEMA
PHYSICAL DATA

An Idealised Example

Here's an example showing the three types of schema:

1. EXTERNAL (PL/1)		2. EXTERNAL (COBOL)	
DCL	1 LECTP	01	LECT_RECORD.
	2 LECT# CHAR(6),		02 LECTNO PIC X(6).
	3 SAL FIXED BIN (31);		02 DEPTNO PIC X(4).
CONCEPTUAL			
LECTURER			
	LECT_NUMBER		CHARACTER(6)
	DEPT_NUMBER		CHARACTER(4)
	SALARY		NUMERIC(5)
INTERNAL			
STORED_LECT			LENGTH = 18
	PREFIX		TYPE = BYTE(6), OFFSET = 0
	LECT#		TYPE = BYTE(6), OFFSET = 6
	DEPT#		TYPE = BYTE(4), OFFSET = 12
	PAY		TYPE = FULLWORD, OFFSET = 16
INDEX	LECT# = LECTX		

Notes:

1. Mappings are not shown but should be obvious.
2. User group 1 is not interested in department numbers.
3. User group 2 is not allowed to see salaries.

Why have three descriptions of the database?

Why should an idealised system have three levels of schema?

- each description is at a different level of abstraction:
  1. the internal schema is low-level; it describes how the machine represents the data and which access methods are used;

2. the conceptual schema is high-level; it is a global view and is independent of the low-level machine representation;
3. the external schemas provide a view of the data specific to an application, a user or a group of users; this tailors the data to that application's/ user's/ group's needs.

So, having all three types of schema might make implementation easier.

- the use of external schemas allows us to hide things from a user for the sake of simplicity or security.
- the *main* reason is that it gives us a high degree of *data independence*.

5.1.3 Data Independence

> Data independence is a characterisation of the extent to which one part of a database system is *immune* to changes in other parts of the system.

Data independence comes in two flavours:

Logical data independence: (also known as data-application independence) characterises the extent to which we can add new applications and modify existing applications without affecting other applications.

The ANSI/SPARC model has high logical data independence because, e.g.:

- to add a new application to a system, you simply define a new external schema and external/conceptual mapping;
- if a new field is added to the conceptual schema, the external schema of an application that doesn't use that field can remain unchanged.

Physical data independence: (also known as data storage independence) characterises the extent to which we can change the underlying storage mechanisms (e.g. change of access method, storage medium, physical distribution of data, format of data, etc.) without affecting user applications.

The ANSI/SPARC model has high physical data independence because, after a change to the internal schema, we would only need to change the conceptual/internal mapping.

In practice, no commercial system has ever looked exactly like the ANSI/SPARC architecture and so, while commercial systems do achieve some degree of data independence, none of them has as much data independence as it might otherwise have.

In practice, most commercial systems allow only one schema corresponding roughly to the conceptual schema. One is then allowed to augment this schema with internal level information of a limited kind. The extent to which you can also specify external level information varies from system to system. In some systems (most notably network and hierarchical database systems), you can specify subsets of the schema (subschemas) as an approximation to the idea of external schemas. In other systems (most notably relational database systems), more complicated external schemas can be specified. (In relational systems, they are called *views*. Different relational systems vary in how complex they allow views to be.)

5.1.4 Database Languages

Languages with which users can communicate with the DBMS are needed. In particular, we need:

A Data Definition Language (DDL): this is the language we use for defining the schema, i.e. for telling the DBMS about the *structure* of the data it can expect to be dealing with.

Data Manipulation Language (DML): this is the language we use for retrieving data from, inserting data into, modifying the data within and deleting data from the database, i.e. it concerns the *contents* of the database.

In principle, in the ANSI/SPARC idealised model, one could have a different DDL and DML for each schema (i.e. a DDL and a DML for the internal level, another pair for the conceptual level and even a pair for each different external schema). However, in practice, given that commercial systems don't distinguish the three types of schema very much (see previous subsection), you are provided with just one pair of languages which is supposed to be suitable for work at all levels. And, often the DDL and the DML have similar syntaxes, i.e. they appear as a single language which embraces both data definition and data manipulation functions.

5.1.5 Data Models versus Schemas

So far in the course we have looked at *data models*, and we have looked at two types of model that help us to analyse and design information systems. Now that we reach the implementation stage, we turn our attention to *schemas* because we turn our attention to telling a particular DBMS about the structure and format of the data that we want to hold. Hence, we have reached a stage where we will be mapping from a logical data model, which is an abstract DBMS-independent characterisation of our data storage requirements, to a schema, which is a DBMS-dependent DBMS-understandable description of our data storage requirements.

You can see, therefore, that I am drawing a distinction between data models and schemas. To repeat it: data models are abstract, DBMS-independent descriptions, whereas schemas are implemented DBMS-dependent descriptions. This distinction is often ignored or blurred in the literature.

(Here's an analogy: a data model stands in the same relation to a schema as an algorithm stands in relation to a program, i.e. the latter can be an implementation of the former.)

In the rest of this chapter, we assume that we are mapping our logical data model (set of relations) to a relational database management system (RDBMS). To be more concrete, we will often consider the mapping necessary to transform the model to the R:BASE RDBMS, which we possess in the Department. Two points need to be made: (1) R:BASE is pretty typical of relational database systems, so much of what we say will apply just as well if you ever map to some other relational DBMS, and (2) remember, that the logical model can just as easily be mapped to a non-relational database system or to a file-based system.

5.2 Relational Database Management Systems

Relational database systems represent data using *tables*. Each table has a *name* and a set of *columns*, each having a unique name (unique within the table). The column is associated with a *data type*. The values in the column must conform to the data type, e.g. integer, real, date, time, text, etc. The body of the table consists of a number of *rows*.

There is an obvious correspondence between the features of the relational *model* and features of a relational database system[1]:

[1]A fair question to ask is: why do we have yet another switch of terminology? The

Relational model terminology	*Relational database terminology*
Relation	Table
Attribute	Column
Domain	Data type
Tuple	Row

It might now seem that the next design step is easy: we can assume that there is a one-to-one correspondence between the relations, etc. of our logical data model and the tables, etc. that we implement in the database system.

Indeed, that is the assumption that we will make in the next section. However, it is an assumption that we must revisit later in the course. Often, such a direct mapping will lead to a database that cannot satisfy our performance requirements, i.e. won't be fast enough. In such a case, and this is by no means rare, we will have to tweak the schema.

5.3 R:BASE: A Relational Database Management System

In the Department, we have an RDBMS called R:BASE, marketed by Microrim.[2] R:BASE is very similar to most other commercial RDBMSs. Any lessons learnt from using R:BASE should stand you in good stead for any other RDBMSs you encounter.

You can communicate with R:BASE in two ways. The main way is to navigate around its system of menus. Since we will soon be using R:BASE's menus to illustrate topics, now would be a good time to try the second lesson in the R:BASE tutorial. The instructions are as they were in Chapter 1, but this time choose lesson 2 (Navigating in R:BASE and Using Help). Despite what it says, the lesson should take about 10 minutes.

In general to invoke R:BASE, at the DOS prompt you type:

```
rbase
```

answer is two-fold: (1) it serves to emphasise the switch from data model to database system, and (2) commercial relational systems tend to use terminology such as "table", "column", etc. to make their products more marketable.

[2] R:BASE and Microrim are registered trademarks of Microrim, Inc.

Once the menus appear, you navigate around them using the keys summarised below. There are some function keys which, whenever pressed, have the same function:

Function	Key	Purpose
Help	**F1**	Displays help for the current menu, dialog box, or screen
	Shift-F1	Lists what the other function keys currently can be used for
Do it	**F2**	Performs an action from where you are
Database Information	**F3**	Displays the names of the tables, columns, variables, views, forms, and reports in the open database

Then there are keys that you use to select things from menus:

Function	Key	Purpose
Main menu and other horizontal menus		
Highlight	$\rightarrow$	Moves right
	$\leftarrow$	Moves left
	Tab	Moves right
	Shift-Tab	Moves left
	Alt-key	Moves to option whose first letter is pressed with **Alt**
Vertical menus		
Highlight	$\uparrow$	Moves up
	$\downarrow$	Moves down
	Home	Moves to first option
	End	Moves to last option
	Alt	On some screens, toggles to screen
	Esc	Highlights *Exit*; abandons action
Select	**Enter**	On check-off and sorting menus, checks or selects options
All menus		
Do it	**Enter**	Executes immediately; goes to next menu
	F2	

You can use the mouse instead of pressing keys. (To my mind, there's so much that you must type when using databases, that use of the function keys is faster than continually switching between the mouse and keyboard.)

Mouse Button	Key Equivalents	Purpose
Left (one click on the option)	Arrow keys	Highlights an option
Left (double-click on the option)	**F2**	Highlights and executes an action
Left (one click off the menu)	**Alt**	Moves from the menu to the screen

The other way to communicate with R:BASE is to explicitly type commands using the R:BASE command line. On the Main Menu, highlight *Exit* (e.g. by pressing **Esc**), then choose *R> prompt*. You will get the R:BASE command line; its prompt looks as follows:

```
R>
```

To return to the R:BASE Main Menu, enter **MENU** or press **Esc**.

If the command you are typing is longer than the width of the screen, either just keep typing, or type + as the last thing on the line, hit **Enter**, which will give you the continuation prompt (+>), and then continue typing, e.g.:

```
R> SELECT * FROM STUDENTS +
+> WHERE SNo = '3'
```

If you make an error in a command, you can recall it in order to edit it. Pressing **End** at the prompt will recall the entire last command. You can now edit the command using **Insert** (switches between insert and overwrite mode), **Delete**, the arrow keys, **Home** (go to first character of the command), and **End** (go to last character of the command). Once you've finished, go to the last character and press **Enter** and R:BASE will try to execute your edited command.

As presently set up, R:BASE is not case sensitive at all.

5.4 Defining the Database Structure to the DBMS

There can be four ways in which a schema can be defined to a DBMS:

- Using a CASE tool: CASE stands for Computer Aided Software Engineering. CASE tools allow the user (among other things) to draw ER diagrams using a mouse on a bit-mapped workstation. The system can then do an automatic mapping from entity types on the diagram to relations in a logical model, which it stores as a schema.

- Using a compiled DDL: some systems provide a Data Definition Language for which you enter suitable schema describing commands into a file using an editor and then put this file through a compiler which translates the statements and builds the schema.

- Using an interpreted DDL: other systems have DDLs that are interpreted. You can enter schema definition commands at a prompt line and these are actioned immediately.

- Using a menu-based system: other systems have a menu-based interface which guides you through the business of defining the schema.

R:BASE offers the last two of these, so we will look at them in more detail.

We will consider two ways in which the structure of the database might be described to the DBMS: the use of a command-line DDL and the use of menu-based definition facility.

5.4.1 SQL as a DDL

A language in very common usage in RDBMSs is SQL. We'll cover the background to SQL later in this course when we look at querying databases. For now we ignore its broader aspects and simply look at the functions it provides to enable us to define a database. This is the language used by R:BASE, so you can try this stuff out.

Here are some relevant commands (simplified in most cases). As a convention I use uppercase for keywords and lowercase for arguments.

To try these commands out, start R:BASE and get to the command line using the instructions described earlier.

Creating a schema

Use this command as a first step to tell the system about a new database.

```
CREATE SCHEMA AUTHORIZATION database_name
```

Obviously other commands, e.g. to create tables, etc., can only be issued once a database has been named in this way.

The CREATE SCHEMA command will create three files in whatever directory you issue the command: database-name1.rbf, database-name2.rbf and database-name3.rbf. database-name1.rbf will contain the definition of the database, database-name2.rbf will contain the data in the database, and database-name3.rbf will contain the indexes. "These three files operate together and must never be separated. They are not text files and cannot be modified directly by another program, text editor, or word processor without damaging the database."[R:BASE User Manual, p.2-62]

In fact, no other commands can be issued at all until a database has been *opened*, i.e. before you can create tables, insert data, etc., etc., you must open a database. The CREATE SCHEMA command does in fact automatically open the database that you are creating, so you can go right ahead and start creating tables. However, to issue commands to an existing database, you must open it on the Main Menu before going to the command line. On the Main Menu, the *Databases* option calls up a drop-down menu which has an option to *Change directory*. Use this option to get to the directory from which you created the database in the first place. The name of the database will then appear towards the top of the drop-down menu. Highlight it and press *Enter*. In the bottom left-hand corner of the screen, you will see the word "Database" and then the name of the database that you have just opened.

(Note: Many names, e.g. for databases, tables, columns, etc., in R:BASE have only 8 significant characters.)

Creating a table

To define a new table for an open database.

```
CREATE TABLE table_name
         (column-defn [, column-defn] ... [, UNIQUE(column-list)] )
```

A `column-defn` is of the form:

```
column-name datatype [NOT NULL]
```

or

```
column-name datatype [NOT NULL UNIQUE]
```

Examples are:

```
CREATE TABLE STUDENTS
  (Stud_no      TEXT 5   NOT NULL UNIQUE,
   Stud_name    TEXT 25  NOT NULL,
```

```
   Age          INTEGER,
   Start_date   DATE)

CREATE TABLE ENROLLMENTS
  (Stud_no      TEXT 5  NOT NULL,
   Course_code  TEXT 3  NOT NULL,
   Exam_mark    INTEGER,
UNIQUE (Stud_no, Course_code) )
```

The effect of these should be obvious. We create a table of the given name, with the given columns and their data types, and we can specify whether we want the database to disallow the entry of null values or duplicate values into a column or set of columns. Clearly, this is the way we define primary keys. In `STUDENTS` above, `Stud_no` will effectively act as a key because it cannot be null and must contain unique values. In the second example, `Stud_no` and `Course_code` together will form the key: neither can be null and we have specified that taken together they must be unique. Thus in both examples we capture the two main integrity constraints concerning keys: their uniqueness and entity integrity (their 'non-nullness'). R:BASE is actually quite an improvement over many RDBMSs in this regard, and so this is something to watch out for: many do not allow the specification of keys in an adequate way, particularly composite keys, such as that for `ENROLLMENTS`.

Note that in R:BASE if two tables have a column name in common (as with Stud_no above), then you must give them the same data type in each table. Note though that this doesn't mean that R:BASE will now enforce referential integrity between these two columns. To achieve this, you must explicitly define a rule (see later). (Note that if you define two columns of the same name for a single table, the second definition will be taken to override the first definition.)

Data types

The data types R:BASE and most RDBMSs allow are only a weak approximation of domains, e.g. you cannot have the domain of Student_nos. You have to make do with a data type such as `TEXT 5`. Here is a *selection* of the data types which R:BASE allows:

DATE: A date. The default format is MM/DD/YY. (This can be changed.)

INTEGER: An integer from -999999999 to +999999999.

TEXT n**:** n alphanumeric characters.

(There are a further 13 data types, which we won't bother with here.)

Rules

The `RULES` command defines integrity constraints.

```
RULES 'message' FOR table-name SUCCEEDS WHERE clause
```

or

```
RULES 'message' FOR table-name FAILS    WHERE clause
```

`SUCCEEDS` specifies that a row must meet the conditions included in the `WHERE` clause in order to be added to the database. `FAILS` specifies that a row must not meet the conditions included in the `WHERE` clause in order to be added to the database.

Here are some examples which use the following schema:

> STUDENTS(**Stud_no:** Stud_name, Proj_sup, Proj_no, Mark_attained, Possible_mark)
> COURSES(**Course_code:** Description)
> ENROLLMENTS(**Stud_no, Course_code:** Exam_mark)
> PROJECTS(**Proj_sup, Proj_no:** Description)

```
1) RULES 'Invalid exam mark'
     FOR ENROLLMENTS SUCCEEDS WHERE Exam_mark <= 100

2) RULES 'Mark attained must not be > total possible mark'
     FOR STUDENTS SUCCEEDS WHERE Mark attained <= Possible_mark

3) RULES 'Student no must be non-null and unique' FOR STUDENTS
     SUCCEEDS WHERE Stud_no IS NOT NULL AND
                    Stud_no NOT IN (select Stud_no FROM STUDENTS)

4) RULES 'Duplicate or non-null Student no/Course code'
     FOR ENROLLMENTS
     SUCCEEDS WHERE Stud_no  IS NOT NULL AND
                    Course_code IS NOT NULL AND
                    Stud_no NOT IN
                     (SELECT Stud_no FROM ENROLLMENTS E2
                        WHERE E2.Course_code = ENROLLMENTS.Course_code)

5) RULES 'Invalid course code' FOR ENROLLMENTS
         SUCCEEDS WHERE Course_code IN (SELECT Course_code FROM COURSES)
```

```
6) RULES 'Invalid project chosen for student' FOR STUDENTS
        SUCCEEDS WHERE Proj_sup IN
          (SELECT Proj_sup FROM PROJECTS P1, PROJECTS P2
             WHERE P1.Proj_no = P2.Proj_no)
```

(Obviously a full understanding of what you can do with these rules awaits a full understanding of what can go into a `WHERE` clause. We will look at these when we return to SQL as a DML later.)

Rule (1) gives us something closer to the idea of a domain by restricting the range of the Exam_mark column to integers less than or equal to 100. (This is still only an approximation to a domain. Why?)

Rule (2) is something of a one-off kind of constraint that we hopefully spotted during design.

Rule (3) gives another way of specifying that Stud_no is the primary key of STUDENTS. It checks that the new course code is not null and not already in the table. (Obviously, it is easier to do this in the first place when using the CREATE TABLE command.)

Rule (4) gives another way of specifying that the composite key of ENROLLMENTS is Stud_no and Course_code. It is too complicated to explain here. Come back and look at it again after reading chapter 7 on SQL. (Obviously, again, it is easier to specify this key in the first place when using the CREATE TABLE command.)

Rule (5) is a simple referential integrity constraint, defining ENROLLMENTS.Course_code to be dependent on COURSES.Course_code.

Rule (6) is very complicated in syntax but it too simply implements a referential integrity constraint. However, this constraint makes the composite foreign key comprising STUDENTS.Proj_sup and STUDENTS.Proj_no dependent on the composite key of PROJECTS.

Other commands

There is also the `ALTER TABLE` command to add columns to a table, the `DROP TABLE` command to remove tables from the database, and the `DROP RULE FOR table-name` command to remove all rules from the specified table (there are more elaborate forms of this command).

To see what you've created:

> `LIST TABLES` displays a list of tables in the database.
> `LIST TABLE table-name` displays the header of the specified table.

By following the instructions that have been given in this section, create a simple R:BASE database for, e.g., keeping track of your music collection:

ARTISTE(Artiste_no, Artiste_name)
ALBUM(Album_no, Album_name, Artiste_no)

(Important: do not define any integrity rules or keys.)

5.4.2 Schema Definition using a Menu-Based Interface

The other way to define a database in R:BASE, other than typing commands at the command line, is the use of menus.

Now is the time to do another lesson from the tutorial. Invoke it as before, but this time highlight lesson 5 (Designing and Building Databases). It says you need to have completed lessons 3 and 4. I don't think this is really true, so ignore those lessons for now. The lesson comes in two parts, and at the start you are offered the choice of which part to look at. The part you're interested in is part 2, which concerns schema definition in R:BASE. Part 1 gives Microrim's ideas of how to analyse and design a database (their equivalent of chapters 2, 3 and 4 of these notes). You are obviously free to look at Part 1, but where there are discrepancies between their approach and mine, you should, under exam conditions, use *MY* approach.

Part 2 of the tutorial should give you a good feel for how the menu-based approach works. It ought to take about 10 minutes.

Create another simple database. This time use the menus to do it, based on what you have learnt from the tutorial. For example, create one to keep track of your exam marks:

COURSE_UNIT(Course_code, Year_taken, Total_mark)
ASSESSMENTS(Ass_no, Course_code, Ass_mark)

(Ignore the questions about indexing columns, and, again, don't bother to define any integrity rules or keys.)

5.5 Data Dictionaries

When a user performs any DDL statements, the structure of the database is changed, e.g. creating a new table, altering a table, granting access privileges. Hence the DBMS has to maintain information about the structure of the database. This includes information saying what tables are in the database, what columns they have, and so on. This database structure information we have referred to as the schema. The place where the DBMS stores the schema is these days referred to as the *data dictionary*.

- Whenever a DDL statement is executed, the content of the data dictionary is *amended*.
- Whenever a DML statement is executed, the content of the data dictionary is *consulted*.

A general trend in relational DBMSs is that the data dictionary information is stored as a set of tables, i.e. in relational format. The advantage of storing the data dictionary as tables is that it can be queried by users in the same way as user-defined data.

This is true in R:BASE: the data dictionary is a set of tables. One quirk, though, is that you cannot query the data dictionary using the menu system. You must get to the R:BASE command line to look at these tables.

There are up to 8 tables; for now, we will look at only two, SYSINFO and SYSRULES. Here's an example of (only two of the columns of) the SYSINFO table:

SYSINFO	
SYSSNAME	SYSCNAME
course_unit	
course_unit	course_code
course_unit	year_taken
course_unit	total_mark
assessments	
assessments	ass_no
assessments	course_code
assessments	ass_mark

You can see that for each table, it has a row giving the table name and a blank entry for the column name and then a number of other rows which say what columns the table has.

(Question: Imagine what other columns the SYSINFO table might have and then say why this is a poor design for such a table.)

Here's an example of one row from (and only three of the columns of) the SYSRULES table:

SYSRULES		
SYSTABLE	SYSMSG	SYSWHERE
course_unit	Value for course_code cannot be null.	course_code IS NOT NULL

To see the contents of these tables for yourselves for one of the databases that you have created, go to the R:BASE command line and try, e.g.:

```
SELECT * FROM SYSINFO

SELECT * FROM SYSRULES
```

The SELECT command is SQL's query command. Once you've learned it in its full glory (Chapter 7), then you can use it to do very sophisticated queries on these tables (indeed, just as sophisticated as those you can do on any other tables).

While you can freely query these tables, in principle you shouldn't update them directly, i.e. the only way to cause them to be updated should be by issuing the DDL commands or using the appropriate options on the menus. (Having said that, I tried updating SYSINFO manually; I was successful; but R:BASE seemed to take no real notice of my update.)

The term 'data dictionary' has an additional much broader usage than we have been using so far. Any database (or part of a database) which contains *data about data* can be called a data dictionary. For example, when you are designing a database (and the rest of an information system), you may want to store conceptual and logical models in electronic form. They, just like schemas, can be stored in a data dictionary.

Thus, these are the services which a data dictionary *might* support:

1. Run-time support for the DBMS, i.e. this is the storage of the schema that we have focussed on so far.

2. The storage of 'products' of the development process in one place. Putting this into a data dictionary has the following possible advantages:

    - Documentation standards can be enforced.
    - Communication between designers is enhanced.
    - By querying the data dictionary, we can analyse the effects of design changes.

- Tools might be available for automatic design verification.

It is clear that every database has a data dictionary (in some form) for the first of the above two tasks. However, not all support the second task. If your database doesn't (as, for example, R:BASE doesn't), then it is possible to buy standalone data dictionaries to support the design process. (Oracle sells one, for example.) Of course, this will be separate from the data dictionary of your database. Sometimes these separate data dictionaries are able to communicate with the rest of your database system; sometimes they cannot.

To encourage more consistency in this area, data dictionaries are now being standardised. The standardisation has, unnecessarily, involved yet another switch of terminology. Data dictionaries should now, it seems, be referred to as *information resource dictionary systems*.

5.6 Now use the database...

Once you have defined the schema of a database, you are in a position to put it into operation, i.e. give it to your user community and let them insert, modify, delete and retrieve data. In this section, we take a brief look at how to do this using some of the features of R:BASE.

There are lots of ways to do this in R:BASE. For the moment, we'll look only at the browser.

Begin R:BASE as normal (i.e. type `rbase` at the DOS prompt). On the Main Menu, open a database, i.e. highlight *Databases* on the menu and select one of the named databases on the drop-down menu. After this, highlight *Info* on the Main Menu. Choose a table. This will show the table contents, and allow you to insert, modify and delete. Have a good play. (Remember that **F1** gives context sensitive help, and **Shift-F1** gives help about function keys.)

5.7 Exercise

EXAM QUESTION: Question 9, Parts (ii), (iii) and (iv) only, 1991

A database schema consists of a number of relations. Each relation has a unique name and a number of attributes. Attributes themselves are simply names. While attributes within a relation must be different, the same attribute name can be used in more than one relation.

Each attribute is associated with a domain. It is possible for an attribute name which appears in more than one relation to be associated with different domains, one for each relation in which it appears.

Domains are arranged in a type hierarchy. For example, integers in the range 0..10 are a subtype of short integers; short integers and long integers are subtypes of integer. This type hierarchy allows for a flexible implementation of union compatibility. (Ignore this last sentence —it's explained in Chapter 6.)

Attributes of a relation have a property which signifies whether or not they are primary key attributes; this allows for enforcement of entity integrity.

(ii) [6 marks] Produce a *conceptual data model* in the form of an entity-relationship diagram that captures the description. You should formulate the concepts as concisely as possible, and state any assumptions that you make.

(iii) [4 marks] Using the results of (ii), produce a *logical data model* in the form of a collection of normalised relations.

(iv) [3 marks] List the *referential integrity constraints* that would need to be maintained for this logical data model.

Chapter 6

Relational Algebra

So far we have concentrated on the design of appropriate structures for the data that we want to hold. We haven't really discussed processing of data at all. (Although at the end of the last chapter we did briefly discuss how to insert, modify, delete and retrieve data using R:BASE's browser.) Clearly, processing the data is important and we need something more powerful than a browser for a single table. We need to bring data together from lots of tables, pick out relevant bits and display them to the user.

This is the functionality which the database's Data Manipulation Language (DML) will give us. However, before we look at the facilities and languages provided by commercial systems, we will revert to a more theoretical perspective. By taking a theoretical perspective we will find it easier to learn commercial languages, easier to compare commercial languages and easier to see how DMLs are actually executed in the most efficient way.

6.1 Relational Calculus and Relational Algebra

The relational model provides very high-level manipulation facilities to allow querying and updating. There are two basic query systems: relational calculus and relational algebra.

These two query systems were devised by Tedd Codd when he devised the rest of the relational model. One very important point is that they provide only what Codd called a *data sublanguage*. These query systems are *sub*languages in the sense that they do not aim to offer all the computational facilities of normal programming languages. In particular, they do not offer any way of doing iteration (no recursion, no loop commands) and no way of doing conditionals. Codd argued that plenty of perfectly good languages already provide iteration and conditionals. There was therefore no need to reinvent the wheel. Rather, he would devise a sublanguage tailored to manipulating data held in relations. If you then need iteration or conditionals (and, given how high-level and powerful the facilities of his sublanguage are, the need for explicit iteration and conditional operations ought to be rare), you can get them by embedding commands of the sublanguage into programs written in some other programming language.

The calculus and the algebra share the property of *relational closure*: the results of any manipulation on relations is another relation.

Another property that the calculus and the algebra share is the property Codd called *relational completeness*. Basically, Codd said that any language which has the same expressive power as the relational calculus would qualify as being relationally complete. Subsequently, it was proved that the relational algebra had *identical* expressiveness to the calculus. To give you a *very rough* idea of the sort of expressiveness we are talking about here, a relationally complete language can define any horizontal or vertical subset of the Cartesian products of the relations in the system. (That's a rough idea, not a definition.)

Relational Algebra

This is an *algebra*-based language. It is fairly procedural. Data manipulation takes the form of requesting that sequences of operations be performed on relations. It is rare for commercial query languages to be based directly on the algebra.

We'll leave further consideration of the algebra until the next section.

Relational Calculus

This is a language based on *predicate calculus*. (Predicate calculus is the logic you should have met in DIM that uses variables and quantifiers, e.g. $\forall x(\texttt{ELEPHANT}(x) \rightarrow \texttt{GREY}(x))$.) This gives a very non-procedural language. Data manipulation takes the form of declaring conditions that an answer will satisfy, and leaving the job of pulling the answer out of the database to a general-purpose evaluation mechanism in the DBMS. Commercial query languages tend to be based (in some cases, somewhat loosely) on the calculus, rather than the algebra.

Let's expand on this. The calculus is more *declarative* or *non-procedural* than the algebra. You do not specify the steps saying *how* to retrieve some answer to your query. Rather, you give a definition of the conditions a relation would have to meet for it to be an answer to the query. It is then up to the query evaluation system (which is a part of the DBMS) to supply the tuples that satisfy this definition. (The algebra, by contrast, is more procedural: a certain ordering among the operations is always explicitly specified in a relational algebra expression.) To sum this up, in the calculus you specify the *what*, not the *how*. (But remember that formally they are equivalent in power because both are relationally complete.)

The predicate calculus is adapted to make it suitable for use as a query language. It may be adapted to give a calculus in either of two flavours: *tuple relational calculus* or *domain relational calculus*. In both, a fundamental notion is that of a *variable* which *ranges over* some relation. We explain these flavours in more detail below, making use of the following relational model:

STUDENTS(**SNo:** SName)
ENROLLMENTS(**SNo, CCode:** Mark)

relational tuple calculus: The quantification in this calculus is over tuples, i.e. the variables range over tuples of the relation, i.e. a variable defined on a relation, R, will, at any given time, have some tuple of R as its value. For the relations given above, the query "Get student numbers and names for students who study 'dad' " would be rendered as:

$$\{x\texttt{.SNo, } x\texttt{.SName} \mid \texttt{STUDENTS}(x) \wedge \exists y \ \texttt{ENROLLMENTS}(y) \wedge \\ y\texttt{.SNo = } x\texttt{.SNo} \wedge \texttt{y.CCode = 'dad'}\}$$

This calculus has been implemented (with a heavily sugared syntax) as languages such as SQL (a very common language) and QUEL (less common, mainly associated with the INGRES database). (In only some implementations are SQL and QUEL still relationally complete.)

relational domain calculus: The quantification in this calculus is over attribute domains, i.e. the variables range over domains of the relation, i.e. a variable will, at any given time, have some domain value as its value. For the relations given above, the query "Get student numbers and names for students who study 'dad' " would be rendered as:

$$\{x_1\texttt{, } x_2 \mid \texttt{STUDENTS(SNo : } x_1\texttt{, SName : } x_2\texttt{)} \wedge \\ \texttt{ENROLLMENTS(SNo : } x_1\texttt{, CCode : 'dad')} \}$$

This calculus has been implemented (in an even more heavily sugared syntax) as the language QBE. (In only a few implementations is QBE still relationally complete.)

Although it is rare for commercial query languages to be based directly on the relational algebra (see above), this is the approach we are going to look at in this chapter. It is important that we look at the algebra for a number of reasons: (1) we have seen that the relational calculi (and, therefore, the languages based on them) specify the 'what' and not the 'how', leaving it to the DBMS to work out how to get the requested data. One way the DBMS might do this is to translate from the calculus to the algebra (which is quite straightforward); (2) there are a few systems that *do* offer the algebra as a query language. For example, the Department owns a simple RDBMS called db++, which has an algebraic query language. As another example, the other RDBMS the Department owns, namely R:BASE, while it offers more friendly query facilities, also offers some algebra facilities for those who want them; (3) many important results in database work are expressed using the algebra. For example, when we come to look at query optimisation (making queries run faster), we will do this by using the algebra; (4) when we come to look at the SQL query language in the next chapter, although SQL is (deep down) a language based on the tuple calculus, students find it much easier to learn SQL by thinking of it as a user friendly syntax for the algebra; and (5) as SQL has grown in popularity, new features have been added to it. Ironically, many of these new features lose sight of the fact that SQL is a calculus-based language, because the sorts of things that have been added or that people are proposing should be added are relational algebra operators. Thus, your understanding of future versions of SQL may depend on knowing the algebra.

6.2 Relational Algebra

Relational algebra provides a set of operators which operate upon one or more relations and *produce another relation.*

Codd defined eight operators:

traditional 'set' operators: these are union, intersection, difference and Cartesian product; they are like the set operators whose names they share, but their definitions have been extended to handle relations.

special relational operators: these are restrict, project, join and divide.

6.2.1 Traditional 'set' operators

Cartesian Product

Example:

STUDENTS	
Student_no	Age
a1	20
a2	20

COURSES	
Coursecode	Year_given
dad	2
sra	2
nlp	3

product(STUDENTS, COURSES)			
Student_no	Age	Coursecode	Year_given
a1	20	dad	2
a1	20	sra	2
a1	20	nlp	3
a2	20	dad	2
a2	20	sra	2
a2	20	nlp	3

i.e. product concatenates each tuple of STUDENTS with each tuple of COURSES.

(Question: what does the above result represent? Does it represent who takes what? Or who could take what? *Ask the client.* Beware of *connection traps*, i.e. assuming the existence of non-existent relationships.)

Union, Intersection and Difference

All the other *'set'* operators require that the operands be *union-compatible*. Roughly, this means that they must have the same degree and have pairwise comparable domains. (As a convention we will demand equal attribute names.)

Examples:

GRANT_FUNDED		
Student_no	Name	Dept_code
a1	E.White	CS
a2	E.Brown	EL
a4	J.Black	CS

SPONSORED		
Student_no	Name	Dept_code
a2	E.Brown	EL
a3	M.Green	MA

ALL_STUDENTS		
Student_no	Name	Dept_code
a1	E.White	CS
a2	E.Brown	EL
a3	M.Green	MA
a4	J.Black	CS

DOUBLE_FUNDED		
Student_no	Name	Dept_code
a2	E.Brown	EL

GRANT_ONLY		
Student_no	Name	Dept_code
a1	E.White	CS
a4	J.Black	CS

STAFF			
NHI_no	Name	Age	Dept_code
863-A-4	W.Baker	37	CS
993-A-8	P.Butcher	29	PO

- ALL_STUDENTS = union(GRANT_FUNDED, SPONSORED)
- DOUBLE_FUNDED = intersect(GRANT_FUNDED, SPONSORED)
- GRANT_ONLY = difference(GRANT_FUNDED, SPONSORED)
- STAFF is not union-compatible with any of the other relations: the degree is different and also the NHI_no attribute is not drawn from the same domain as that from which Student_nos are drawn.

6.2.2 Special Relational Operators

Restrict

Restrict is sometimes called 'select'. It acts as a filter. It allows only certain *tuples* to pass through: those that satisfy a predicate. Thus the result is a 'horizontal subset' of the original.

The syntax is:

restrict(R, *predicate-on-attributes-of-R*)

Examples:

restrict(ALL_STUDENTS, Dept_code = 'CS')
restrict(ALL_STUDENTS, (Name = 'E.Brown') $\vee$ (Name = 'J.Black'))

The predicate is constructed from comparison operators $(=, \neq, >, <, \leq, \geq)$ boolean operators $(\wedge, \vee, \neg)$, attributes of the relation and values.

The resulting relation has the same attributes but (possibly) fewer tuples:

restrict(ALL_STUDENTS, Dept_code = 'CS')		
Student_no	Name	Dept_code
a1	E.White	CS
a4	J.Black	CS

Project

Project is also a filter. It allows only certain *attributes* to pass through: those explicitly requested. The result is a 'vertical subset' of the original relation.

Note that project also removes any duplicate tuples that result from stripping out unwanted attributes.

The syntax is:

project(R, *set-of-attributes-of-R-to-be-returned*)

Examples:

project(ALL_STUDENTS, {Student_no, Name})	
Student_no	Name
a1	E.White
a2	E.Brown
a3	M.Green
a4	J.Black

project(ALL_STUDENTS, {Dept_code})
Dept_code
CS
EL
MA

Project reduces the degree of the relation and (possibly) its cardinality (if there are duplicates to remove). (Question: can you be more precise about the circumstances that might result in a relation with a lower cardinality?)

Join

The join operator provides a way of combining information from different relations. It is one of the most powerful features of relational systems.

The syntax is:

θ-join(R1, R2, θ)

θ is the join predicate, which relates attributes of R1 and R2. This general form of join is referred to as a theta-join.

Joins correspond to forming the Cartesian product of R1 and R2 and then using θ in a restrict on this product. A project might then be applied to this.

Many variations of the join have been proposed. The two most important are *equi-joins* and *natural joins*. In both of these, θ simply involves the equality predicate. The difference between equi- and natural joins is that in natural joins there is a project on one of the equated attributes:

An equi-join of R1(A, B) and R2(C, D) is equivalent to

restrict(product(R1, R2), R1.B = R2.D)

A natural join of R1(A, B) and R2(C, D) is equivalent to

project(restrict(product(R1, R2), R1.B = R2.D), {A, B, C})

Examples of joins:

STUDENTS			
Student_no	Name	Dept	Grade
a1	E.White	CS	40
a2	E.Brown	EL	50
a3	M.Green	MA	35
a4	J.Black	CS	60

DEPARTMENTS		
Dept_code	Head	Pass_mark
CS	Mander	35
EL	Fidler	40
MA	Fountain	55

equi-join(STUDENTS, DEPARTMENTS, Dept = Dept_code)						
Student_no	Name	Dept	Grade	Dept_code	Head	Pass_mark
a1	E.White	CS	40	CS	Mander	35
a2	E.Brown	EL	50	EL	Fidler	40
a3	M.Green	MA	35	MA	Fountain	55
a4	J.Black	CS	60	CS	Mander	35

nat-join(STUDENTS, DEPARTMENTS, Dept = Dept_code)					
Student_no	Name	Dept	Grade	Head	Pass_mark
a1	E.White	CS	40	Mander	35
a2	E.Brown	EL	50	Fidler	40
a3	M.Green	MA	35	Fountain	55
a4	J.Black	CS	60	Mander	35

(Note on syntax: since nat-join removes one of the two attributes that are made equal we can come up with some syntactic conventions: (1) the two attributes might have the same name; (2) if they do, then there's no need to specify the equality (e.g. if in DEPARTMENTS, Dept_code had instead simply been called Dept, then we could write nat-join(STUDENTS, DEPARTMENTS) with the assumption that the two attributes with the same name would be made equal); (3) if they don't have the same name, then an explicit equality must be specified, and we will assume that the first of the two is kept, the second is discarded (as above).)

θ-join(STUDENTS, DEPARTMENTS, (Dept = Dept_code) $\wedge$ (Grade > Pass_mark))						
Student_no	Name	Dept	Grade	Dept_code	Head	Pass_mark
a1	E.White	CS	40	CS	Mander	35
a2	E.Brown	EL	50	EL	Fidler	40
a4	J.Black	CS	60	CS	Mander	35

Divide

Divide is a sort of filter. It is very powerful, but rarely used. It takes R1, the *dividend*, which should be of degree $m + n$, and R2, the *divisor*, which should be of degree m. The m attributes are the *undivided attributes*;

the n attributes are the *divided attributes*. R1's divided attributes must be pairwise-compatible with all R2's attributes. (In practice, we take this to mean that they have the same names and domains.)

The undivided attribute values of a tuple of R1 will appear in the result iff the divided attribute values associated with each instance of those undivided attribute values contains the divisor.

Example:

R1	
A	B
a1	b1
a1	b2
a2	b1
a2	b2
a2	b3
a3	b1

R2
B
b1
b2

divide(R1, R2)
A
a1
a2

Further examples:

R1	
A	B
a1	b1
a1	b2
a1	b3
a1	b4
a1	b5
a1	b6
a2	b1
a2	b2
a3	b2
a4	b2
a4	b4
a4	b5

(a)

R2
B
b1

divide(R1, R2)
A
a1
a2

(b)

R2
B
b2
b4

divide(R1, R2)
A
a1
a4

(c)

R2
B
b1
b2
b3
b4
b5
b6

divide(R1, R2)
A
a1

A more concrete example:

ENROLLMENT	
Student_no	Course_code
a1	dim
a1	mom
a2	dim
a3	dim
a3	asm

THEORY_COURSES
Course_code
dim
mom

divide(ENROLLMENT, THEORY_COURSES)
Student_no
a1

(i.e. all students who enrolled for all theory courses —notice how division captures a universal.)

6.2.3 Other

A further two operations are useful when we are writing queries in relational algebra. They are not part of the algebra, but if we were to try to base a practical language on the algebra, we would probably need them.

Assignment

Example:

T1 := product(STUDENTS, COURSES)

Assignment allows us to specify queries as simple steps with results 'stored' in intermediate relations.

Attribute Renaming

Example:

rename(STUDENTS, Dept → Dept_code)

This will be useful for avoiding clashes of attribute names, especially if we join a relation with itself.

6.2.4 Some Examples

Note that query formulation in relational algebra is fairly procedural: you specify a sequence of operations.

In the examples below, we use the following schema, showing students, courses and who takes/took what):

STUDENTS(Student_no, Name, Dept)
COURSES(Course_code, Year)
ENROLLMENT(Student_no, Course_code, Grade)

1. Get student names for students who take dad:

   T1 := nat-join(STUDENTS, ENROLLMENT)
   T2 := restrict(T1, Course_code = 'dad')
   T3 := project(T2, {Name})

   i.e. project(restrict(nat-join(STUDENTS, ENROLLMENT),
   Course_code = 'dad'), {Name})[1]

2. Get student numbers for students who study at least one year 2 course:

[1] For exam answers, I prefer you to present queries using a sequence of assignments to T1, T2, etc., which is easier to understand than this highly nested format.

T1 := restrict(COURSES, Year = 2)
T2 := nat-join(T1, ENROLLMENT)
T3 := project(T2, {Student_no})

Alternatively:

T1 := nat-join(ENROLLMENT, COURSES)
T2 := restrict(T1, Year = 2)
T3 := project(T2, {Student_no})

Generally, the former version is likely to be more efficient than the latter. Why?

3. Get student names for students who study all courses:

T1 := project(ENROLLMENT, {Student_no, Course_code})
T2 := project(COURSES, {Course_code})
T3 := divide(T1, T2)
T4 := nat-join(T3, STUDENTS)
T5 := project(T4, {Name})

4. Get student numbers for students who study at least all those courses taken by student a1:

T1 := project(ENROLLMENT, {Student_no, Course_code})
T2 := restrict(ENROLLMENT, Student_no = 'a1')
T3 := project(T2, {Course_code})
T4 := divide(T1, T3)

5. Get student names for students who do not study sra:

T1 := project(S, {Student_no})
T2 := restrict(ENROLLMENT, Course_code = 'sra')
T3 := project(T2, {Student_no})
T4 := difference(T1, T3)
T5 := nat-join(T4, STUDENTS)
T6 := project(T5, {Name})

6. Get all pairs of student numbers such that the two students are in the same department (avoid duplicate answers):

T1 := rename(STUDENTS, Student_no → AStudent_no,
Dept → ADept)
T2 := rename(STUDENTS, Student_no → BStudent_no,
Dept → BDept)
T3 := product(T1, T2)

T4 := restrict(T3, ADept = BDept $\wedge$ AStudent_no < BStudent_no)
T5 := project(T4, {AStudent_no, BStudent_no})

6.3 Relational Algebra: Discussion

Relational algebra is important theoretically but less important in practical systems: it does not lead to a user-friendly interface. It is hard for naïve and casual users to master due to its procedural flavour: they must specify the steps that say *how* to get the data they want.

Systems that do support the algebra might make it available to programmers but provide something else for casual users. (*Programmers*, who are not naïve users, are usually quite happy with the algebra: for complex queries, the opportunity to break the query down into a sequence of steps is a boon.)

We now draw out a few miscellaneous points.

6.3.1 Updates

Union and difference can be used to insert and delete.

However, if a system does have a relational algebra interface for its users, it is usual to add extra primitives to handle insertion and deletion. This is because union and difference will not give the sort of error messages we would like, e.g. on inserting a duplicate or deleting a non-existent tuple.

6.3.2 Primitives

Note that intersect, join and divide are not primitive operations: they can be defined in terms of the other operations:

- intersect(R1, R2) $\equiv (R1 \cup R2) - (R1 - R2) - (R2 - R1)$
- θ-join $\equiv$ optional project after a restrict (using θ) after a product
- divide(R1, R) where the schemas are R1(A, B) and R2(B) $\equiv$
  difference(project(R1, {A}),
  project(difference(product(project(R1, {A}), R2),
  R1) {A}))

6.3.3 A Limitation

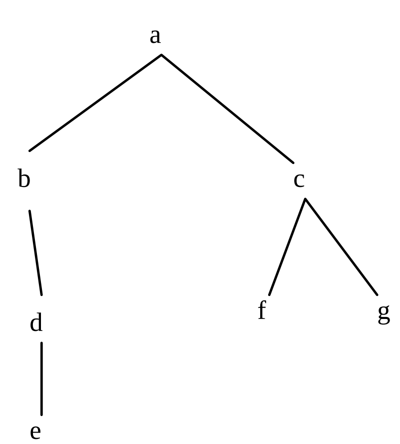
We present now a limitation in the expressiveness of relational algebra. You will, of course, realise that since relational algebra and relational calculus have identical power, then this is a limitation of the calculus, and languages based on the calculus, too. It is a consequence of Codd's notion of a data sublanguage being relationally complete but not computationally complete (no conditionals, no iteration/recursion).

The limitation is: there is no expression of relational algebra (or calculus) that can specify the *transitive closure* of a two- (or more) attribute relation.

Consider the following relation:

R	
Parent	**Child:**
a	b
a	c
b	d
c	f
c	g
d	e

which captures the following tree:



The transitive closure of R, R^*, is:

R*	
Ancestor	**Descendant:**
a	b
a	d
a	e
a	c
a	f
a	g
b	d
b	e
c	f
c	g
d	e

Because the hierarchy can have arbitrary depth, no single query can give you R* from R. (If we know the depth of the tree, d, in advance, we can issue $d-1$ joins.) The problem arises because this is a recursive query and recursion is outside of Codd's sublanguage.

6.4 Exercises

A) An Expanded Version of an EXAM QUESTION: Question 7, Part (i) only, 1990

Sunday Soccer league

Consider the following relations which represent a simple database for a Sunday soccer league (primary key attributes in bold type):

CLUBS(**Club_name:** Town, Points)
PLAYERS(**Club_name, Player_no:** Player_name)
FIXTURES(**Fixture_date, Home_club_name:**
Away_club_name, Home_club_goals, Away_club_goals)

(A) This example has some interesting integrity constraints beyond uniqueness of keys, entity integrity and referential integrity. What might they be?

(B) Write statements in *relational algebra* to perform the following queries.

1. Get the names of all players that belong to the club 'Sudbury Strikers'.
2. Find out who Billy The Fish plays for.
3. Get the names of players who do *not* play for Fulchester Utd.

4. Get the names of players who play for clubs based in Rossdale.

5. Get the names of players who do *not* play for clubs based in Rossdale.

6. Get the towns in which matches were played between 11/09/90 and 03/03/91.

7. Get the dates of all the fixtures that Fulchester Utd. have played.

8. Get the names and points of clubs who have won at least one match.

9. Get the dates of all fixtures for which the two participating clubs are based in the same town.

B) EXAM QUESTION: An expanded version of Question 7, Part (i) only, 1991

Consider the following relations which represent a simple database for the Achin' Hearts Dating Agency (primary key attributes in bold type):

CLIENTS(**Client_no:** Name, Sex, Age)
HOBBIES(**Hobby_no:** Hobby_desc)
CLIENTS_HOBBIES(**Client_no, Hobby_no:**)

Write statements in *relational algebra* to perform the following queries.

(a) Retrieve the ages of all clients at least one of whose hobbies is "Swimming".

(b) Retrieve the names of clients who are not male.

(c) Retrieve the names of clients who do not like hobby number 100.

(d) Retrieve the client numbers and client names of all clients who are recorded as being interested in **all** the hobbies.

(e) Retrieve pairs of client numbers for compatible clients, where 'compatible' is defined as

- the clients are of different sexes,
- the clients' ages are within 5 years of each other, and
- the clients have at least one hobby in common.

(5 marks)

Chapter 7

Structured Query Language — SQL

Structured Query Language, SQL, is the most important relational DML (and DDL) in use today. It is used by numerous commercial systems including R:BASE, Oracle, and DB2. Standards for SQL have been defined by the American National Standards Institute (ANSI) and the International Standards Organisation (ISO). As well as defining standards for SQL itself, these efforts have effectively made SQL *the* standard relational database language. (Most commercial systems diverge from the standard in small ways.)

SQL's popularity means that it is also in increasing use as a language for information interchange between computers. Since there is a version of SQL that will run on almost any hardware/system software configuration, systems exchange data by passing SQL requests and responses to each other. So, not only is it a language in which you can express queries to be evaluated interactively, and not only is it a language that can be embedded into other programming languages such as C, but it is also the language that many off-the-shelf applications software packages use to get data from databases, e.g. the Datalens facility in Lotus 1-2-3 allows an end-user to use Lotus 1-2-3 as normal but will transparently issue SQL commands when data is needed from the DBMS application.

SQL provides both sorts of DML facilities: queries and updates. It also, as we have seen, provides DDL facilities (CREATE, DROP and ALTER). It further provides ways of expressing internal-level and external-level information (see later).

For the most part, this chapter concentrates on the DML querying facilities, which focus on the command SELECT. This turns out to be a 'highly-sugared' syntax for the tuple calculus (although we'll find it easier to explain in terms of the algebra). Since it is based on the calculus, SQL ought to be relationally complete (and most versions are). But despite its broad coverage, it is still simply a data *sub*language: there is no attempt to make SQL computationally complete (there are no conditionals, loops, etc.). To provide conditionals and loops, you must embed SQL into a proper programming language. We will discuss this later. For now, we concentrate on SQL as an interactive, interpreted query language.

7.1 The SELECT statement

SELECT is used for querying. Its basic form is:

```
SELECT target-list
  FROM  table-list
 WHERE  conditions
```

All our examples will be based on the following simple relational database:

S		
SNo	SName	Dept
a1	E.White	CS
a2	E.Brown	EL
a3	M.Green	MA
a4	J.Black	CS

C	
Ccode	Year
itp	1
dad	2
fss	2
nlp	3

SC		
SNo	Ccode	Grade
a1	itp	63
a1	dad	54
a2	dad	70
a3	itp	51
a4	itp	12
a4	nlp	45

To try the examples out, create a database with the tables above, filled with the data above (use the browser to do this), and then go to the R:BASE command line where you can enter the SQL queries presented below.

7.1.1 Basic Queries

(1) Get names of all students

`SELECT SName FROM S`
SName
E.White
E.Brown
M.Green
J.Black

(2) Get all student details

`SELECT * FROM S`		
SNo	SName	Dept
a1	E.White	CS
a2	E.Brown	EL
a3	M.Green	MA
a4	J.Black	CS

(or `SELECT SNo, SName, Dept FROM S`)

(3) Get students departments

`SELECT Dept FROM S`
Dept
CS
EL
MA
CS

To get rid of duplicates, we use SELECT DISTINCT:

`SELECT DISTINCT Dept FROM S`
Dept
CS
EL
MA

(4) Get student numbers for students in Dept 'CS'

SELECT SNo FROM S WHERE Dept = 'CS'
SNo
a1
a4

Notice from the queries below that the answer should always be a table.

(5) Get student numbers for students in Dept 'MA'

SELECT SNo FROM S WHERE Dept = 'MA'
SNo
a3

(6) Get student numbers for students in Dept 'PO'

SELECT SNo FROM S WHERE Dept = 'PO'
SNo

(A table with no records in it.[1])

The results from a query are not guaranteed to be in any particular order. However, the order can be explicitly stated using ORDER BY after the WHERE clause:

```
ORDER BY column-name [ASC | DESC] [, ...]
```

ASC is the default.

(7) Get departments and student numbers in alphabetic order

[1]In fact, R:BASE doesn't return an empty table. Rather, it returns a message:

-WARNING- No rows exist or satisfy the specified clause.

As such, R:BASE breaks the principle of relational closure.

SELECT Dept, SNo FROM S ORDER BY DEPT ASC, SNo ASC	
Dept	SNo
CS	a1
CS	a4
EL	a2
MA	a3

(8) Get year 2 or earlier courses in year order

SELECT Ccode FROM C WHERE Year <= 2 ORDER BY Year ASC, Ccode ASC
Ccode
itp
dad
fss

A query may involve more than one table (to give the equivalent of a join).

(9) Get student names for students who take 'dad'

SELECT SName FROM S, SC WHERE S.SNo = SC.SNo and SC.Ccode = 'dad'
Name
E.White
E.Brown

(10) Get names of students who got $<$ 55 in 'itp'

SELECT SName FROM S, SC WHERE S.SNo = SC.SNo and SC.Ccode = 'itp' and Grade < 55
SName
M.Green
J.Black

(11) Get names of students who take courses offered in years > 1

```
SELECT SName FROM S, C, SC
WHERE S.SNo = SC.SNo and
SC.Ccode = C.Ccode and
C.Year > 1
```

SName
E.White
E.Brown
J.Black

(12) Get departments and grades < 55 where a student of the department got that grade in a course

```
SELECT Dept, Grade FROM S, SC
WHERE S.SNo = SC.SNo and
Grade < 55
ORDER BY Dept ASC, Grade ASC
```

Dept	Grade
CS	12
CS	45
CS	54
MA	51

For some queries we need to label two different occurrences of one table in the same query (i.e. this is an example of joining a table with itself). This is done by giving the table what are variously referred to as aliases, nicknames, correlation names or labels:

(13) Get all pairs of student numbers such that the two students are in the same department (don't repeat pairs)

```
SELECT S1.SNo, S2.SNo
FROM S S1, S S2
WHERE S1.Dept = S2.Dept AND
S1.SNo < S2.SNo
```

S1.SNo	S2.SNo
a1	a4

7.1.2 Built-in Functions

SQL also provides some built-in functions: COUNT, SUM, AVG, MAX, MIN, GROUP BY, HAVING and UNION. (For SUM and AVG the column must hold numeric values).

(14) How many students are there?

```
SELECT COUNT(*) FROM S
```

COUNT(*)
4

(15) How many different departments are students in?

```
SELECT COUNT(DISTINCT Dept) FROM S
```

COUNT(DISTINCT)
3

(16) How many students take 'itp'?

```
SELECT COUNT(*) FROM SC
WHERE SC.Ccode = 'itp'
```

COUNT(*)
3

(17) What is student a1's total mark?

```
SELECT SUM(Grade) FROM SC
WHERE SNo = 'a1'
```

SUM(Grade)
117

(18) What is student a1's average mark?

SELECT AVG(Grade) FROM SC WHERE SNo = 'a1'
AVG(Grade)
58

(Averages of integers are rounded.)

If we wanted to find the average mark of *each* student, we would have to collect together the entries in SC by SNo and average each *group* of entries.

The GROUP BY operator allows the definition of 'partitions' that permit calculation of averages, etc. on the partitions. (The data is also sorted automatically without you asking for this explicitly.)

(19) What is each student's average mark?

SELECT SNo, AVG(Grade) FROM SC GROUP BY SNo	
SNo	AVG(Grade)
a1	58
a2	70
a3	51
a4	28

(20) What are the average marks on year ≥ 2 courses?

SELECT SC.Ccode, AVG(Grade) FROM SC, C WHERE C.Year >= 2 AND C.Ccode = SC.Ccode GROUP BY SC.Ccode	
SC.Ccode	AVG(Grade)
dad	62
nlp	45

To refine after grouping, the HAVING clause allows comparison of some property of a group (COUNT, SUM, AVG, etc.) with a value. (Thus, HAVING is like WHERE but is applied to groups.)

(21) Get the course codes of all courses taken by $>$ 1 student

SELECT Ccode FROM SC GROUP BY Ccode HAVING COUNT(*) > 1
Ccode
dad
itp

(22) Get each student's average mark where the average < 55

SELECT SNo, AVG(Grade) FROM SC GROUP BY SNo HAVING AVG(Grade) < 55	
SNo	AVG(Grade)
a3	51
a4	28

UNION allows the results of two SELECTs to be put together.

(23) Get all CS students and all MA students

SELECT * FROM S WHERE Dept = 'CS' UNION SELECT * FROM S WHERE Dept = 'MA'		
SNo	SName	Dept
a1	E.White	CS
a4	J.Black	CS
a3	M.Green	MA

Obviously, this example is somewhat artificial since the following does the job just as well:

```
SELECT * FROM S WHERE Dept = 'CS' OR Dept = 'MA'
```

Notes: (1) By default, UNION removes duplicates. This can be inhibited by including the word ALL after UNION. (2) You cannot UNION together *nested* SELECTs (see below). (3) The tables must have an equal number of columns and their columns must have pairwise identical data types.

7.1.3 Nested Queries

An alternative way of writing queries that have 'joins' in them is to *nest* a sub-query within another. For example, 'Get student names for students who take 'dad' ' was earlier formulated with multiple entries in the FROM clause and a complex WHERE clause. Here we use nesting:

```
SELECT SName FROM S
 WHERE SNo IN
   (SELECT SNo FROM SC
     WHERE SC.Ccode = 'dad')
```

The nested query is evaluated first, to supply a set of values for use in the outer query.

Here's another example. 'Get names of students who take courses offered in years > 1'. Using nesting:

```
SELECT SName FROM S
 WHERE SNo IN
   (SELECT SNo FROM SC
     WHERE SC.Ccode IN
       (SELECT C.Ccode FROM C
         WHERE C.Year > 1))
```

Note: there are restrictions on the form of nested SELECTs: they can include only one column name between the keywords SELECT and FROM, and they may not include an ORDER BY clause.

Most nested queries can be 'de-nested' (Obviously, not all queries can be nested.) 'De-nesting' a nested query can often improve performance (see query optimisation) and can be done automatically. This leaves the user with a choice of how s/he writes her/his query. As queries become more complex, nesting is preferred by some people as it allows them to break the problem down into sub-problems. I tend to avoid nesting as much as possible. This is a matter of personal preference because, for me, nesting is less comfortable. However, there can be quite sophisticated examples in which nesting cannot be avoided, particularly where instead of IN we use some other operators. Examples follow.

EXISTS is true if there are rows that satisfy the nested SELECT statement.

(24) Get the student numbers for students enrolled on more than one course

SELECT DISTINCT SNo FROM SC A WHERE EXISTS (SELECT * FROM SC B WHERE A.SNo = B.SNo AND A.Ccode <> B.Ccode)
SNo
a1
a4

(We needed the aliases, A and B, to distinguish two uses of the SC table.) The idea in the above is to return the student numbers for students in the SC table for whom there exists two or more rows in the SC table with that student number but different course codes.

NOT EXISTS is true if there are no rows that satisfy the nested SELECT statement.

(25) Get the names of the students who have taken all the courses, i.e. get the names of those students for whom there are no courses that they didn't take

SELECT SName FROM S WHERE NOT EXISTS (SELECT * FROM C WHERE NOT EXISTS (SELECT * FROM SC WHERE C.Ccode = SC.Ccode AND SC.SNo = S.SNo))
SName

(No answers in this case.) The way this works is that the most nested SELECT finds the courses the student did take. The middle SELECT determines for each student whether any courses were not taken. If not, the outer SELECT then returns the name for that student.

(26) Get student numbers for students who are in the same department as a1

```
SELECT SNo FROM S
WHERE Dept =
(SELECT Dept FROM S
WHERE SNo = 'a1')
```

SNo
a1
a4

(27) Get student numbers for students who have at least one grade which is less than the overall average grade

```
SELECT DISTINCT SNo FROM SC
WHERE Grade <
(SELECT AVG(Grade) FROM SC)
```

SNo
a4

(28) Get all CS students' numbers if they have a higher mark than some Maths student

```
SELECT DISTINCT SC1.SNo FROM SC SC1, S S1
WHERE S1.SNo = SC1.SNo AND
S1.Dept = 'CS' AND
SC1.Grade > SOME
(SELECT SC2.Grade FROM SC SC2, S S2
WHERE SC2.SNo = S2.SNo AND
S2.Dept = 'MA')
```

SC1.SNo
a1

You are allowed < SOME, <= SOME, >= SOME, = SOME and <> SOME. = SOME is identical to IN. The keyword ANY is synonymous to SOME. You are also allowed ALL.

(29) Get all CS students' numbers if they have a higher mark than all Maths student

```
SELECT DISTINCT SC1.SNo FROM SC SC1, S S1
       WHERE S1.SNo = SC1.SNo AND
           S1.Dept = 'CS' AND
            SC1.Grade > ALL
   (SELECT SC2.Grade FROM SC SC2, S S2
       WHERE SC2.SNo = S2.SNo AND
            S2.Dept = 'MA')
```

SC1.SNo
a1

7.1.4 Summary of the SELECT Command

The SELECT command obviously has a gloriously complicated syntax. Here's an attempt to summarise it.

```
SELECT [DISTINCT] {* | {colname | function} [, ...]}
  FROM table-name [alias-label] [, ...]
[WHERE clause]
{ [ORDER BY colname [ASC | DESC] [, ...] ] |
  [GROUP BY colname [ASC | DESC] [, ...] [HAVING clause] }
```

and

```
SELECT-statement UNION [ALL] SELECT-statement
```

Functions in the SELECT clause (and in the HAVING clause below) are of the form:

```
{{COUNT | SUM | AVG | MIN | MAX} ( [DISTINCT] colname ) | COUNT(*) }
```

The WHERE and HAVING clauses may contain:

- comparison operators $(=, <>, >, >=, <, <=)$ where the first operand can be a column name or a value and the second operand can be a column name, a value or a nested SELECT statement enclosed in parentheses. Additionally, for HAVING clauses only, the operands can be functions.
- boolean operators: `AND OR NOT`
- the EXISTS function: `EXISTS (SELECT-statement)`

- ALL, ANY and SOME: {colname | value} comparison-op {ALL | ANY | SOME} (SELECT-statement). Again, in HAVING clauses you can use a function rather than a column name.

- the BETWEEN operator: colname [NOT] BETWEEN {colname | value} AND {colname | value}

- pattern matching: column [NOT] LIKE pattern, where pattern is a string which may contain

  - % —meaning any sequence of zero or more characters
  - _ —meaning any single character

  e.g. SName LIKE '_.BR%' matches 'P.Brown', 'J.Browning', etc.

It's important that you get to grips with the SELECT command. My advice is that you answer the exercise at the end of the chapter, typing queries in at the R:BASE command line. An alternative, and I don't know how effective it will be, is to use R:BASE's Prompt-By-Example facility. Go to the R:BASE command line, enter prompt. Select (6) on the menu. You now have a facility which allows you to enter SQL commands but which guides you at each step about what comes next so that it is impossible to make a syntax error.

7.2 Update in SQL

Querying is one facet of a DML. The other facet of a DML is update (insertion, modification and deletion).

7.2.1 Insertion of data

```
INSERT
  INTO table-name [(column [, column] [, ...])]
  VALUES (constant [, constant] [, ...])
```

or

```
INSERT
  INTO table-name [(column [, column] [, ...])]
  SELECT ... FROM ... WHERE ...
```

Here are some simple examples:

- To insert a new SC record for a2 and course nlp with Grade unknown:

  ```
  INSERT INTO SC (SNo, Ccode)
    VALUES ('a2', 'nlp')
  ```

  (Note that Grade will be null. In R:BASE this is -0-. In some systems, it is blank.)

- To insert a new student, a5, name of R.Silver and department maths:

  ```
  INSERT INTO S VALUES ('a5', 'R.Silver', 'MA')
  ```

  Note that missing out the list of column names (as in this example) is equivalent to specifying all of them. The order of values must then be the same as when the table was created.

- Let's create a new table, MARKS:

  ```
  CREATE TABLE MARKS (SNo TEXT 2, Mark INTEGER)
  ```

  For each student, to put his/her average mark into MARKS:

  ```
  INSERT INTO MARKS (SNo, Mark)
    SELECT SNo, AVG(Grade)
      FROM SC GROUP BY SNo
  ```

  This executes the SELECT to return a table which is plopped into MARKS.

7.2.2 Modification of data

```
UPDATE table-name
  SET column = expression [, column = expression] [, ...]
  [WHERE conditions]
```

Here are some examples:

- To change the year of dad to 1:

  ```
  UPDATE C
    SET  Year = 1
    WHERE Ccode = 'dad'
  ```

- To double all marks for dad:

```
UPDATE SC
  SET Grade = 2*Grade
  WHERE Ccode = 'dad'
```

- To change the code for dad to dbt:

```
UPDATE C SET Ccode = 'dbt' WHERE Ccode = 'dad'

UPDATE SC SET Ccode = 'dbt' WHERE Ccode = 'dad'
```

  Note that we need *two* separate operations. It is not possible to modify more than one table in a single operation.

  Note that even if you have defined a referential integrity constraint from SC.Ccode to C.Ccode, R:BASE cheerfully changes dad to dbt in the C table if you ask for this to be done first, even though this leaves SC rows with dad in them, thus violating the constraint. (If you issue the commands in the other order, R:BASE does complain.)

7.2.3 Deletion of data

```
DELETE FROM table-name [WHERE conditions]
```

Here are some simple examples:

- To delete student a4:

```
DELETE FROM S WHERE SNo = 'a4'
```

  Again, even if you've defined a referential integrity constraint from SC.SNo to S.SNo, R:BASE will not check whether your deletion from S leaves entries in SC that violate the constraint.

- To delete all SC records for a4:

```
DELETE FROM SC WHERE SNo = 'a4'
```

- To delete all SC records:

```
DELETE FROM SC
```

7.3 Discussion of SQL

7.3.1 Completeness of SQL

To see that SQL is (in at least some implementations) relationally complete, we demonstrate (informally) an equivalence with relational algebra (which we already know to be complete).

For the relations R1(A), R2(A), R3(B), R4(A, B) R5(C, D) and R6(B).

- product(R1, R3) ≡ SELECT * FROM R1, R3
- union(R1, R2) ≡ SELECT * FROM R1 UNION SELECT * FROM R2
- difference(R1, R2) ≡ SELECT * FROM R1 WHERE A NOT IN (SELECT A FROM R2)
- intersect(R1, R2) ≡ SELECT * FROM R1 WHERE A IN (SELECT A FROM R2)
- restrict(R1, A = 'a') ≡ SELECT * FROM R1 WHERE A = 'a'
- project(R4, {A}) ≡ SELECT DISTINCT A FROM R4
- nat-join(R1, R5, B=D) ≡ SELECT A, B, C FROM R1, R5 WHERE B = D
- divide(R4, R6) ≡ SELECT DISTINCT A FROM R4 ALIAS1 WHERE NOT EXISTS (SELECT B FROM R6 WHERE NOT EXISTS (SELECT A FROM R4 ALIAS2 WHERE ALIAS2.B = R6.B AND ALIAS2.A = ALIAS1.A))

7.3.2 The Pros and Cons of SQL

The Pros of SQL

- SQL supports a major proportion of the relational data model as expounded by Codd.
- It combines all the standard DDL operations (table creation, alteration and dropping) and the standard DML operations (querying, insertion, modification, deletion) in one uniform syntax. (It also allows some external and internal level stuff —see subsequent chapters.)
- It is based on the theory but is still fairly easy to use.

- It can be used as a standalone query language (interactive) or embedded within a conventional programming language (see subsequent chapters).
- It is (in at least some implementations) relationally complete.
- It supports an amount of data independence. To see this, you must wait for us to tackle the external and internal levels.

The Cons of SQL

- Missing relational features
  - It has no *direct* means of declaring the primary key of a relation. The approach in R:BASE (which is better than most systems) is cumbersome.
  - By default, it allows duplicate rows in a table.
  - It has no *direct* means for declaring foreign keys. Again, R:BASE is better than most but is still cumbersome (and look at Sections 7.2.2 and 7.2.3 to see how inadequate R:BASE's treatment of referential integrity is).
  - In the SQL standard there is no means of allowing user-defined integrity constraints. (Again R:BASE scores moderately well.)
  - The notion of domains is weak.
- Lack of coherency: this relates to the quirkiness of the language; different implementations have different quirks. Here are some that can be found:
  - There is no user-friendly way to store temporary results of queries. Nested queries which were supposed to overcome this need are cumbersome.
  - Some relational algebra operators are difficult to achieve in some systems, e.g. division and, in some systems, union.
  - Null values are treated in different ways by the in-built functions: they are ignored by SUM, AVG and COUNT(column) but taken into account by COUNT(*). (This can mean that AVG(column) = SUM(column) $\div$ COUNT(column) $\neq$ SUM(column) $\div$ COUNT(*).)
  - A SELECT clause cannot contain the keyword DISTINCT more than once at any level.

- Redundancy (Is this a pro or a con?) There is usually more than one way of getting a job done in SQL. e.g. Get names of students who study ‘dad’:

```
SELECT SName FROM S WHERE SNo IN
   (SELECT SNo FROM SC WHERE Ccode = 'dad')

SELECT SName FROM S WHERE SNo = ANY
   (SELECT SNo FROM SC WHERE Ccode = 'dad')

SELECT SNAME FROM S WHERE EXISTS
   (SELECT * FROM SC WHERE SC.SNo = S.SNo AND SC.Ccode = 'dad')

SELECT DISTINCT SName FROM S, SC
   WHERE S.SNo = SC.SNo AND SC.Ccode = 'dad'

SELECT SName FROM S WHERE 0 <
   (SELECT COUNT(*) FROM SC
     WHERE S.SNo = SC.SNo AND SC.Ccode = 'dad')

SELECT SName FROM S WHERE 'dad' IN
   (SELECT Ccode FROM SC WHERE S.SNo = SC.SNo)

SELECT SName FROM S WHERE 'dad' = ANY
   (SELECT Ccode FROM SC WHERE S.SNo = SC.SNo)
```

Some praise this ‘feature’. Others, notably Chris Date, criticise it. (The rationale behind the criticism is that this redundancy, like the quirkiness, is symptomatic of appallingly poor language design, e.g. it’s really rather unacceptable that UNION cannot be used in nested SELECTs: what possible reason can there be for such an artificial restriction? Its cause is the mish-mashy way that the language is developing.)

7.4 Query-By-Example — QBE

While SQL is based on the relational *tuple* calculus, QBE (Query-By-Example) is based on the relational *domain* calculus.

QBE is a very pictorial style of query language. You request that some templates of the database tables be displayed on your screen and then you enter special characters into the columns of these templates to exemplify the nature of the returned data.

Different versions of QBE operate in slightly different ways. This shows what your screen might show in R:BASE once you have specified the query to ‘Get student names for students who study at least one year 2 course’:

T_1.S	SNo	$\surd$SName	Dept
	<link-1>		

T_2.C	Ccode	Year
	<link-2>	= 2

T_3.SC	SNo	Ccode	Grade
	<link-1>	<link-2>	

Ticks ($\surd$) show which columns are wanted in the result. Links define columns on which joins are being done. And conditions can be entered into columns, as with C.Year.

You’ll get to play with QBE at the end of the next chapter.

7.5 Exercise

A) NOT AN EXAM QUESTION: The Films Database

We have set up a database containing information about films. To access it, copy the following files into your own directory:

```
vol1:\pub\rbfiles\filmdb1.rbf
vol1:\pub\rbfiles\filmdb2.rbf
vol1:\pub\rbfiles\filmdb3.rbf
```

The schema, which you can see by typing `list tables` at the R:BASE command line, is:

```
Actors(title, act_surname, act_firstname)
Films(title, dir_surname, country, year_made, film_length)
Directors(dir_surname, dir_firstname, dir_yob, dir_bplace)
```

Write statements in *SQL* to perform the following queries and test them on the films database:

1. Films directed by Hitchcock

2. Films made since 1970
3. The film called A MAN FOR ALL SEASONS
4. All directors born in the USA
5. All directors born between 1940 and 1950
6. All directors born in 1904, 1906 and 1908
7. Years in which a film was made
8. All directors' first names
9. Titles of films and corresponding years
10. Films and the actors in them and years made
11. Films whose directors were born in England
12. Directors of films made in the 1930s
13. Birthplaces of directors who directed films starring Woody Allen
14. Titles of films directed by directors born in Japan
15. Actors who appear in films directed by Minnelli
16. Actors who appear in films they directed themselves
17. Lengths of films made in England during the 1950s
18. Titles of films directed by American-born directors which exceed one-and-a-half hours in length
19. Birthplaces of directors associated with films of less than 1.5 hours duration which were made in the USA and do not star Hitchcock
20. Average film length
21. Maximum film length and minimum film length
22. Number of films made in each year
23. The oldest film
24. The number of films directed by Hitchcock
25. Film lengths shared by > 2 films
26. Pairs of directors born in the same town

Questions similar to the more difficult ones in the list above have appeared in exam papers.

B) A question similar to exam question 9, part (iii), 1993

The films database was designed (if that's not too strong a word) by one of my predecessors. Given what we know about films, and what we might want to store in such a database, criticise the schema. (One way to do this would be to draw an ER diagram, map it to a logical model, and compare what you come up with to the existing schema.)

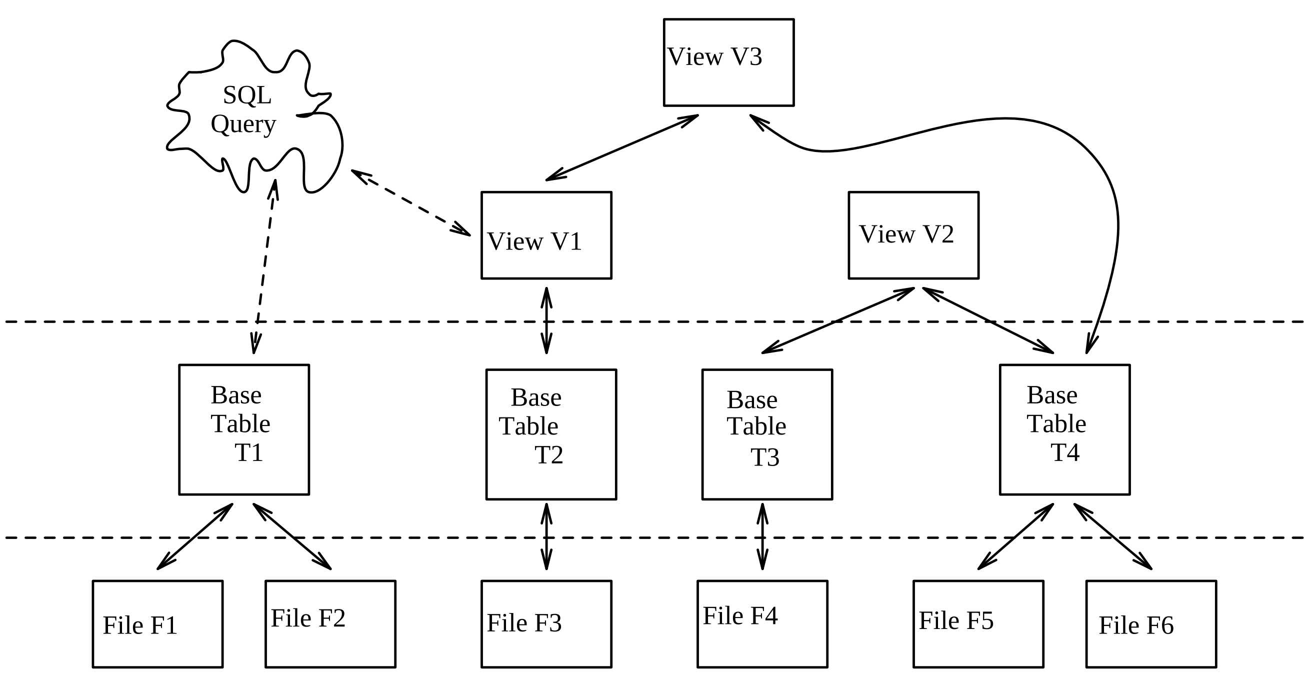
Chapter 8

The External Level

So far we have looked at DDL and DML operations which operate at the conceptual level. Now we turn to specifying external level information. In relational DBMSs the external level is not done by a set of external schemas (as was the case in the ANSI/SPARC architecture). Rather, we use SQL to specify *views*.

8.1 Views

Consider the following characterisation of a database system such as R:BASE:



The query is formulated in terms of data from base table T1 and view V1.

The database consists of three types of object:

- Files (which hold actual data)
- Base tables (the conceptual perspective on the actual data)
- Views (derived tables)

So far we have been looking at *base tables*. That is, tables of data occupying a particular amount of physical storage. The more rows in the base table, the more storage is used.

In SQL, a *view* is a table that is visible to the user, but is not physically stored. It is a *derived* or *virtual* table. For data manipulation, a view (*ideally*) behaves exactly like a base table: it can be used in a SELECT statement in the same way as a base table.

To illustrate with a simple example. Suppose we want a table of CS students. We can write a query for this:

```
SELECT SNo, SName FROM S WHERE Dept = 'CS'
```

If this was a query we performed often, we could create a view on this data: this is a way of giving a name to this query. Here's how in SQL:

```
CREATE VIEW CS_STUDENTS AS
  SELECT SNo, SName FROM S WHERE Dept = 'CS'
```

For retrieval, we can now use the view as if it were a base table, e.g.:

```
SELECT * FROM CS_STUDENTS

SELECT SName FROM CS_STUDENTS

SELECT SName FROM CS_STUDENTS WHERE SNo = 'a1'

SELECT SName FROM CS_STUDENTS, SC
  WHERE CS_STUDENTS.SNo = SC.SNo AND
               SC.Ccode = 'itp'
```

Note that the use of the view makes the queries less complicated and easier to read.[1]

A view can be defined by giving a name to any SQL query (or any QBE query). (Indeed, views are pretty much first-class citizens. Wherever we have used a table up to now, you can use a view instead, e.g. you can use views as well as ordinary tables in QBE.) At least, that is so in theory. R:BASE and other systems have a number of restrictions -see later. Here are some more complex views.

The first involves more than one table and captures all students who study second year courses:

```
CREATE VIEW YEAR_2_STUDENTS AS
  SELECT SNo, SName, Dept FROM S, C, SC
    WHERE S.SNo = SC.SNo AND
       SC.Ccode = C.Ccode AND
         C.Year = 2
```

The second shows that a view can use other views in its definition. Given that we defined a view for CS_STUDENTS earlier, this one is for all CS students who got < 55 in a course:

```
CREATE VIEW RESIT_CS_STUDENTS AS
  SELECT SNo, SName FROM CS_STUDENTS, SC
    WHERE CS_STUDENTS.SNo = SC.SNo AND
                 SC.Grade < 55
```

8.2 How Views Work

The definition of a view is stored in the data dictionary. In R:BASE, it is stored in the data dictionary table called SYSVIEWS, which you can query in the same way as you queried SYSINFO and SYSRULES. Here's an example entry:

[1]Views can be removed from the system using:

```
DROP VIEW view-name
```

SYSVIEWS	
SYSVNAME	SYSVTEXT
CS_STUDENTS	SELECT SNo, SName FROM S WHERE Dept = 'CS'

Whenever an operation is specified which uses a view, the definition is found and used in the operation. For example:

```
SELECT SName FROM CS_STUDENTS
```

after consultation of the data dictionary, is converted by the DBMS into:

```
SELECT SName FROM S
  WHERE Dept = 'CS'
```

This new query is then executed as normal.

This can be carried out for all retrieval operations. The view acts as a 'dynamic window' on the underlying base tables. On each retrieval, the view definition is found and executed. (An analogy is macros.)

I repeat the crucial point about views: we are giving a name to the query (as you can see by the entry in SYSVIEWS); we are not giving a name to the answer.

Try out the following for the database we used in the last chapter (the one with tables S, C and SC):

1. Create a view:

   ```
   CREATE VIEW CS_STUDENTS AS
     SELECT SNo, SName FROM S WHERE Dept = 'CS'
   ```

2. Look at its 'contents':

   ```
   SELECT * FROM CS_STUDENTS
   ```

3. Now add a tuple to the S table:

   ```
   INSERT INTO S VALUES ('a5', 'R.Silver', 'PO')
   ```

4. Look at the 'contents' of the view again:

   ```
   SELECT * FROM CS_STUDENTS
   ```

The above should convince you that views do not name answers, they name queries. And, as such, they provide a dynamic window onto the database's data.

8.3 Updatability

Data retrieval on views can take place for all possible views, using the automatic conversion process ('macro expansion'). However, *not all views are updatable.*

Here is an example that *is*:

```
CREATE VIEW V1 AS
  SELECT SNo, Dept FROM S
```

(assume SNo is defined as NOT NULL).

Now consider an insertion into V1:

```
INSERT INTO V1
       VALUES ('a6', 'MA')
```

This is converted into an equivalent operation on S:

```
INSERT INTO S
       VALUES ('a6', -0-, 'MA')
```

(where -0- is the null symbol in R:BASE).

(Remember: the reason this gets converted into an insertion into S is because it is base tables that occupy physical storage and are therefore affected by update operations. The insertion affects the view only indirectly.)

Here is an example that is *not* updatable:

```
CREATE VIEW V2 AS
  SELECT Dept FROM S
```

Now consider an insertion into V2:

```
INSERT INTO V2
       VALUES ('MA')
```

This is converted into the equivalent:

```
INSERT INTO S
       VALUES (-0-, -0-, 'MA')
```

This will *fail* because SNo is defined as NOT NULL.

The example with V2 shows that we have a problem in deciding on an *equivalent* translation from view update to base table update.

Things get worse if we have more complex views, e.g.:

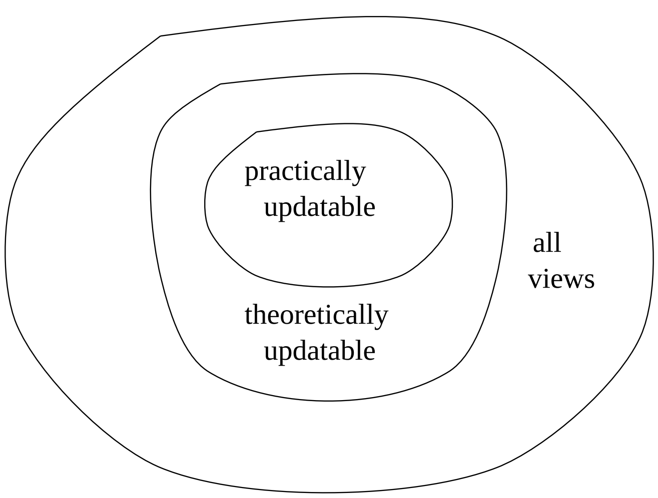
```
CREATE VIEW V3 AS
  SELECT SNo, SName FROM S, SC
    WHERE S.SNo = SC.SNo AND
       SC.Grade < 55
```

What does it mean to insert a record into V3? We can easily translate this into an insertion into S with a null Dept. But do we create an SC record? If so, what values do we give it for C.Ccode and Grade knowing only that the grade < 55. Do we create only one SC record or several?

There are thus some views which are *inherently* updatable, and other which are *inherently* not updatable. (This, of course, has the unfortunate consequence that views no longer behave just like normal tables: some views are updatable, others are not.)

The problem is that theoreticians do not know the exact conditions that make a view updatable.

In commercial systems, the designers devise a set of simple rules which say which views are updatable. However, often the rules they devise are too restrictive. This means that some views which are theoretically updatable cannot be updated in that DBMS:



The R:BASE manual, in line with the current ISO SQL Standard, says: "A view is updatable when it does not contain the UNION operator, and its SELECT clause:

- Does not specify DISTINCT.
- Includes columns from one table only.
- Does not include sub-SELECT in the WHERE clause.
- Does not include the GROUP BY or HAVING clause."

They forget to mention also that for a view to be updatable there can be no functions (MAX, COUNT, etc.) in the columns associated with the view.

This is *very* restrictive. It basically only allows updates to views which are row or column subsets of a *single* base table.

(Don't be confused: you can *create* views which violate the above rules and you can cheerfully use these views for *querying*; it's just that you *cannot* perform *update* operations on views that violate the rules.)

8.4 Why Are Views Useful?

1. They provide an amount of logical data independence when the database is restructured in some way.

   Example. Suppose it was found necessary to split the student table (S) into two (maybe one part is stored at one node of a network and the other part at a different node):

   S_PartA(SNo, SName)
   S_PartB(SNo, Dept)

   It looks as though existing applications would need to be rewritten as S no longer exists. However, we could instead define a view S which recreates the full table and manipulation of S can be as before (except for update restrictions):

   ```
   CREATE VIEW S (SNo, SName, Dept) AS
     SELECT S_PartA.SNo, S_PartA.SName, S_PartB.Dept
       FROM S_PartA, S_PartB
      WHERE S_PartA.SNo = S_PartB.SNo
   ```

2. They allow the same data to be seen differently by different users, e.g. a statistician might be interested only in a student's average grade. A view can be defined for this.

3. The user's perception is simplified, e.g. the statistician need not write a complex query to get his/her data so s/he does not need to know that we actually store grades for each course and not average grades.

4. An amount of automatic security is provided since some data is hidden. Of course for this to work, we have to be able to make sure people can only access data through views which we define for them and by no other route. More on this perhaps later in the course.

You have seen how to create views using SQL commands, and you ought to have tried this out using R:BASE. Views can also be created using R:BASE's menu system. I suggest that now is the time to use the R:BASE tutorial again. Work through lesson 3 (Viewing Data in Tables), which re-explains the difference between queries and views quite nicely and shows you some of the query facilities of the browser, and lesson 6 (Creating and Saving Queries), which takes you through some QBE as well as creating a view.

8.5 Exercise

EXAM QUESTION: Question 4, 1994

Alice Brownloaf runs a Sudbury newsagents. She uses a database to organise the paper rounds. The database has relations to hold details of the delivery boys and girls (DELIVERERS), newspapers (NEWSPAPERS, where NEWSPAPERS.Type is either 'Tabloid' or 'Broadsheet') and customers (CUSTOMERS). The SUBSCRIPTION relation says which newspapers a customer takes, and the CUST-DELIVERER relation says which deliverer is assigned to which customer.

DELIVERERS(**PersonId:** PName)
NEWSPAPERS(**PaperName:** Price, Type)
CUSTOMERS(**CustId:** CName, Address)
SUBSCRIPTIONS(**CustId, PaperName:**)
CUST-DELIVERERS(**PersonId, CustId:**)

(i) [5 marks] State in English what the following SQL queries would retrieve. (Make your translations as natural as possible.)

(a) (2 marks)

```
select PersonId, count(Type)
  from NEWSPAPERS, SUBSCRIPTIONS, CUST-DELIVERERS
 where NEWSPAPERS.PaperName = SUBSCRIPTIONS.PaperName
```

```
     and SUBSCRIPTIONS.CustId = CUST-DELIVERERS.CustId
group by PersonId, Type
```

(b) (3 marks)

```
select CustId from CUSTOMERS
 where not exists
       (select PaperName from NEWSPAPERS
         where Type = 'Tabloid'
           and not exists
               (select PaperName from SUBSCRIPTIONS
                 where NEWSPAPERS.PaperName = SUBSCRIPTIONS.PaperName
                   and CUSTOMERS.CustId = SUBSCRIPTIONS.CustId))
```

(ii) [9 marks] Write statements in *SQL* to perform the following queries:

(a) Retrieve the identifiers and names of customers who subscribe to 'The Tory Rot' and live on the route of delivery person number 3.

(b) Retrieve customer identifiers of customers who do not take any tabloids.

(c) Retrieve pairs of newspaper names for newspapers that cost the same.

(iii) [3 marks] Explain what actions the DBMS would carry out for each of the following instructions, assuming that they are executed in the sequence given below.

(a) The user creates a view called TABLOIDS as follows:

```
create view TABLOIDS as
   select * from NEWPSPAPERS
    where Type = 'Tabloid'
```

(b) The user issues the following query:

```
select * from TABLOIDS
```

(c) The user asks for a tuple to be inserted into TABLOIDS:

```
insert into TABLOIDS
       values ('The Daily Sleaze', '22p', 'Tabloid')
```

(iv) [3 marks] According to the ISO SQL standard, views that are defined in terms of the join of two tables are not updatable. This is a too rigid constraint: some such views are theoretically updatable. A necessary but not sufficient condition is that if two attributes are equated in the join, only one of the pair appears as an attribute of the view. Explain why this is a necessary condition on the updatability of views defined in terms of joins.

Chapter 9

The Internal Level

One of our main requirements is the ability to store large amounts of data. The physical data is stored on a secondary-storage device, *not* in main memory. Access times to secondary-storage (tape, drum or disk) are much slower than main memory access times and CPU instruction execution times. So our overriding performance objective is:

> ********MINIMISE THE NUMBER OF********
> *****SECONDARY-STORAGE ACCESSES*****

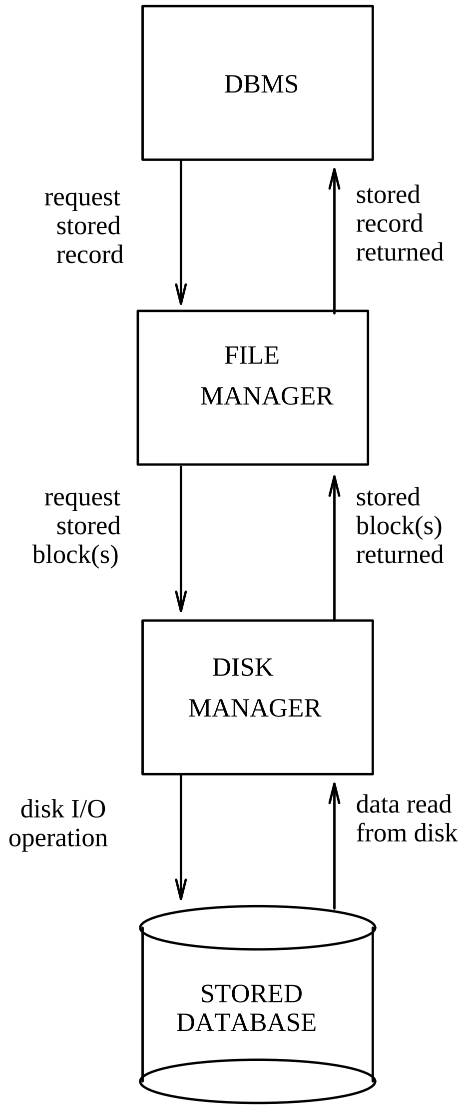
More specifically, we must balance the possibly conflicting demands of our applications:

- fast retrieval versus convenient update;
- fast manipulation versus economy of storage.

We attempt to satisfy these demands by imposing an organisation or *storage structure* on the data. No single structure will be optimal for all applications because of the trade-offs we have to make.

Practical database storage structures are only possible on direct access storage devices (such as disks). Henceforth, we will assume the use of disks.

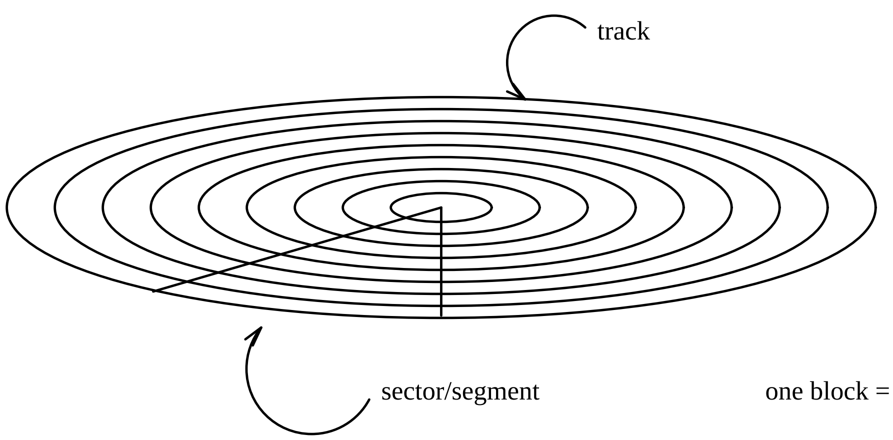
9.1 Database Access



- The *DBMS* decides which *stored record* is required.
- The *file manager* decides which *block* (or blocks) is required. The file manager:
  - can be part of the operating system or a special-purpose part of the DBMS;
  - considers the disk as a collection of stored files;
  - one file occupies one or more blocks;
  - each file has a unique identifier;

  - each stored record has a unique identifier within the file;
  - maps record requests to block requests.

- The *disk manager* reads that block (or those blocks) requested by the file manager from the correct part of the disk, The disk manager:
  - is part of the operating system, responsible for disk I/O;
  - considers the disk as divided into blocks of data;
  - blocks are identified by means of a physical address (sector within track);
  - controls all device-specific information.



one block = 1 or more sectors

We are going to look at the storage structures used by the *file manager*.

9.2 Storage Structures

We will briefly consider serial files (piles/heaps), sequential files, hashed files, indexed files, indexed-sequential files, inverted files, multi-ring files, and multi-index files (B+trees). Note that a file can have more than one organisation.

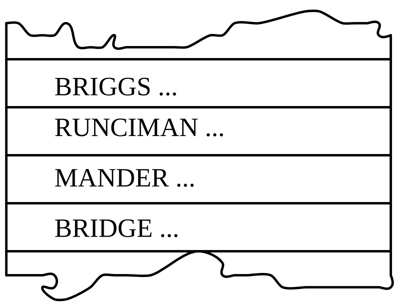
One should assess each organisation on the basis of storage overheads, time to fetch an arbitrary record, time to get the next record in a sequence, time to insert a new record, time to delete a record[1], and the reorganisation needs of the structure.

(Note: we assume fixed-length records. Insertion and modification of records can become more complicated if records are variable-length.)

[1]We'll assume that a modification of a record is not interesting because modification is like a deletion followed by an insertion.

9.3 Serial Files

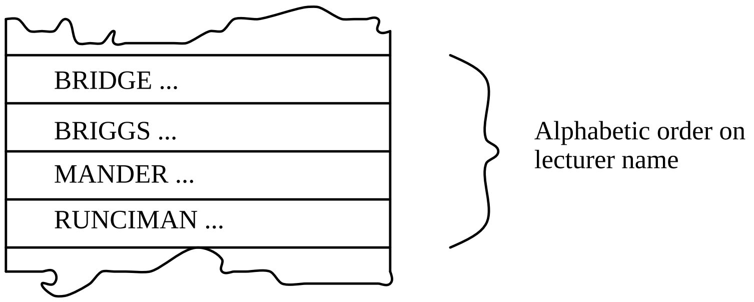
(Also known as heap or pile organisation.) This is really no organisation at all. It implies no particular storage sequence (none which can be relied on). It is common in relational databases (since relation extensions are sets, this implies no ordering) and is used for transaction files in batch processing systems.



This organisation has no extra storage overheads; needs possibly exhaustive searches for an arbitrary record; needs sorting to get records in sequence; has quick insertion (keep an end of file pointer and append new records); deletion requires serial search and then overwrite with a *tombstone*; may need regular compaction to get rid of tombstones.

9.4 Sequential Files

The physical order of the records is maintained according to the sequence of some access key field(s), e.g. alphabetic order.



There are no extra storage overheads; to find an arbitrary record, you can use (possibly exhaustive) serial search, binary search (binary chop) or 'probing' (where you make a guess and then serial search forwards or backwards); to find the next record in a sequence, assuming the file sequence =

the search sequence, then simply look at immediate successors; insertion into the proper place is difficult: you must either shuffle records about (which is time-consuming) or pend them in a transaction file, sort the transaction file and do a merge (this is batch-processing and means that the file is out of date until the next merge and means that to search for a record you need to search both the main and transaction files); deletion is search as mentioned above followed by replacement by a tombstone; the reorganisation needs are the merging in of transaction files (the compaction through disposal of tombstones can be done during this merge process).

9.5 Hashed Files

(Not covered in the lectures, because too similar to first year material. But this *is* examinable: note the differences between this and first year material, i.e. collisions on buckets and overflow. I expect answers to exam questions to show an appreciation of these differences; repetition of first year material gets a zero mark.)

In a hashed file there is a relationship between the key field of a record and its disk address. Some writers give different names to two cases:

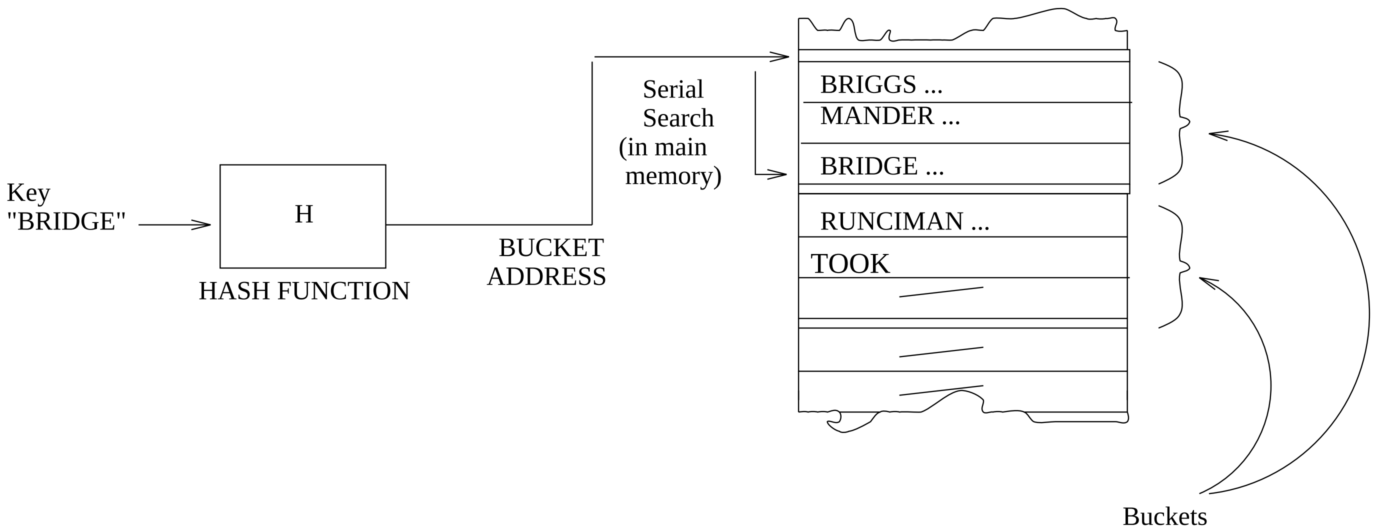
- direct file: the key *is* the address;
- indirect file: the key is 'transmogrified' into the address by means of a *hashing function*/randomising algorithm.

Only the second is realistic.

An amount of disk space must be pre-allocated for a hashed file. The space is divided into fixed size *buckets* which are large enough to hold an integral number of records (possibly one, usually more). A bucket can be read into main memory in a single read operation, and so each is the same size as the input buffer. This is usually the same size as a block (typically 1K on medium-sized machines). Each bucket has an address. Keys are transformed to bucket addresses. Records within buckets have a serial organisation.

To store a record: transform its key to give a bucket address, plonk it into the next free slot in the bucket.

To retrieve a record: transform its key to give a bucket address, read that bucket into main memory, search the bucket serially.



The hash function must give an even distribution of records to buckets. One of the most common is the remainder-of-division method: determine the number of buckets to be allocated to the file, b; select a prime number, p, that is approximately equal to b; take a key, if it is alphabetic, make it numeric, k; the address is the remainder of dividing k by p.

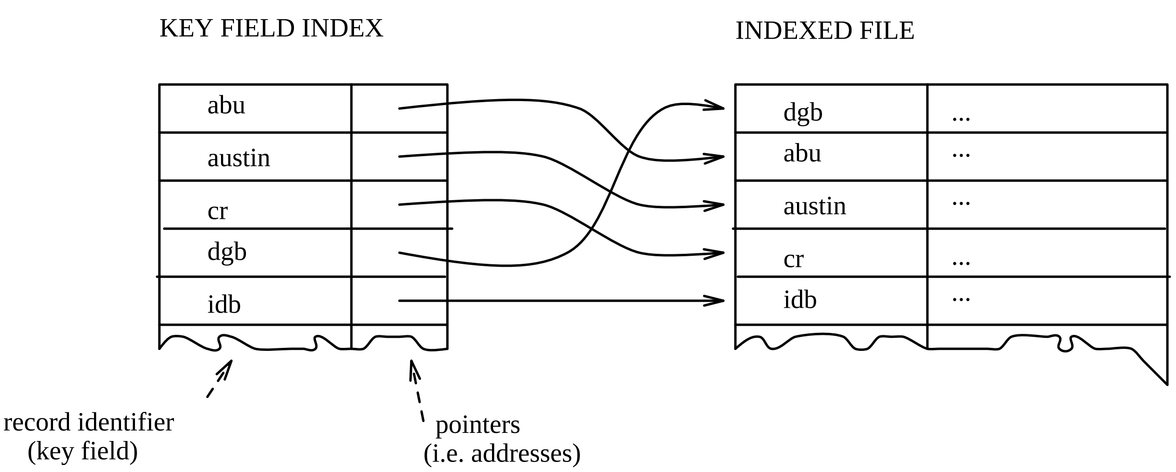
If two (or more) records are hashed to the same bucket, we have a *collision*. This is to be expected: the space allocated to the file will not be enough to accommodate the full range of key values, e.g. a 3-letter key allows 26^3 (17,576) combinations, but if we think only 200 of them, say, will be in use at any one time, we wouldn't want to allocate space for 17000 records; rather, we would allocate space for about 250 records (based on the rule of thumb of treating your estimated maximum (200) as 80% of what you need to allocate). *The whole point of having buckets with more than 1 slot is to cope with collisions.*

But what if the bucket is full and we want to insert into it? This is called *overflow*. We will deal with this when we turn to indexed-sequential files. But note that *hashed files need to deal with overflow*, just as much as indexed-sequential ones do.

Briefly assessing these files: there are storage overheads because the space for the file must be pre-allocated and there is then a trade-off between the space needed for large buckets having lots of empty slots versus smaller buckets but incurring the time lost due to overflow area searching; fetching an arbitrary record is quick provided there is no overflow; getting the next record in a sequence is only possible if you know the next key value (unlikely); insertion is quick if there is no overflow; deletion is simply search followed by replacement by a tombstone; reorganisation in the form of compaction is not really needed (although, after a few deletions, you *might* move overflowed records back to their home buckets); a more radical form of reorganisation might be carried out if there is an unacceptable number of collisions and lots of overflow: you might allocate more space, change the hash function and rehash each record.

9.6 Indexed Files

A separate file, *the index*, defines an access path to the main file (*the indexed file*). The index associates values of the *indexed field* with pointers to the main file's records. Here is a file indexed on its key field:

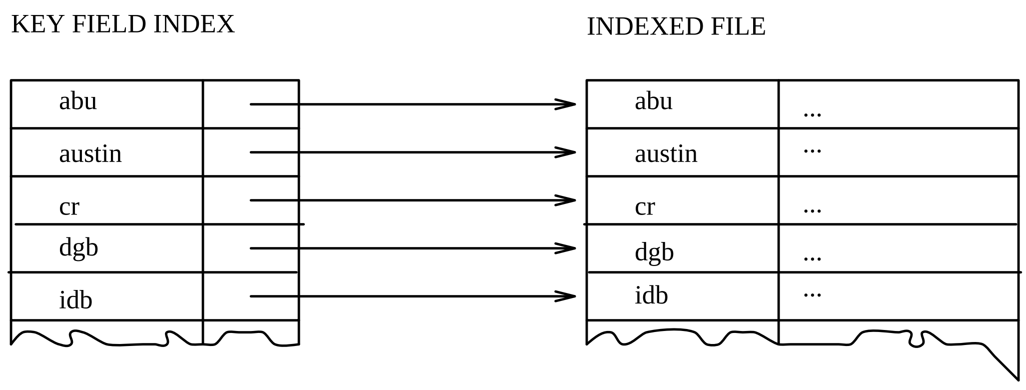


The index needs an organisation itself (sequential in the example above). The indexed file is just a serial file (heap). If space is at a premium, key values need not be duplicated in the indexed file.

The index above is a *dense index*: it contains the same number of entries as the main file. But because it holds less data it can be searched more quickly, e.g. it might fit into main memory.

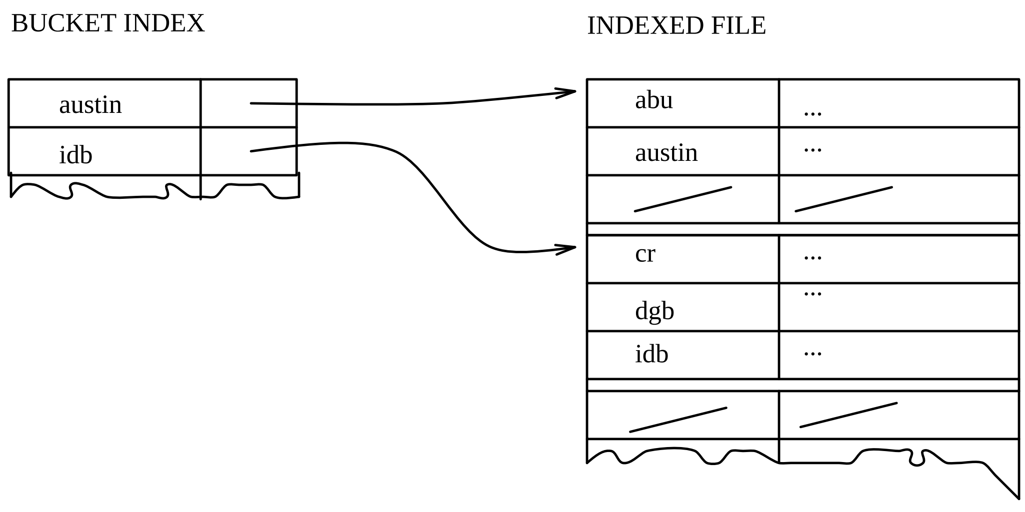
9.7 Indexed-Sequential Files

The indexed file need not be just a heap. It can have a better organisation. Most commonly it would have a sequential organisation, giving an *indexed-sequential file*.



If this were all there was to indexed-sequential files, then they would have no special advantages over plain indexed files.

However, the advantage is that the index need no longer be dense. We can use a *non-dense index*. We split the indexed file into buckets (which again comprise one or more blocks and are the same size as the main memory buffer), and we have *one index entry per bucket* rather than one entry per record. In this scheme, the index has the value of the record with the highest key value presently in the bucket.



The hope is that a bucket will have space for insertions. However, if it does not, then we must have a way of coping with *overflow*. Many methods exist. A selection of them includes:

1. Use the next free slot found by serial search. (This complicates the search algorithm: if a record you are searching for cannot be found in its 'home bucket', then if the bucket is full, we must continue searching for the record until either we find it or we find the next free slot. This implies that deletions must leave tombstones not free slots.)

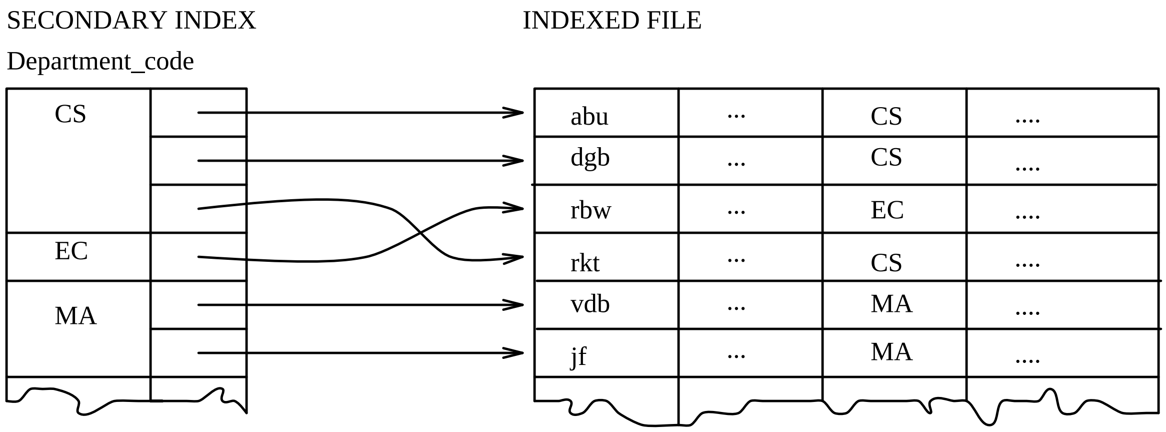
2. Use an overflow area, i.e. a separate set of buckets set aside to hold overflow records. This would be searched serially. (One might distinguish *local* and *global* overflow areas.)

3. For either of the last two methods, the serial search can be avoided by using *chaining*, i.e. maintaining a chain of pointers from the overflowing bucket to its displaced records.

Briefly assessing indexed and indexed-sequential files...there are storage overheads in the form of the index, and, where buckets are used, while this makes the index smaller, the main file becomes bigger because of the need to leave some spare space in the buckets. To fetch an arbitrary record is quick, although this depends on the size and organisation of the index (e.g. is it in main memory, is the index a serial file, a sequential file or a hashed file) and the amount of overflow. To fetch a record in a sequence is trivial in indexed-sequential (you ignore the index and use the main file, which is already ordered, but watch out for overflow), but in plain indexed it is harder. To insert a record if the main file is serial, then simply append the record, if the main file is sequential then there is no problem as long as the record's bucket has space; either way, the index may need updating too; if the index is dense, then a reorganisation of the index may be needed to keep it in sequence; if it is non-dense, then we simply change a value if a bucket has a new highest member. Deletion similarly involves changing the main file *and* the index. Reorganisation needs will depend on the the primary organisations of the index and main file.

9.8 Inverted Files

We can index on a non-key field. The index is then referred to as a *secondary index*. This is always done in conjunction with another organisation for the file.



Again the index needs an organisation and so do the *lists* of pointers. And, again, the index field (Department_code) need not be duplicated in the main file.

The problem with secondary indexes is that each index is associated with an unpredictable number of pointers. One way to avoid this is to use a multi-ring organisation (see next section). Another method is to use a bit vector in the secondary index, i.e. for each index entry, there would be a bit vector having one bit per record in the file. For a value in the index, a particular bit in its vector is set to 1 if the corresponding record in the file has the index value. For the above file, we would get the following index:

SECONDARY INDEX
Department_code

CS	1 1 0 1 0 0
EC	0 0 1 0 0 0
MA	0 0 0 0 1 1

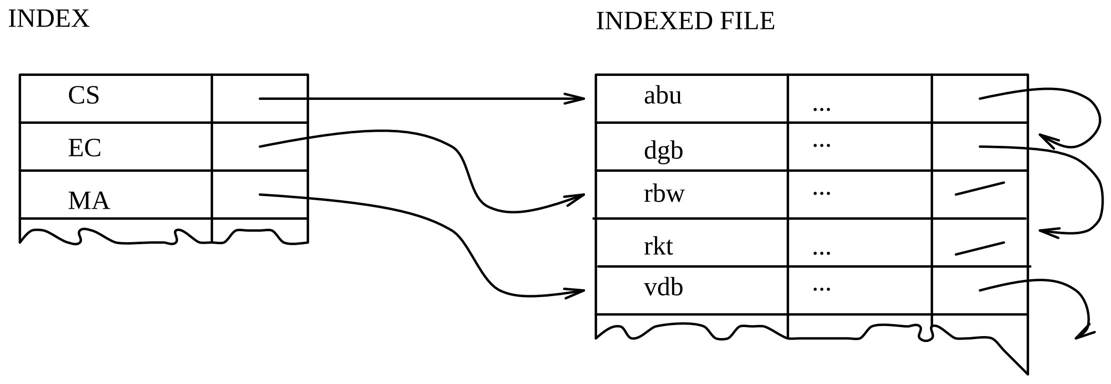
A file with a secondary index is sometimes referred to as having an *inverted organisation*. When *all* the fields have indexes, we have a *fully inverted file*.

Inverted organisations are *very* commonplace in so-called "information retrieval systems" or "document retrieval systems". In such systems, documents such as journal articles are stored together with keywords indicative of their content. Users of these systems express requests in the form of sets of keywords. All the documents that are indexed by all the user's keywords are retrieved.

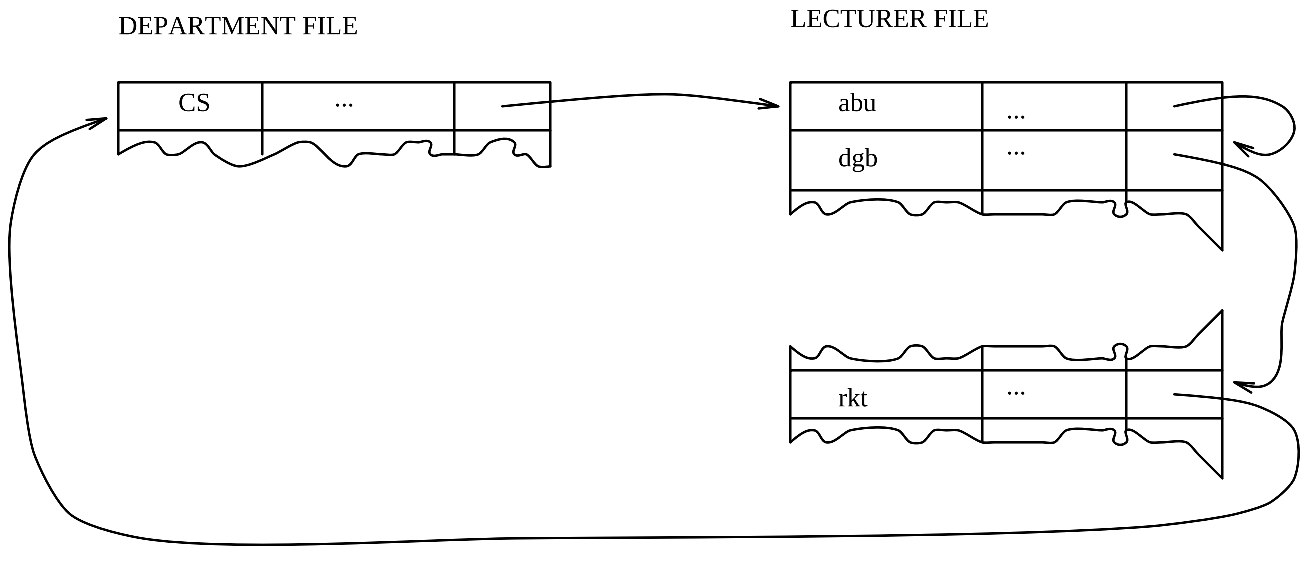
9.9 Multi-Ring Files

Multi-ring files are also referred to as threaded files.

Records with a common attribute value are explicitly chained together.



The index could be a bona fide applications file with departmental data in it. This suits hierarchically arranged data: the origin of the chain is the parent. The structure becomes a ring when the last member points to the parent:



More generally, lecturer records may themselves be parents of chains; a lecturer record could be a member of several different chain *types*; and other pointers may be used, e.g. backwards pointers.

Assessing multi-rings briefly: there are the overheads of pointer storage; to find an arbitrary record is not easy and it is best to use whatever other organisation the file has; to find the next record in a sequence is easy if the chains are ordered (which they might or might not be); insertion involves finding the right position in the chain, getting a free record and linking it in; deletion involves finding the record, de-linking it and linking it in to the free space chain; reorganisation is not really necessary (as long as the free space chain is managed well).

9.10 Multi-Index Files

Searching a large index can be time-consuming. But the index is a file so why shouldn't it have an index? Indexes to indexes gives us trees of indexes. Note that it follows that the non-terminal indexes *must* be non-dense. But then there are many variations on this theme. For example, the terminal indexes can be dense (a pointer to each record in the file) or non-dense (a pointer to a bucket of records). We look at the latter case. The most common such organisations are the *B-tree*, *B+tree* and *B*tree*. In a B-tree/B+tree/B*tree, the tree of indexes is balanced. We concentrate on B+trees which are most common in commercial database systems.

9.10.1 B+trees

A B+tree has an *order*, m. (When $m = 3$, we get a 2-3 tree.) (Note, Whittington's definition of order is different from that used here.)

A B+tree of order m has the following properties:

- tree properties:
  - each node of the tree has at most m children;
  - each node therefore has space for m pointers;
  - each node also has space for $m - 1$ key values, which determine which pointer to follow in a search;
  - there is an ordering: if n_1 and n_2 are children of n and n_1 is to the left of n_2 then the keys of the children of n_1 are all less than the keys of the children of n_2.
- balanced properties:
  - the root has at least 2 children (unless the root has only one node);

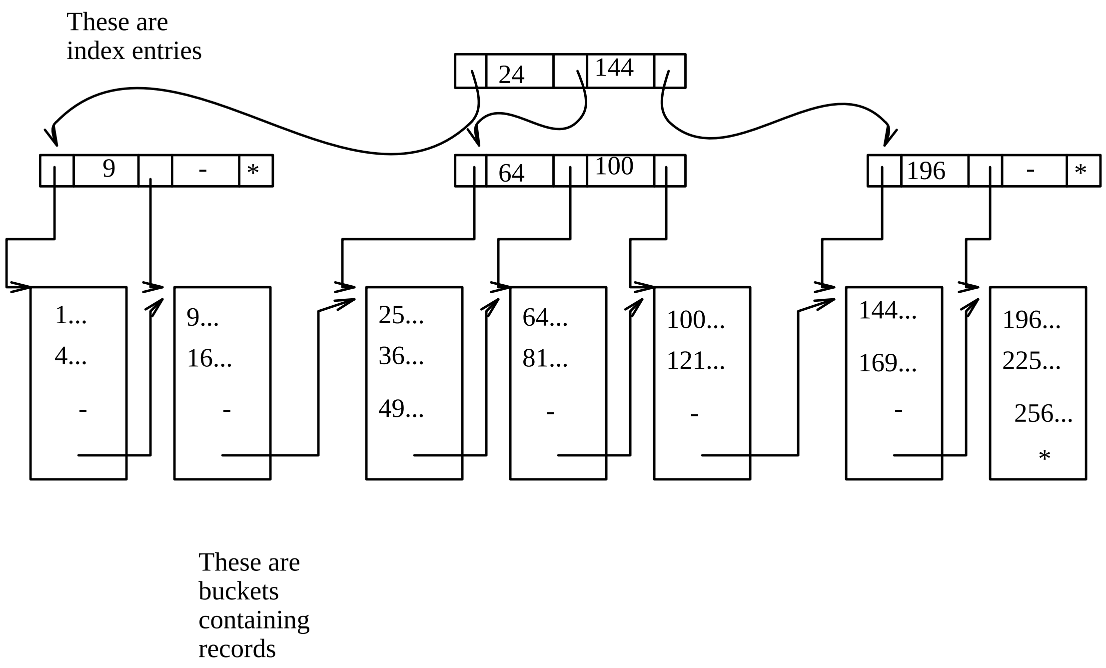
– every index node except for the root will have at least $m/2$ children;

– each path from the root to a bucket of the file has the same length.

Hence the tree must be at least half full (else it would be made into a smaller tree).[2]

Example: B+tree of order 3

(In this example, the buckets also just happen to contain 3 records too.)

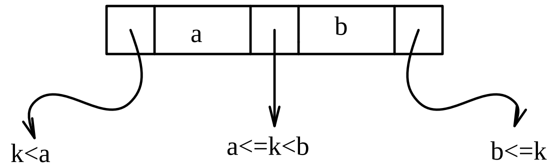


Buckets have been shown chained: they therefore do not need to be contiguous.

9.10.2 Searching a B+tree

- First examine the root node.
- Comparison with keys will indicate which pointer to follow, e.g. searching for record with key, k:

[2]In saying that the tree must be at least half full, what is actually meant is that the index parts will be at least half full. The statement does *not* apply to the fullness or otherwise of the buckets.



- Repeat this until you reach a bucket.
- Read the bucket in and search it for the record. If it's there, then success; else, failure (*it will not be anywhere else; in particular, there is* **no** *overflow path to search.*)

9.10.3 Inserting into B+trees

- Search for the record, as if it were there;
- When you reach the bucket in which it should be stored, simply add the new record to the bucket;
- But the bucket has a fixed size. What if it is full? We do *not* use overflow techniques. We split the node and rebalance the tree, i.e. if bucket, B, is full:
  - Get a new bucket, B';
  - Move the last half of the records from B to B', putting the new record into its rightful place;
  - Take the smallest key in B', and try to insert it into the index node, P, the parent of B;
  - If P itself is full, get a new index block, P', and *split* data between P and P';
  - Now P' is a new node with a new lowest: try to add this to the parent of P;
  - This splitting propagates up the tree until space is found or the root is split;
  - If the root is split, create a new root: the tree will have increased in depth by one level.

9.10.4 Examples of insertion

A. B+tree order 4, bucket length 3

Initial:

(* n *)

(* c * g * j *) (* r * u *)

[a b] [c f] [g h] [j k l] [n o] [r s] [u v w]

Insert e and q: trivial:

(* n *)

(* c * g * j *) (* r * u *)

[a b] [c e f] [g h] [j k l] [n o q] [r s] [u v w]

B. B+tree order 4, bucket length 3

Initial:

(* m *)

(* c * g * j *) (* q * u *)

[a b] [c e f] [g h] [j k l] [m n o] [q r s] [u v w]

Insert p: split [m n o] + p into [m n] [o p], insert o into parent:

(* m *)

(* c * g * j *) (* o * q * u *)

[a b] [c e f] [g h] [j k l] [m n] [o p] [q r s] [u v w]

C. B+tree order 4, bucket length 3

Initial:

(* m *)

(* c * g * j *)   (* o * q * u *)

[a b]   [c e f]   [g h]   [j k l]   [m n]   [o p]   [q r s]   [u v w]

Insert t: split [q r s]+t into [q r] [s t]; try to insert s into parent; parent is full, so split (* o * q * u *) + s into (* o * q *) (* u *) and insert s into root:

(* m * s *)

(* c * g * j *)   (* o * q *)   (* u *)

[a b]   [c e f]   [g h]   [j k l]   [m n]   [o p]   [q r]   [s t]   [u v w]

Insert d into the above: similarly:

(* e * m * s *)

(* c *)   (* g * j *)   (* o * q *)   (* u *)

[a b]   [c d]   [e f]   [g h]   [j k l]   [m n]   [o p]   [q r]   [s t]   [u v w]

D. B+tree order 3, bucket length 3

(Note we have switched from B+trees of order 4 to B+trees of order 3.)

Initial:

(* e * m *)

(* c *) (* i * k *) (* o *)

[a b] [c d] [e g h] [i j] [k l] [m n] [o p q]

Insert f: Split [e g h] + f into [e f] [g h]; insert g into parent; parent is full, so split (* i * k *) + g into (* g *) (* k *) and insert i into root; root is full, so split (* e * m *) + i into (* e *) (* m *) and create a new root (* i *):

(* i *)

(* e *) (* m *)

(* c *) (* g *) (* k *) (* o *)

[a b] [c d] [e f] [g h] [i j] [k l] [m n] [o p q]

9.10.5 Deleting from a B+tree

- Search for the record and remove it from its bucket.
- Changes to the B+tree will be needed in two cases:
  - if the deleted record had the lowest key field in its bucket, modify the B+tree to reflect the new lowest;
  - if the bucket is now empty, dispose of it. Then remove the pointer to this bucket from the parent index node, P. But this might leave P with less than $m/2$ children and thus unbalanced. Redress this in one of two ways:

    redistribution: if a node P' to the right (left) of P at the same level in the tree has more than enough children ($> m/2$), move some from P' to P so both have at least $m/2$; this means that P' (P) has a new low value, so change its parent;

merging: if a node P' to the right (left) of P at the same level in the tree has exactly $m/2$ children, merge P and P'. Now P' (P) no longer exists, so delete the pointer to it from its parent; such a deletion may also cause redistribution or merging. If deletion propagates back to the root, we may have to merge the children of the root. If this leaves the root with one child, the new merged node becomes the new root and the old root is discarded; this reduces the depth of the tree by one.

A. Example of deletion from B+tree order 4, bucket length 3

Initial:

(* n *)

(* c * g * j *) (* r * u *)

[a b] [c f] [g h] [j k l] [n o] [r s] [u v w]

Deletion of k is trivial; to delete g, since it's the lowest in its bucket, the key value appears in the B+tree, so we must find this and replace it with the new lowest; similarly for deletion of n:

(* o *)

(* c * h * j *) (* r * u *)

[a b] [c f] [h] [j l] [o] [r s] [u v w]

Now delete h from the above: discard the empty bucket and change the index. The parent still has $\geq 4/2$ children, so no further work to be done:

(* o *)

(* c * j *) (* r * u *)

[a b] [c f] [j l] [o] [r s] [u v w]

B. Example of deleting from B+tree order 5, bucket length 3

Note that the order is now 5.

Initial:

(* m * s *)

(* c * g * j *) (* o * q *) (* u * v *)

[a b] [c e f] [g h] [j k l] [m n] [o] [q r] [s t] [u] [v w]

Delete o: discard empty bucket; remove pointer from parent , P; P now has $< 5/2$ children; go to P's parent (the root) and look at P's siblings; the left sibling has $\geq 5/2$ children so *redistribute* (* c * g * j *) (* q *); since j is the new lowest child of P, put j in root and demote m to P:

(* j * s *)

(* c * g *) (* m * q *) (* u * v *)

[a b] [c e f] [g h] [j k l] [m n] [q r] [s t] [u] [v w]

C. Example of deleting from B+tree order 5, bucket length 3

Initial:

(* m * s *)

(* c * g * j *)   (* o * q *)   (* u * v *)

[a b] [c e f] [g h] [j k l] [m n] [o] [q r] [s t] [u] [v w]

Delete o: discard empty bucket; remove pointer from parent P; P now has $< 5/2$ children; go to P's parent (the root) and look at P's siblings; the right sibling has $= 5/2$ children so *merge* (* q *) (* u * v *); delete the pointer to (* u * v *) from the root; (interject key value s into merged node); root now has $< 5/2$ children (but still > 1); since this is the root, this is OK (if it hadn't been, we'd have propagated the deletion):

(* m *)

(* c * g * j *)   (* q * s * u * v *)

[a b] [c e f] [g h] [j k l] [m n] [q r] [s t] [u] [v w]

D. Example of deleting from B+tree order 5, bucket length 3

Initial:

(* m *)

(* c * g *)   (* q * s *)

[a b] [c e f] [g] [m n] [q r] [s t]

Delete g: discard empty bucket; remove pointer from parent, P; P now has $\leq 5/2$ children; go to P's parent (the root) and look at P's siblings; the right sibling has = 5/2 children so merge (* c *) (* q * s *); delete pointer to (* q * s *) from the root; (interject key value m into merged node); this give the intermediate position:

```
                         ( * )

                 ( * c * m * q * s * )

  [a b]      [c e f]          [m n]      [q r]      [s t]
```

Now the root has only one child, so discard the old root:

```
                 ( * c * m * q * s * )

  [a b]      [c e f]          [m n]      [q r]      [s t]
```

We end with a quick assessment of B+trees.

Storage overheads are high due to the several layers of indexes but the guarantee that the index tree is always at least half full ensures that this extra index space is controlled. When fetching an arbitrary record, there is a fixed length path from the root to the main file; once this path has been traversed, the bucket is read in and searched sequentially; the path length is less than $(1 + log_{m/2}(n/b))$, where m is the order, n is the number of records in the file and b is the *average* number of records in each bucket, e.g. for 2,000,000 records and order 199 and bucket size of 10, then there will be less than 4 comparisons; remember that there is no overflow searching. Fetching in sequence is easy because the underlying file is sequential. The insertion time is the search time plus the index updating. The deletion time is similar. Reorganisation is not really necessary (it's done on the fly), as long as the free space is managed well.

9.11 The Internal Level of a RDBMS

RDBMSs such as R:BASE will, by default, store all tables as serial files. You may explicitly request alternative storage structures. Unfortunately, the choice is poor. The only alternative structure mentioned in the SQL standards, and hence the only alternative structure that commercial vendors bother to supply, is the use of indexes. (No sequential files, no hash files, no multirings.)

You indicate that you want a table to be stored as an indexed file with an index file using the following SQL command:

```
CREATE INDEX ON S SNo
```

The make SNo an index into table S.

Indexes can be created for any or even each of the columns of a table (i.e. primary key indexes are possible, but so are secondary indexes, and even fully inverted files). (In R:BASE, you can also define an index for more than one column in combination, but the syntax for this is not worth the trouble in this course.)

In the light of this paucity of choice, I think that it is small wonder that many people continue to question whether a RDBMS can meet the performance requirements of their desired system. Yet it would take almost no effort on the part of RDBMS vendors to add other storage structures such as hash files.

That was the bad news. The good news is that the way all of this happens in RDBMSs gives a high degree of *physical data independence*. Remember that physical data independence characterises the extent to which changes to the storage structures affect other parts of the system.

In RDBMSs, you can have a table which is already full of data, and then decide to add an index to it. The index will be built without affecting other parts of the system (e.g. views or programs which use the table). Then, you can remove the index (`DROP INDEX column in table`) later on if you want, and again the functioning of the rest of the system will not be affected. Thus, changes at the internal level do not affect the functioning of other parts of the system (there's no need to rewrite view definitions, etc.). The only affect of adding an index will be possibly to increase the speed of queries and to decrease the speed of updates.

9.12 Exercises

EXAM QUESTION: Question 8, Part (i) only, 1992

[12 marks] Consider the following B+tree whose *order* and *bucket size* are both 3:

(* 50 * 58 *)

[42 47]　　[50 52 55]　　[58 59]

Perform the following operations *successively*, and draw the state of the B+tree after each one, together with an explanation of why the tree changes in the way it does:

(a) insert 45

(b) insert 56

(c) insert 57

(d) delete 55

(e) delete 58

(f) delete 59.

Chapter 10

Implementing Query Languages

We've seen the storage structures at the internal level and we've seen how to specify internal level information to a RDBMS. In this chapter, we show in more detail how the choice of physical storage structure affects the efficiency of various queries. In subsequent chapters, we will show you how these considerations determine what indexes to create, etc.

10.1 Implementing SQL

The easiest way to implement any calculus-based language, such as SQL, is to convert the queries to their algebra equivalents. For example, the query to get the names of students in Computer Science who study year 2 courses:

```
SELECT SName  ---------------------(1)
  FROM S, SC, C -------------------(2)
 WHERE S.Dept  = 'CS'    AND
       C.Year  = 2       AND
       S.SNo   = SC.SNo  AND-------(3)
       C.Ccode = SC.Ccode
```

The SELECT list (1) is basically a projection; the FROM list (2) is basically a product; and the WHERE clauses (3) are basically restricts:

$$\begin{array}{l} \text{project (restrict (product (product S, SC), C),} \\ \qquad\qquad \text{S.Dept} = \text{`CS'} \wedge \text{C.Year} = 2 \wedge \\ \qquad\qquad \text{S.SNo} = \text{SC.SNo} \wedge \text{C.Ccode} = \text{SC.Ccode), \{SName\})} \end{array}$$

This can now be executed in the fashion described in the rest of this chapter.

However, there is a complication with this approach: nested queries. Many nested queries (particularly those that use simply IN) can be de-nested, but others (particularly those that use something other than IN) cannot. If a query cannot be de-nested, then it cannot be converted in the automatic way shown above into algebra. (It can still probably be converted; it's just that no *simple* algorithm can be given.)

And then there is another complication: SQL has, over the years, acquired a lot of features that have no counterparts in the algebra, e.g. the built-in functions (SUM, AVG, GROUP BY, etc.). These make SQL more useful as a language but they lack a proper theoretical foundation in the calculus/algebra.

Therefore, many queries will have to be evaluated by some more ad hoc method, rather than by translating to the algebra. And, if some have ad hoc methods, why not give all SQL queries ad hoc methods? This is what is done in most RDBMSs.

However, a careful consideration of efficiency improvements is best gained by looking at the better motivated algebra. Much of what we look at will apply equally well to SQL.

10.2 Implementing Relational Algebra

We will look at the operators of relational algebra and consider how they might be implemented at the internal level.

10.2.1 Cartesian Product: product(R, S)

There is no real alternative to having two loops, one nested within the other:

```
for each tuple in R
    for each tuple in S
        create a new tuple
```

If the cardinality of R is m, and the cardinality of S is n, then the cardinality of product(R, S) is $m \times n$. If the degree of R is p, and the degree of S is q, then the degree of product(R, S) is $p + q$.

The implementation is inefficient and the result can be a very large relation. Therefore, we want to avoid Cartesian products.

Question: Does it matter which way round the loops are nested?

10.2.2 Union, Intersection and Difference

Method 1: Sort each relation first into ascending order of key value (if this order of access is not already available). Then step through both sorted files. Write out a tuple to the result relation if

- it is in either (union)
- it is in both (intersection)
- it is in the first but not the second (difference)

The problem with this method is that sorting is an expensive operation.

Method 2: Create a new empty hash file. Hash tuples of R into the file. Hash tuples of S into the file. In the event of a collision with matching tuples:

- remove one of the equal tuples (union)
- mark one of the equal tuples as being a result tuple (intersection)
- remove both of the equal tuples (difference)

This can be faster than Method 1 but has the space overheads associated with the hash file (which may need to be bigger than the cardinality of R + the cardinality of S).

10.2.3 Restriction: restrict(R, θ)

Suppose A and B are sets of attributes of R (possibly singleton sets).

Method 1: Retrieve every tuple and test it.

Method 2: If the file is ordered on A and θ is of the form A = 'c', then you can binary search R for the tuple where A = 'c'.

Method 3: Similarly, if A is the field on which the file is indexed/hashed and θ is of the form A = 'c', then you can use the index/hashing function to retrieve the tuple where A = 'c'. (This applies to B+trees too.)

Method 4: If there is a secondary index on B and θ is of the form B = 'd', then use the secondary index to retrieve all the tuples where B = 'd'. (This applies to multirings too.)

Method 5: If the file is indexed-sequential with the ordering of the file and the index defined on A and θ is of the form A > 'c' (or A $\geq$ 'c', A < 'c', A $\leq$ 'c'), use the index to get the tuple where A = 'c' and then search sequentially forwards (or backwards). (This applies to B+trees too.)

Method 6: If θ is of the form $\theta_1 \wedge \theta_2$, use one of methods 1-5 to get the tuples satisfying θ_1, then check them individually to see if they satisfy θ_2. (Obviously, we want to have θ_1 more restrictive than θ_2.)

Method 7: If θ is of the form $\theta_1 \wedge \theta_2$, you might be able to intersect the index entries satisfying θ_1 with the index entries satisfying θ_2.

10.2.4 Project: project(R, B)

The only problem that arises when implementing projection is that a projection may cause the result to contain duplicates. These must be removed.

If B (the set of attributes we keep) includes the key of R, then this duplicates problem does not arise. If B does not include the key then:

either sort the result and remove the duplicates

or create a new hash file, hash the result into the hash file; in the case of collisions of duplicate tuples, discard the duplicate.

(Note most RDBMS do not remove duplicates unless this is explicitly requested. For example, in SQL, the default in a SELECT command is not to remove duplicates; duplicates will only be removed if you explicitly request a SELECT DISTINCT. In db++, which has an algebraic query language, the project operator does not remove duplicates. To remove them, you first explicitly request that a special kind of sort be done on the relation, where the sort removes duplicates, then you issue the project command. In R:BASE's algebra facilities, `PROJECT` does not remove duplicates.)

10.2.5 Equi-join: equi-join(R, S, R.A = S.B)

(This applies to nat-join too.)

This can be implemented as a nested loop, as with Cartesian product, where you only put a tuple into the result relation if R.A = S.B. The performance of this will be as dreadful as that of Cartesian product. We can do better than Cartesian product by exploiting the fact that R.A must be equal to R.B.

Method 1: If B is indexed or hashed, loop through R, and for each value x of A, retrieve the tuple (or tuples if B is a secondary index) where B $= x$ from S using the index/hash function.

(If both A and B are indexed/hashed, loop through the one which is likely to contain more tuples that satisfy the join condition, and index/hash into the one where fewer tuples are likely to satisfy the condition.)

Method 2: If R is ordered on A and S is ordered on B (or they can be efficiently sorted so that this is so) and A and B contain no duplicates, go through R and S as if you were going to merge them, but instead of merging them, create new tuples where R.A = S.B, and discard other tuples.

This can be very efficient (as long as R and S are already ordered) as we need work through each file only once.

Method 3: Create a new hash file. Hash R into the file. Hash S into the file. Create result tuples where you have collisions and R.A = S.B.

This also works through each file only once and so can be quite efficient. However, there is the space overhead of the hash file.

10.2.6 What Does A Typical RDBMS Do?

We will consider what R:BASE does in executing SQL. It does not de-nest and translate to algebra. Rather, it has some (probably quite hacky) SQL evaluation routines for executing the SQL directly (i.e. it works by lots of nested loops).

It will generally therefore do a serial search through the tables mentioned in a query. But if a table has an indexed column, then, under fairly restrictive conditions, that index might be used.

R:BASE uses indexes to improve the performance of WHERE clauses and ORDER BY clauses. We'll describe the former.

It appears that R:BASE will use a maximum of one indexed column per query. Its choice algorithm is:

```
IF an indexed column is followed by =,
   Use that column
ELSE
   IF an indexed column is followed by IS NULL,
```

```
      Use that column
   ELSE
      IF an indexed column is followed by BETWEEN,
         Use that column
      ELSE
         Don't use any indexes at all.
```

Notes: (1) Clearly this is pretty poor. Indexes could be used for operators other than =, IS NULL and BETWEEN, but they are not. (2) This algorithm is supposed to pick for use the index used in the most restrictive condition (roughly) (but this doesn't explain why it won't *also* use other indexes mentioned in the query). (3) If there are several indexed columns followed by =, it picks the one that appears first in the query (and similarly for IS NULL and BETWEEN).

10.2.7 Summary

Let's summarise what we saw with the algebra.

So far we have looked only at individual operations. However, queries usually consist of several operations. Performing the operations individually one after the other can be less efficient and can generate lots of temporary files for holding intermediate results.

For example, we have already seen that a Cartesian product followed by a restrict might be less efficient than the equivalent join (which is effectively a Cartesian product and a restrict in combination), because the condition of the join can help us to reduce the number of tuples we need to consider.

The file manager of a clever system will try to spot operations that can be done as combinations or might change the order in which operations are done.

This leads us into *query optimisation*.

10.3 Query Optimisation

In the previous section we were looking at what the file manager can do to improve efficiency. Here we are looking at what a clever DBMS can do, or, even if the DBMS won't do it automatically, what a clever user can do, to improve efficiency.

Query optimisation is not really "optimisation". It is merely improvement. There are three basic types:

1. We have seen the first already (section 10.2): choosing the best access strategy on the basis of the availability of indexes, etc. and the size of the relations, and the opportunities to combine operations.

2. "Semantic query optimisation": uses AI/theorem-proving techniques to spot redundancies, e.g. if it is a constraint that a course is offered to no more than one year group, then asking for a course that is given to both first and second years is redundant.

3. Transform queries into algebraically *equivalent* ones that give the same result but with *probably* better performance.

We look at (3) in more detail.

10.3.1 Algebraic Optimisation

Basically, since restricts and projects never increase the size of a file, we try to do these *before* doing joins, unions, etc.

The method is to represent the query as a *query tree*, and then follow these heuristics for transforming the tree into an equivalent, but possibly more efficient, one:

1. break up complex restricts (to allow greater freedom of movement)

2. move restricts down the tree as far as possible

3. combine products followed by restricts into joins

4. move projects down the tree as far as possible.

Then use the implementation methods discussed in section 10.2 (i.e. exploit indexes, etc. where possible).

Example

> Get the names of students in the Computer Science Department
> who study year 2 courses.

Somewhat artificially, I will use the query:

$$
\begin{aligned}
&\text{project(restrict(product (product(SC, S), C),}\\
&\qquad\qquad \text{SC.SNo} = \text{S.SNo} \wedge \text{SC.Ccode} = \text{C.Ccode} \wedge\\
&\qquad\qquad \text{Dept} = \text{`CS'} \wedge \text{Year} = 2), \{\text{SName}\})
\end{aligned}
$$

First we represent this as a query tree:

```
         project {SName}
                |
             restrict
     SC.SNo ∧ SC.Ccode ∧
    Dept = 'CS' ∧ Year = 2
                |
             product
                |
           ┌────┴────┐
        product      C
           |
        ┌──┴──┐
        SC    S
```

Evaluate tree from leaves up to root.

(1) Break up complex restricts.

Use

$$\text{restrict}(R,\ \theta_1 \wedge \theta_2) \equiv \text{restrict}(\text{restrict}(R,\ \theta_2),\ \theta_1)$$

```
           project {SName}
                  |
       restrict SC.SNo = S.SNo
                  |
     restrict SC.Ccode = C.Ccode
                  |
         restrict Dept = 'CS'
                  |
          restrict Year = 2
                  |
               product
                  |
             ┌────┴────┐
          product      C
             |
          ┌──┴──┐
          SC    S
```

(2) Move restrict down the tree as far as possible.

Use

(a) $\text{restrict}(\text{restrict}(R,\ \theta_2),\ \theta_1) \equiv \text{restrict}(\text{restrict}(R,\ \theta_1),\ \theta_2)$

(b) $\text{project}(\text{restrict}(R,\ \theta),\ A) \equiv \text{restrict}(\text{project}(R,\ A),\ \theta)$

as long as θ only contains attributes in A, the set of attributes being kept

(c) restrict(nat-join(R, S), θ) $\equiv$ nat-join(restrict(R, θ_1), restrict(S, θ_2))
as long as θ can be rewritten as $\theta_1 \wedge \theta_2$ where θ_1 involves only attributes of R and θ_2 involves only attributes of S
(This obviously works for product too.)

For the example this gives:

```
                  project {SName}
                        |
             restrict SC.SNo = C.Ccode
                        |
                     product
                        |
          +-------------+--------------+
          |                            |
restrict SC.SNo = S.SNo      restrict Year = 2
          |                            |
       product                         C
          |
   +------+---------+
   |                |
  SC   restrict Dept = 'CS'
                    |
                    S
```

(3) Combine products followed by restricts into joins.

```
          project {SName}
                |
        nat-join on Ccode
                |
       +--------+---------+
       |                  |
nat-join on SNo    restrict Year = 2
       |                  |
  +----+-------+          C
  |            |
 SC   restrict Dept = 'CS'
               |
               S
```

(4) Move projects down the tree as far as possible.

Use:

(a) project(project(R, A $\cup$ B), B) $\equiv$ project(R, B)

(b) project(nat-join(R, S), A) $\equiv$

project(nat-join (project(R, $A_1 \cup A_2$), (project(S, $A_3 \cup A_4$))
A)
where $A_1 \cup A_3 = A$ (the attributes we want) and A_2 and A_4
are not
in A but are needed in the join condition.

For the example this gives:

- project {SName}
  - nat-join on Ccode
    - project {Ccode, Name}
      - nat-join on SNo
        - project {SNo, Ccode}
          - SC
        - project {SNo, Name}
          - restrict Dept = 'CS'
            - S
    - project {Ccode}
      - restrict Year = 2
        - C

This is the final tree. Evaluating it from the leaves upwards ensures that restricts and projects are done before joins and products whenever possible, thus ensuring that the smallest possible arguments are fed into the joins and products.

10.3.2 What Does a Typical RDBMS Do?

R:BASE does no query optimisation, i.e. it does not chop and change the SQL into equivalent, faster SQL. Unfortunately, it is not unusual in this.

However, the clever SQL programmer will try to write optimised queries himself or herself. (Disclaimer: the suggestions below are based on careful measurements carried out on Empress RDBMS. They have not yet been verified carefully for R:BASE.) Here are some rules of thumb:

- Put your most restrictive WHERE-clauses first.
- Remember that R:BASE chooses the first index it encounters that has an = after it. So to influence which of several indexes gets used, again order your WHERE-clauses carefully.

- On the whole, put WHERE-clauses that do single table comparisons before those that do multiple table comparisons (i.e. the equivalent of trying to do restricts before joins).
- Order the tables in the FROM-clause so that they appear in an order that is similar to the order in which their columns are mentioned in the WHERE-clauses.

10.4 Exercises

A) EXAM QUESTION: Question 8, Part (ii) only, 1992

(ii) [5 marks] Suppose we have two relations as follows:

R(**A:** B, C, D)
S(**E:** F, G)

R occupies 12 blocks of disk space; S occupies 8 blocks of disk space.

R is an indexed-sequential file with its primary index defined on attribute A. It also has secondary indexes defined on attributes B and D.

S is a serial file.

For each of the following relational algebra queries, explain what is likely to be the most efficient way of executing the query given what you know about the storage structures of R and S.

(a) restrict(R, A $<$ "32")

(b) restrict(R, B = "12" $\wedge$ C $>$ "37")

(c) restrict(R, B = "12" $\wedge$ D = "97")

(d) product(R, S)

(e) project(R, {A})

B) EXAM QUESTION: Question 8, 1991

Canter Pollsters compile a "Hit Parade" of record sales in the Eastern Counties using the following database:

RECORDS(**Record_id:** Title, Artiste)
SHOPS(**Shop_name:** Town)
SALES(**Shop_name, Record_id:** Qty_sold)

The following relational algebra query retrieves sales of the record "Fiesta" by the Pogues for each shop in Sudbury:

$$
\begin{array}{l}
\text{project (restrict (product (product (SALES, RECORDS), SHOPS),} \\
\qquad\qquad \text{SHOPS.Shop\_name} = \text{SALES.Shop\_name} \wedge \\
\qquad\qquad \text{RECORDS.Record\_id} = \text{SALES.Record\_id} \wedge \\
\qquad\qquad \text{RECORDS.Title} = \text{“Fiesta”} \wedge \\
\qquad\qquad \text{RECORDS.Artiste} = \text{“The Pogues”} \wedge \\
\qquad\qquad \text{SHOPS.Town} = \text{“Sudbury”}), \\
\qquad \{\text{SALES.Shop\_name, SALES.Qty\_sold}\})
\end{array}
$$

(i) [2 marks] Represent the query as a *query tree*.

(ii) [14 marks] Using the following equivalences of relational algebra, transform the query tree as much as possible to give an algebraically equivalent tree which is likely to execute more efficiently. For each transformation you apply, make clear which equivalence justifies the transformation.

`At this point, the question included the necessary equivalences.`

(iii) [4 marks] You are now told the following information:

- at the moment, all relations are stored as serial files;
- there are about 240 tuples in RECORDS at any one time;
- there are about 80 tuples in SHOPS; there are usually about 4 shops per town;
- there are about 19200 tuples in SALES.

One of the relations is to have its storage structure changed to *indexed-sequential*.

From the point of view of improving the performance of your optimised query, which relation's storage structure should be changed, and on which attribute(s) should the index be defined?

Justify your answer and state any assumptions that you make.

C) EXAM QUESTION: Question 9, 1992

Sudbury Bird Sanctuary needs cash. Therefore, it has decided to launch a Fund-A-Fowl scheme whereby members of the public (SPONSORS) can adopt birds from the sanctuary (BIRDS) and help to pay for their keep. The *many-to-many* relationship between SPONSORS and BIRDS is expressed in the ADOPTION relation:

SPONSORS(**Sponsor_id:** SName, Address)
BIRDS(**Bird_id:** BName, Species)
ADOPTION(**Sponsor_id, Bird_id:** Cash_paid_so_far)

The following relational algebra query retrieves the names of sponsors who have made contributions towards the upkeep of swans called "Stevie".

project(restrict(product (product (SPONSORS, ADOPTION), BIRDS),
BIRDS.Bird_id = ADOPTION.Bird_id $\wedge$
SPONSORS.Sponsor_id = ADOPTION.Sponsor_id
$\wedge$
BIRDS.BName = "Stevie" $\wedge$
BIRDS.Species = "Swan",
{SPONSORS.SName, ADOPTION.Cash_paid_so_far})

(i) [2 marks] Represent the query as a *query tree.*

(ii) [14 marks] Using the following equivalences of relational algebra, transform the query tree as much as possible to give an algebraically equivalent tree which is likely to execute more efficiently. For each transformation you apply, make clear which equivalence justifies the transformation.

`At this point, the question included the necessary equivalences.`

(iii) [4 marks] Write statements in *relational algebra* to perform the following queries.

(a) Retrieve the identification number and name of each sponsor who has *both* contributed more than £3,000 towards the upkeep of Mandy the moorhen *and* contributed something towards the upkeep of coots.

(b) Retrieve the identification number and name of each sponsor who is recorded as having adopted *all* the birds.

D) EXAM QUESTION: Question 8, 1993

This is the schema of the database used by Sudbury's oldest cat breeder, Miss Genevieve Bienfaisance, showing, for each cat, a unique identification number, its name, its sex and the identification numbers of its mother and father:

CATS(**Cat_no:** Catname, Sex, Mother_catno, Father_catno)

(i) [7 marks] Write statements in *relational algebra* to perform the following queries:

(a) Retrieve the names of all cats who are not parents.

(b) Retrieve the names of both parents of all the cats called Muffin, Crumpet and Teacake.

(ii) [4 marks] Explain *in detail* why you cannot write queries for the following using only the 8 operators of relational algebra:

(a) Retrieve the ancestral tree (parents, grandparents, great grandparents, etc.) of cats called Bathbun.

(b) Retrieve the equivalent to what is retrieved by the following SQL query:

```
select Mother_catno, count(*)
  from CATS
group by Mother_catno
```

(iii) [9 marks]

(a) State in outline what steps you would use to 'optimise' a relational algebra query. (You are not being asked to give the relational algebra equivalences.) [2 marks]

(b) Here is a relational algebra query which retrieves the sex of all mother cats:

T1 := rename(CATS, Cat_no → Cat_no1, Catname → Catname1
Sex → Sex1, Mother_catno → Mother_catno1,
Father_catno → Father_catno1)
T2 := join(CATS, T1, Cat_no = Mother_catno1)
T3 := project(T2, {Sex1})

Here is a partial version of the optimised query:

T1 := rename(CATS, Cat_no → Cat_no1, Catname → Catname1
Sex → Sex1, Mother_catno → Mother_catno1,
Father_catno → Father_catno1)
T2 := project(CATS, Ψ)
T3 := project(T1, Φ)
T4 := join(T2, T3, Cat_no = Mother_catno1)
T5 := project(T4, {Sex1})

What would the sets at Ψ and Φ be? [3 marks]

(c) Being able to retieve the sex of a mother cat (as in part (b)) does have a use. What is it? [1 mark]

(d) If `R` is a relation, θ is a restrict-condition, and `A` is a set of attributes, state what restrictions must be placed on θ for the following to be an equivalence of relational algebra:

`project (restrict (R,` θ`), A)` $\equiv$ `restrict (project (R, A),` θ`)`

Is there a use for this equivalence in a more sophisticated version of the optimisation algorithm you gave in part (iiia)? If you think there is, state it. If you think there is not, explain why not. [3 marks]

E) EXAM QUESTION: Question 5, 1994

(i) [2 marks] Simplify the following statement of relational algebra:

project(project(R, {A, B, C}), {A, B})

where R is a relation name and A, B and C are attributes of R.

(ii) [18 marks] Sudbury Health & Safety Inspectorate visit restaurants and record violations of Health and Safety Laws. Here is their database schema.

RESTAURANTS(**RId:** RName)
VIOLATIONS(**VId:** VDesc)
REPORTS(**RId, VId:** Date-reported, Date-resolved)

The following query computes the dates on which rats were found in the kitchens of the restaurant called 'The Cooks Tour':

project(restrict(product (product (RESTAURANTS, REPORTS),
VIOLATIONS),
VIOLATIONS.VId = REPORTS.VId $\wedge$
RESTAURANTS.RId = REPORTS.RId $\wedge$
VIOLATIONS.VDesc = 'Rats' $\wedge$
RESTAURANTS.RName = 'The Cooks Tour'),
{REPORTS.Date-reported})

Represent the query as a *query tree* and then, using the following equivalences of relational algebra, transform the query tree as much as possible to give an algebraically equivalent tree which is likely to execute more efficiently. For each transformation you apply, make clear which equivalence justifies the transformation.

`At this point, the question included the necessary equivalences.`

Chapter 11

Applications Design

We discussed application design briefly in Chapter 2, where we showed how database analysis and design fit within a fuller system development cycle (that of SSADM). Since this is a database course, we look only at those aspects of application design that impinge on database design.

Just as database analysis and design form a three stage process (conceptual data modelling, logical data modelling, and implementation), so application design can also be seen as a three stage process:

Conceptualisation: conceptual process modelling/ process analysis — interview the users, construct, for example, data flow diagrams.

Representation: logical process modelling/ process design — design the application control facilities (such as the menu structure), design the forms, design the reports, and specify the programs.

Implementation: physical process modelling/ 'programming' — build the applications using DBMS application facilities or by good, old-fashioned programming, or a mix of the two.

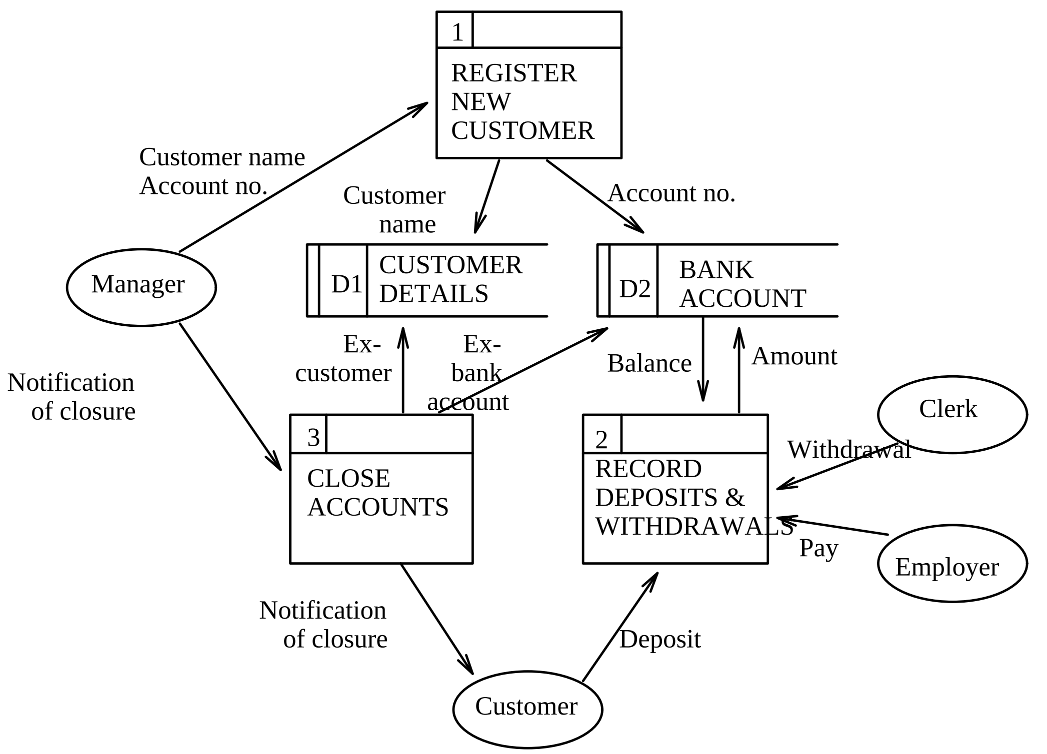
Here's a bit more detail on the above.

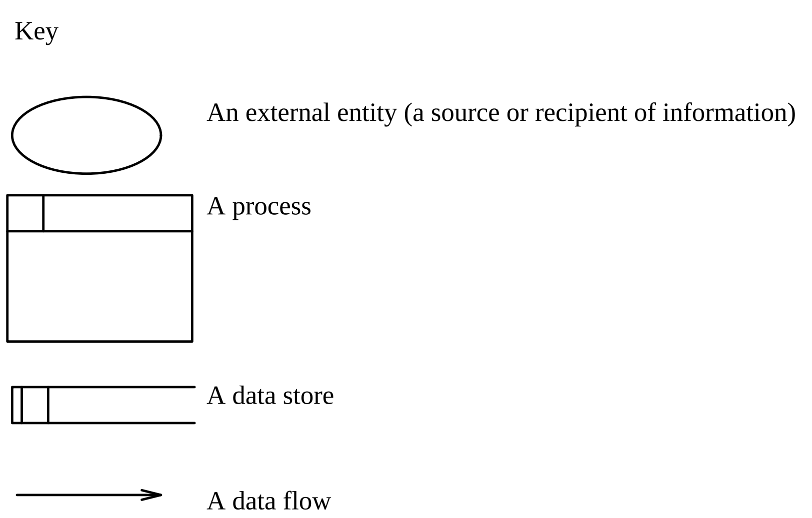
11.1 Conceptual Process Modelling

You have been learning many ways to model the processes that go on in organisations in the software engineering part of your degree course. We'll ignore all that because perhaps the most common method for doing this in industry (partly because it is prescribed by SSADM) is the use of data flow diagrams (DFDs). DFDs show the information flows within an organisation.

Specifically, they show how information enters and leaves the organisation, what changes the information, and where the information is stored.

Here's a simple example for a bank:



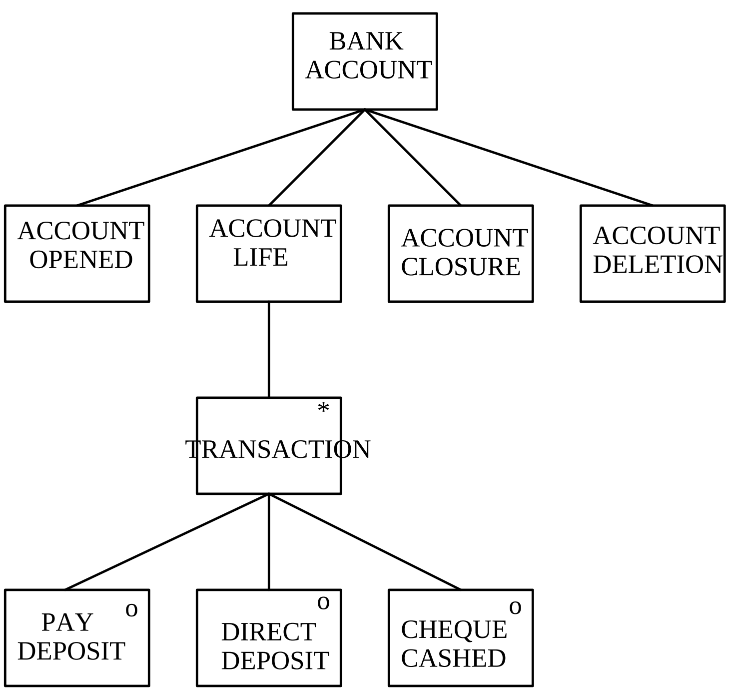


(I assume that it's pretty obvious what's going on here. What's less obvious is what methodology you would use to come up with such a diagram.

Happily, I can assume that your software engineering courses have supplied you with plenty of ideas about that.)

By combining information taken from the ER diagram and the DFD, we can now construct Entity Life Histories. An Entity Life History is a diagrammatic representation of the life of a single entity, from its creation to its deletion. Basically, a life history is drawn for every entity type on the ER diagram. Each flow on the DFD that affects a data store will have a corresponding event on the life history.

Here's a simple example of an Entity Life History for a bank account entity type:



Sequence is conveyed by the left to right ordering. *Alternatives* are indicated by circles in the top right corners of boxes. *Iteration* is shown by an asterisk in the top right corner of a box.

11.2 Logical Process Modelling

It ought to be clear that Entity Life Histories can provide a good handle on some of the work needed for program specification and the design of a hierarchy of menus. In Logical Process Modelling, more detail is put onto Entity Life Histories for exactly these purposes.

Additionally, you would consider the input and output formats that might be used. Most commonly, input and output are done using forms and reports respectively.

Forms: A form is a screen display that can be used to enter and edit data more conveniently than can be done using a browser or explicit SQL commands. A form relates to one or more tables or views in the database. Obviously, a form will probably look like some physical object in the real world, e.g. a paper order form.

At this stage of design, you need to decide which forms you want, what tables/views they pertain to, and what information they will display/allow to be entered/edited.

Reports: A report is a listing of data from the database that is to be printed. It ensures that data is formatted more attractively than it would be if we were simply to use the browser or to query using SQL or QBE. A report relates to a table or view in the database.

At this stage of design, you need to decide which reports you want, what tables/views they pertain to, and what information they will display.

11.3 Implementation

At this stage, you implement the forms, the reports and the menu system, and you write any programs that are needed.

Many RDBMSs, including R:BASE, come with *Fourth Generation Language* (4GL) facilities, which make application implementation *relatively* straightforward.

1GLs are machine code, 2GLs are assembly languages, 3GLs are high-level procedural languages (e.g. COBOL, Pascal, Ada). 4GLs are high-level and non-procedural. Most importantly, the detailed procedures are not written by the programmer but by the fourth generation software itself using 'parameters' supplied by the programmer.

> "Compared to third generation languages, they require fewer lines of coding, and programs are quicker to write and test and easier to maintain. Fourth generation systems are designed for interactive use... by menus... so that non-computing people can develop their own applications."[Avison 1992, p.261]

If these advantages are true, then life will be wonderful: the applications backlog should be reduced, the programmer shortage (if there still is one in a Britain destroyed by too many years of Tory rule) will be alleviated, communication problems between users and computing people will be irrelevant (as end-users will develop their own systems) so systems will be more likely to meet their requirements, there will be peace in our time and jam for tea. However, before we get carried away, we should note that 4GLs need increased computer resources, and the code they generate usually runs slower than that which might be hand-crafted by a programmer. Furthermore, naïve managers who accept the 4GL hype get quite a shock when they find out just how much 'real' programming still needs to be done to handle the things that the 4GL cannot do.

The components of a 4GL might be:

- A *DBMS*, usually a RDBMS, preferably including a sophisticated data dictionary and an interactive query language such as SQL or QBE.

- A *Forms Generator*: this allows the easy drawing of a template form on a screen. The 'programmer' places symbols on the template to indicate the location of places where the user will be allowed to type values and links these locations to database columns. When the 'programmer' has completed specification of the form, the Form Generator automatically generates a program that will enable the form to be used for data entry and edit.

- A *Report Generator*: this allows the easy drawing of a template report on a screen, with symbols indicating the locations where data will be displayed, and specification of headings, etc. Again, the system automatically generates a program that enables the report to be used.

- An *Applications Generator* (or Program Generator): this automates the production of some other code. Usually, as a minimum, you can specify a menu structure and say which programs (corresponding to forms and reports) should be run when the user chooses each menu option. Again, the system automatically produces the code that enables the menu system to be run.

  Note that more complex programs can usually not be specified to the Applications Generator and therefore in applications of any real interest it is *very common* for the programmer to have to resort to writing some code himself or herself. In some systems, the 4GL will contain a simple 3GL to allow this (indeed, this 3GL is likely to be the language used for the output of the Applications Generator). In all systems, it ought to be possible to write programs in one or more conventional 3GLs (e.g. COBOL or C) but to embed special database commands in the code: before compilation such a program is passed through a preprocessor to deal with the database commands.

R:BASE offers all of the above facilities. To find out some more about them, you should use the tutorial lessons:

4. (Entering Data in Tables and Forms) —this starts with some stuff about editing data using the browser but then it goes on to how to use forms.
7. (Creating and Using Forms)
8. (Creating and Using Reports)
11. (Creating and Using Applications)

11.4 Conclusion

It is in applications design that you consider how data will be input by and displayed to users given what you know about the processing requirements that users have. You might choose views, forms or reports. More complex processing may require the writing of programs, and the whole system may be brought together under a suitable menu interface.

Using 4GL facilities for application implementation ought to boost productivity. Note also that, as well as producing final applications, 4GLs can be used to generate prototypes (Chapter 2).

Chapter 12

Implementing Relational Databases: Chapter 5 Revisited

In Chapter 5 we looked at the implementation phase of relational database design. I tried to sell you the unconvincing line that there would be a trivial 1:1 mapping from relations in your logical data model to tables in your relational database schema. So all you would need to do, I implied, is tell a RDBMS about the structure of these tables using a DDL or menus.

We then seemed to take a massive detour, looking at query systems (relational algebra) and query languages (SQL), the external level (views) and the internal level (indexes, etc. and query optimisation), and application design.

Now I will try to bring all this together by showing you how factors we have learnt about on our detour make Chapter 5's 1:1 mapping unrealistic.

The basic problem is that, if you simply perform the 1:1 mapping from relations to tables, then the relational database that you end up with has no chance of being efficient enough (in most cases). In the real world, to deal with particular data access or update patterns, we may have to choose other database structures.

12.1 Using Logical Process Modelling

As well as using the logical *data* model to work out what tables we will implement, we also use the logical *process* model. You will recall that this model specifies what views, reports, forms, menus and programs are needed.

These will not only influence the external level, they will also influence the schema we choose at the conceptual and internal levels.

12.1.1 Performance Estimation

The first thing we do is try to get figures on the expected use of the database. This is difficult. There is much uncertainty. Hence, we are likely to go wrong, and this emphasises the advantage of using a flexible DBMS (with high data independence) so that we can make changes when the system is actually operational.

We can represent the performance estimates as a *transaction matrix*. This is a table that shows the database access requirements of applications. Columns of the table represent particular 'transactions', i.e. the processing requirements that come from the logical process model, and the rows represent data fields. Each transaction (column) has an associated frequency (e.g. 50 per week) which gives some idea of the importance of that transaction. (There are other factors that affect a transaction's importance. For example, the efficiency of a batch program is likely to be less important than that of an interactive transaction.)

The entries in the cells of the matrix show the operation being performed: if field i is manipulated by transaction j, then in cell (i, j) there will a one-letter code indicating the nature of the manipulation, e.g.:

U: field i is updated (inserted, modified, deleted) by transaction j;

R: field i is retrieved (but not updated) by transaction j;

C: field i is computed by transaction j.

Let's look at an example using a students and courses database.

Suppose our logical process model tells us that the following are the significant transactions with their expected frequencies:

T1 — insert a new student (100 per year);

T2 — compute new grade-to-date (300 per year);

T3 — insert enrollment data (3000 per year);

T4 — look up course details (400 per year);

T5 — look up enrollments for a course (80 per year);

T6 — look up the courses a student takes (150 per year);

(Other transaction are insignificant, e.g. there will be only a handful of insert a new course transactions per year.)

We then draw up rough queries, bits of pseudo-SQL or whatever to get an idea of what bits of the database each transaction will affect. I stress that you do not write the actual code (you can't: the schema hasn't been finalised yet; after all, finalising the schema is what we're presently trying to do). Rather, you just get a feel for what bits of what tables in the simplistic schema are affected.

For example:

```
T1 ---  INSERT INTO S
         VALUES ( an SNo and an SName )
        (Grade can default to null)

T2 --- UPDATE S
        SET Grade-to-date = Grade-to-date +
            (SELECT SUM(ExamMark) FROM SC
             WHERE SC.SNo = some specified SNo)
        WHERE S.SNo = the specified SNo

T3 --- INSERT INTO SC
        VALUES ( an SNo and a Ccode )
       (ExamMark can default to null)

T4 --- SELECT * FROM C

T5 --- SELECT SNo, SName FROM SC, S
        WHERE SC.SNo = S.SNo AND
              SC.Ccode = some specified Ccode

T6 --- SELECT Ccode FROM SC
        WHERE SNo = some specified SNo
```

We are now in a position to draw up the matrix:

	T1 (100)	T2 (300)	T3 (3000)	T4 (400)	T5 (80)	T6 (150)
S.SNo	U	R			R	
S.SName	U				R	
S.Grade-to-date	U	C				
C.Ccode				R		
C.CName				R		
C.Year				R		
SC.SNo		R	U		R	R
SC.Ccode			U		R	R
SC.ExamMark		R	U			

Now we look at what you might do to ensure that frequent, important transactions will not take too long to carry out.

12.1.2 Internal Level Considerations

One way of meeting the performance requirements might be to make use of as many of the facilities of the internal level as possible.

In R:BASE and many other RDBMSs we have see that this boils down to deciding which fields should be indexed. As a guideline, the fields to choose are those that have nice data types (e.g. integer rather than 25 alphanumeric characters), are preferably (although not necessarily) unique and not null, have a corresponding R in a cell of the transaction matrix for a transaction whose frequency is high, preferably not having a corresponding U in a cell of the matrix for a transaction whose frequency is high (because indexes slow update), and/or are equated to another field in a join.

It is difficult to give any advice about other systems: the advice would depend on what storage structures they offer and the relative efficiencies of these structures.

It is also difficult for you to predict whether indexing a field will give you the performance improvement that you want or whether the more radical measures proposed in subsequent subsections are needed. You can only gauge this by gaining experience of your DBMS and thus knowing what sort of improvement your are likely to achieve. To give you an example, the addition of an index to a column of an R:BASE table that is heavily used in queries but not much used in updates meant that a query which was taking 2 minutes and 15 seconds took only 1 minute and 15 seconds, and indexing all key fields (thus allowing R:BASE to decide which of several indexes to make use of) reduced evaluation time to just under 20 seconds.[1]

[1]In fact, the query was to work out what exam papers 2nd year Computer Science

Other things to do include not using the code generated by the applications generator but hand-crafting something more efficient yourself.

12.1.3 Conceptual Level Considerations

If it appears that features of the internal level cannot give you the performance you want, then you may want to consider ways of changing the schema ('the conceptual schema') itself. There are obviously risks involved in this because it makes your schema less like the models you built up, and thus make it less like the real world that it is trying to capture.

The transaction matrix allows us to identify a number of possible changes.

Compose small records

A transaction might have a lot of entries in cells that correspond to different tables. This may imply that carrying out the processing for this transaction involves lots of joins of tables. We already know that joins are expensive to perform. An example of when this might occur is if your conceptual data model had a lot of subtypes with few distinguishing properties in them. This leads to lots of little relations in the database that need to be joined to get useful information. The solution is to combine these tables into a single table with more columns.

Here are two examples in more detail.

Suppose we have the following tables:

UNIVERSITY_MEMBERS(**Person_no:** PName, Date_of_birth, Position)
STUDENTS(**Person_no:** Grade_to_date)
LECTURERS(**Person_no:** Salary)

We may often need to join STUDENTS (or LECTURERS) with UNIVERSITY_MEMBERS because when a user wants to see a student's marks (or a lecturer's salary), the user will also want to see the name of the student (or lecturer). We can avoid the need for these joins by replacing these three tables with one single table:

UNIVERSITY_MEMBERS(**Person_no:** PName, Date_of_birth, Position, Grade_to_date, Salary)

students would be sitting and the first column I indexed was the student number in the STUDENTS table.

with Grade_to_date being null if Position = ‘Lecturer’ and Salary being null if Position = ‘Student’.

Another example concerns foreign keys:

UNIVERSITY_MEMBERS(**Person_no:** PName, Dept_code)
DEPARTMENTS(**Dept_code:** Full_dept_name, Size)

Suppose that users who retrieve university members prefer to see the full department name displayed on the screen for each person’s data that they retrieve. We must join the two tables. To avoid this, we can change the tables as follows:

UINVERSITY_MEMBERS(**Person_no:** PName, Dept_code, Dept_full_name)
DEPARTMENTS(**Dept_code:** Full_dept_name, Size)

Now the join is not needed. But note what we have done here: we have *denormalised* the database. Dept_full_name is only transitively functionally dependent on Person_no. Thus, this improvement in retrieval times is bought at the expense of more work in updates to achieve consistency.

Decomposing large tables

In the case of tables that have lots of columns, it is probable that only a particular subset of them will be needed in a particular transaction. You can spot this in the transaction matrix: if one transaction uses one subset of the table’s columns and another transaction uses a nearly disjoint different subset. Time will be wasted moving lots of useless data to/from disk.

The solution is to split the table into smaller tables. Here are two examples.

Consider:

COURSES(**Course_code:** Course_name, Lecturer_no, Course_desc)

If Course_desc is a very large text field and is not used by most transactions, then you can perform a *vertical decomposition*:

COURSES(**Course_code:** Course_name, Lecturer_no)
COURSE_DESCS(**Course_code:** Course_desc)

Most queries will only need to access the much smaller COURSES table. The cost is that for those rarer transactions that need all fields, a join is now needed.

Consider a different example:

STUDENTS(**SNo:** SName, Degree_scheme)

If most times we only want to manipulate students from one particular year group, then we might perform a *horizontal decomposition*:

YR1_STUDENTS(**SNo:** SName, Degree_scheme)
YR2_STUDENTS(**SNo:** SName, Degree_scheme)
YR3_STUDENTS(**SNo:** SName, Degree_scheme)
YR4_STUDENTS(**SNo:** SName, Degree_scheme)

Now dealing with, e.g., third year options will work more efficiently because it need only look at one of the tables. If some rare processing must be done on *all* students, then the tables can be unioned.

Invent Simple Fields

Suppose a table presently has as its key a large number of fields or a large cumbersome field, as here:

COURSES(**Course_name:** Lecturer_no, Course_desc)

You might invent a different field that can more readily be used as an index field, for example, a three-letter course code:

COURSES(**Course_code:** Course_name, Lecturer_no, Course_desc)

To compute or Not to compute?

Any field in the transaction matrix which is computed from other fields needs careful consideration. Should it appear as a column in the table or should it not? For example, should Grade_to_date (which is the sum of a student's exam marks for different courses taken) be stored explicitly or not?

If you make it a column of the table and therefore explicitly store it, then you get fast retrieval, but update is slower because, e.g. any change to a student's exam marks must also be reflected in his or her grade to date. If you do not make it a column of the table, you improve update times but anyone wanting to retrieve the value of a student's grade to date will have to wait while it is computed afresh each time.

Summary

Remember that we are making these changes to improve the performance of the most frequent transactions. However, all these changes have costs associated with them (e.g. duplication of data which costs us in storage space and update time, and/or reduced performance of the less frequent transactions).

In my opinion it is right to start off with a normalised database, do some performance analysis and then decide what changes to make. This will make clear in what ways the new schema deviates from the 'norm' and how to compensate for this.

12.1.4 Other Considerations

There remain yet further ways of getting improved performance, which we will look at briefly.

Distributed Databases

Up to now, the course has focussed on databases that are physically stored in one place. A *distributed database system* is one that may span several sites. DBMSs that can handle databases that are distributed in this way do exist. (Oracle sells a distributed version of its system, for example.) But distributed databases are expensive and the capabilities of the commercial systems lag a long way behind the theory.

> "Although we have not introduced this issue earlier in the development process, in practice it would be unusual to have proceeded as far as this without having considered it, at least in principle."[Whittington 1988, p.362]

However, if you considered a distributed database earlier and rejected it, you may want to reconsider it now to see if it can help with your performance needs. Distributed databases can improve the performance of some parts of your system because:

- each node in the network will be holding only a fragment of the database; this reduction in data volume will make searching faster;
- by having many processing nodes, you have an overall increase in computing power; what's more you have the capability to process transactions in parallel;

- the storage of data and the storage and execution of programs will take place closer to the end-user, so transmission costs for remote users will be reduced.

There's an element of wishful thinking in this: clearly if the data that you need is *not* available at your local node, then the execution time of your transaction may take even longer than ever before. Thus the quality of the *fragmentation scheme* used is critical. The fragmentation scheme determines what parts of the data go where. You might use some form of vertical decomposition, horizontal decomposition or even data replication.

Note that distributed databases have advantages over and beyond their possible performance advantages: they may be more resilient to failure (one part of the system failing need not stop everybody from working), they may be easily scaled by the addition of further sites, etc. Their disadvantages are due to their complexity: cost of purchase, cost of running, need for specialised staff, etc.

Database Machines

Another option, which is presently only worth considering for some specialised applications, is the use of hardware dedicated to database processing. In its simplest form, this is just an extension of the file server idea: a machine which is dedicated to handling all the file management or, in our case, all the database processing. By off-loading DBMS work from other machines, you can increase the throughput on those machines. It is questionable whether query response time will be improved though, due to the increased transmission overheads. An obvious extension to this idea though is the use of special hardware to act as the database server.

The basic idea of this special hardware is usually to put a bit of processing power into the disk drive. In ordinary systems, data is read from disk and processed in main memory, e.g. to see if it meets a restrict condition. In these special search engines, a hardware-assisted filtering mechanism in the disk drive reduces the amount of data that is transferred to main memory. Since disk access and the volumes of data that get transferred between the disk and main memory are the bottlenecks in any DBMS, this solution can dramatically improve performance.

An example of a commercially available search engine is the ICL Content Addressable File Store (CAFS). Basically, CAFS has a hardware filter on the data transmission channel between a normal disk controller and the host computer. When records that satisfy certain conditions are needed, the DBMS loads the filter hardware with the selection criteria. As records flow from the disk, they pass through the filter. Only those records that satisfy the selection criteria are passed on to the host computer.

It has to be said that while there are a number of special hardware devices that are commercially available, they are not in widespread use.

Other Types of DBMS

In this course we have concentrated on relational DBMSs. Time has not allowed us to look at other types but we mention them here briefly.

Before relational databases came into widespread use there were two other main types of database system: *hierarchical* databases and *network* databases. Both are still quite common. IBM still sells a hierarchic database (IMS), and, I believe, that this is still the single most widely used DBMS in the world. Many companies still supply network databases (although they often come with additional relational-like facilities these days).

By comparison with relational databases, hierarchical and network databases are much closer to the internal level. You tend to think of network databases, for example, as large numbers of multi-ring files. Relationships between records are handled by pointer structures, rather than by the use of keys and foreign keys that we saw in relational databases. They also tend to offer a wider range of storage structures, e.g. network databases typically supplement multi-rings with options to have sequential files, indexed files, indexed-sequential files and hashed files.

These types of database tend not to offer interactive query languages. Rather, programmers write lots of code to produce the reports your organisation needs. The code is usually written in a version of COBOL, extended with database access commands. These database access commands give full control to the programmer in deciding how the data will be retrieved, e.g. whether to use a hash key if it is available, whether to trace the pointers in a forwards or backwards direction (if both are available), etc. To retrieve a set of records (which is easy in RDBMS languages) requires writing appropriate loop statements which perform repeated accesses of the database. It is often said that getting data out of one of these databases is a question of *navigating* around the pointers.

The consequence of all these differences is that these old-fashioned databases can be a lot more efficient than RDBMSs. They place full control in the hands of the programmer to decide how to get the data he or she wants. Thus, a skilled programmer will work out the most efficient route to the required data and write his or her program accordingly. Compare this with a RDBMS such as R:BASE where all the low-level details are hidden from you: you cannot tell the system to retrieve the data by 'this' route rather than by 'that' route; you are pretty much at the mercy of the people who wrote the RDBMS.

Thus, if meeting performance requirements is a real problem for you, you might consider one of these more old-fashioned systems.

Other Types of System

In a similar vein to the previous topic, if meeting performance requirements is really a problem, then you might consider using, for example, a file-based system. Under the right circumstances, these might be highly efficient for certain types of processing. Of course, such a decision does take you full circle, back to Chapter 1, where we indicated the problems with such systems.

Another possibility is to consider writing what amounts to your own DBMS, highly tailored to your application. This is not as outlandish as it sounds. Specialised applications, such as airline booking systems, may have such unusual requirements (e.g. for the number of concurrent users, the degree of remote usage, etc.) that no general purpose off-the-shelf software could possibly do the job well enough. In such cases, writing your own DBMS may be the only solution.

12.1.5 Summary

The implication of what we have been saying is that you do some performance estimation and then make various decisions about what indexes to use, what changes to make to the schema, etc. Actually it is very hard to do this in advance. To some extent you are only going to know your performance requirements when the system is up and running and people are complaining about it.

What's more, demands on the database will evolve, causing unpredictable changes in the pattern of usage of the system.

This is why having a system with a high degree of data independence is crucial. For example, it is easy to add new indexes (or remove indexes) from columns in R:BASE and other RDBMS without affecting any of the forms, reports, or applications (none of them need recompiling). This can make it relatively easy to adapt to changing circumstances.

12.2 Database is GO!

Well, that's about it. You've worked out the final form of your schema, what indexes there are, and you've typed that in; you've implemented forms, reports and applications; you've coded and tested programs. There may be some DBMS parameters that need defining, e.g. buffer sizes, the logging frequency for recovery and for audit.

Finally, you take on any existing data which you want to have in the system. Note that this can be a quite enormous task.

Then, as the title of this section says, database is GO!

12.3 Database Administration

We end by pointing out that even after database is GO, the users are wetting themselves with joy at using your superb database system, and you and the other database designers are down the pub, there is still a technical role that needs to be performed. This is the role of the *database administrator* (DBA). This is an important job carrying a great deal of responsibility.

The DBA has overall control of the database. Some of his or her functions might have taken place during design but have an ongoing side to them as well, e.g.:

- deciding the information content of the database
- deciding the storage structures and access strategies
- liaising with users
- defining security and integrity checks
- defining a strategy for backup and recovery (and enacting it as necessary)
- monitoring performance and responding to changing requirements
- observing legal requirements (e.g. Data Protection Act, 1984)
- defining and enforcing standards

This is a difficult role since it involves making trade-offs. The DBA must be a politician so he or she can deal with users who have conflicting demands, but he or she must also be a technocrat since he or she must manage a complex piece of software.