Databases

A C Norman, Lent Term 1995

These notes prepared by A C Norman and N H M Caldwell, 1995

1 Introduction: What is a database?

First let me introduce these lecture notes. They are not a complete presentation of all the material covered in the lectures — rather they provide an overview of the topics around which the course is based. The textbooks listed at the end should be consulted to find complete and accurate technical discussions of all these issues. However some of the textbooks are sturdy 800+ page volumes, and so obviously they contain much more than the lecture course can possibly cover: these notes should help you sort out which sections are vital and which you should read just for interest.

The term "database" may be used loosely as a term for almost any body of information kept on a computer. Here the emphasis will be on cases that tend to have most of the following properties:

Persistent data: The information must be preserved beyond the end of the run of just one program. Thus I will not view the contents of a computer's memory while it is running a program to be (of itself) a database unless perhaps steps have been taken to keep that computer running and reacting over a very extended period of time;

Multiple uses: Data become most interesting when related information is collected together and multiple applications share access to it.

Scale: Utterly tiny amounts of information do not call for database technology, so we will often be thinking about substantial amounts of information;

Enterprise Data: Frequently databases hold information that is central to the operation of some organisation ("mission critical" in software engineering parlance), and anything that compromises either the accuracy of the data or smooth access to it would have severe repercussions. Thus reliability becomes a serious issue;

Modelling the Real World: The data stored will be a representation of some information that has a relevance and reality outside the computer. Thus (usually) the storage of temporary results computed during some lengthy mathematical calculation will not count;

Security Matters: Information in many databases will be sensitive in one way or another, so although it is important to make the intended patterns of access to data easy, preventing improper access is usually also vital;

Flexibility is needed: All the other issues listed here imply that use of any real database will evolve over time, and so abstractions must be used so that incidental implementation decisions do not inhibit this.

Coping with these demands will naturally lead to costs: in the size and complexity of the database management software, in the design effort needed to establish a database and set up ways in which controlled subsets of the information stored can be accessed, and in the machine resources needed to support it all. So perhaps the first suggestion from this course is that if you have an application with simple data, structured in a way that is not going to change, without need for multiple users or security or with severe constraints on the absolute delay that can be tolerated for one operation, then avoid database technology and store your information in simple files with some clever index structure. But as soon as you start to find multiple uses for (subsets of) your data, and multiple concurrent users need support, it is time to move to use of a data base management package.

When considering the costs of using database technology (and that will often need to include the cost of learning how to apply it) it is perhaps instructive to think about the value of data. I will give a simple (perhaps over-simple) calculation that suggests that perhaps it is quite high. First suppose that the real-world data exists somewhere outside any computer, that it is either textual or numeric, and that there is no obvious way of setting up machines to scan it directly. Given these assumptions I will just look at the cost of having somebody sit at a keyboard typing the data into a computer. I will not make any allowance for collecting the information in the first place, proof-reading it once it has been typed, or editing it later to correct the mistakes.

A respectable typing rate is around 60 words per minute, at around five letters per average word, so I will estimate (coarsely) that data can be presented at five bytes per second. This is 18 Kbytes per hour, which figure might be accurate to within a factor of two. Continuing this calculation leads to the conclusion that a simple floppy disc (that costs 25p when bought in bulk) takes two weeks to fill even if we have somebody typing at fairly high speed full time. Rounding all estimates fairly vigorously, I will count this as having cost £500, or 2000 times the cost of the floppy disc.

The amount of information stored on a single floppy disc is still pretty small by today's standards. Consider a CD-ROM, which can be stamped out in bulk for not much more than a pound, or where you can have a version that you can record for yourself for a unit cost of under £10. If I consider such a disc totally filled (around 600 Mbytes) and again suppose that this is data that I have had to get typed in at some stage, the cost of the keyboarding stage alone is around £200,000. Even if I store the data on a hard disc (as of early 1995 a suitable one could be bought for around £160) there is a **huge** difference between the cost of the computer hardware involved in storage and the value of the data itself.

Of course sometimes the information in a database can be collected in some other way that might be more efficient e.g. some supermarkets will use bar-code scanners at their checkouts to collect information. But in general the cost of col-

lecting data will increase as the need for reliability increases (spotting errors and making corrections is time consuming and expensive), and the value (as distinct from the cost) of data will be higher yet. The message I want to get across is that even medium sized databases may represent significant investments, and large ones have to be recognised as enormously valuable - and hence worth treating with care.

Date[1] lists as some of the potential benefits of use of a DBMS:

Control of redundancy: If all your data is stored together you may avoid the need for multiple copies of the same information. While this may save disc space a much more important issue is that it removes the possibility that the multiple copies get out of step with one another.

Access Control: DBMS software should be able to support some form of authentication, and only allow access to data to users or applications that are properly registered.

Persistence: Some modern (often experimental!) database managers are moving towards being able to capture and preserve almost any data structure that can be created within a program, so that it can be re-loaded and used again on another day.

Integrity Constraints: A severe problem with real data is that of errors — for instance those caused by the original incorrect input of text or values. A DBMS must make it easy to specify various consistency checks that can help flush out erroneous entries and which can prevent bad data from corrupting the results of too many queries.

Backup and Recovery: Database software should also provide coherent strategies to allow the reconstruction of information after almost arbitrary hardware (and software) crashes.

Economy of Scale: Use of a database may focus explicit attention on the support of uses of that data, and make it possible to concentrate the people with relevant in-depth technical knowledge in a small group rather than letting distributed amateurs deal with each separate application.

This course is **not** going to be about the detailed procedures for laying out data on discs or algorithms for searching it in clever and fast ways. The low level details of database implementation are of course fun and complicated. However a first course on databases should assume that suitable database management packages already exist: it should worry about how raw real-world data can be organised in ways that avoid unnecessary hardware dependence and risk of error, and

talk about how the data can then be mapped onto the major models that database managers support.

2 Applications of Databases

Database technology is pervasive wherever computers are used in support of large enough projects. However the fact that the **full** problems associated with them do not arise until there are multiple uses (and possibly users) of the data, or until there is a large amount of information to be stored will mean that most students (and indeed a great many academics) will not have personal and direct experience of them. The uses of a database may have a strong influence on how it should be organised and implemented, so here are a few sample applications, together with brief commentary on the aspects of them that are likely to be most challenging:

Bank Customer Records: You would presumably be upset if anything caused your bank to lose track of how much money you have lodged with them (or less upset if they forgot about your overdraft). Banks and Building Societies will often have very large databases, with a large volume of updates, and a need for real-time response to some queries (such as checks to determine if enough money is available to make it reasonable to withdraw cash from a hole-in-the-wall automated teller. There are very heavy peak periods for bank transactions. A bank will have a large number of local branches, and one might like to be able to take advantage of the fact that **most** transactions involving any one customer involve activity quite close to that branch. A key issue is the high volume of updates to the data.

Retail Business: A retail business (consider a supermarket chain) will need to keep records that relate to stock levels, suppliers, orders, sales volumes, VAT, payroll, shop repairs and its advertising campaigns. At least! In different parts of the company quite different sorts of access to this information will be required — individual departments should only be able to access the information relevant to their work. A key issue is the rich collection of ways in which information has multiple uses.

Computer Aided Manufacture: The manufacture of almost any reasonably complicated object will involve a range of computer activity. This may involve engineering design, with elaborate computer models and simulations of the object, through to the keeping of detailed records of suppliers of components and failure rates of units that involve those components. In most cases the object being made will go through a number of different revisions or models, and there will be financial as well as directly technical information

to be kept. In some cases this may require the support of very complicated forms of data in a database — much more than records that just count the number of tins of baked beans sold.

Telephone-based insurance companies: will need to be able to retrieve customer records whenever you phone them. I include this to suggest the extra problem that can arise if there are real-time constraints on how queries to the database must be satisfied.

Historical records: These may provide an example of a sort of database that is, by its nature, not subject to update (unless of course an archaeologist unearths new information). Being read-only might in many ways simplify database management, but in some applications this might be balanced out by the need to perform very elaborate sorts of search to find the information that is wanted, for instance in an attempt to trace the life of an individual by piecing together information from many different sorts of record. Note particularly that a retrieval from such a database may involve **much** more effort than just using a single word as a key that points to the required information.

It should be clear from the above that databases can be very varied. There will be major parameters that influence how they should be implemented. These include the ratio of updates to inspect-only accesses (enquiries), the degree of uniformity and simplicity in the data stores, whether the database is enormous or just large and whether queries will all fall into a small number of regular patterns.

3 Problems in database design and use

The real concern of this section is more to explain what issues about databases are most tricky and critical. Obviously performance will be one difficult issue. Arranging for robustness in the face of potentially fallible hardware will raise other problems. But the really critical challenges that face database administrators are at a quite different level:

1. If the same information is extracted from the database twice, using different access routes, is it certain that the results will match?

2. Is it possible to prevent "obviously" wrong information from ending up in the database?

3. If it is necessary to reconfigure the database — either to accommodate changed hardware or to allow for new styles of use of the data — can existing uses of it continue uninterrupted?

4. Can the database be documented well enough that it will be possible to support and extend its structure many years after its initial installation.

5. Will the formal database structure that has been set up form an adequate model for the real-world objects or activities that it is supposed to record information about?

These challenges relate more to controlling the structure of the database than to any details of its algorithmic implementation.

4 Separation of concerns: the ANSI/SPARC architecture

A database can be considered from three major perspectives, and keeping the various issues that arise neatly attached to one of these is referred to as "using the ANSI/SPARC (three-schema) architecture". The three views considered are:

1. An **internal** one, which is concerned with physical disc drives, the layout of data onto blocks, detailed indexing procedures and in general the interpretation of how data is mapped onto a real computer. A *schema* for this would be a document describing all relevant internal-level details of how the database in question is configured.

2. The **conceptual level** also has a schema to document its view of the data. This time the idea is to suppose that low level details have been handled elsewhere, and concentrate on specifying the types of information to be stored, the relationships that the database may be called upon to handle and what operations will be provided for accessing and updating information. It is the job of a conceptual schema to describe the complete structure of a database, so when setting one up it will be necessary to understand **all** the uses that the data will be put to.

3. At any one time a database can be described by just one internal and one conceptual schema. But there can then be several **external** models of the data. In general there will be one external schema for each major use that the database has, and this must document just the parts of the information that is relevant to that use. Note that this does not at all mean that an external schema will be just a subset of the conceptual one — data that it exposes may need to be retrieved indirectly from the conceptual model, and sometimes information will need re-arranging or filtering before it is made available to the (external) user.

Before continuing it is perhaps necessary to stress that the "customer" of a database will, for the purposes of this course, always be an application program. In the simplest cases this program will just sit there allowing a user to type in queries, it will forward these to the database proper, receive a result and display it for the user. But in more realistic cases the program will make several (perhaps many) database queries in response to one interaction with its human user, and will update records in the database as well as just read from them. The parts of this program that perform calculations, run communications networks and organise convenient interactive user interfaces are **not** of concern to a database course. The external schema of a database explains how this program talks to the more central database manager.

The three levels all describe the same database, but at least in principle different people design and work with each. A major goal of database design, and one that has made the ANSI/SPARC architecture very successful, is referred to as **data independence**. This has two sub-flavours:

Physical: If the conceptual schema of a database can be kept separated from the internal one, it *should* be possible to change the internal schema without disrupting anything else. A circumstance where this becomes a vital practical concern is when the computer on which the database is mounted has its configuration changed, or when new low level data management algorithms are invented. There are also possibilities that following measurements on the actual pattern of use of a database the administrator could want to rearrange the internal schema so that especially frequently performed operations take less time. Changing the internal schema will very often alter the performance of a database, but we do not want that to make any previously supported styles of access to become totally unavailable.

Logical: Logical data independence represents the ideal state where conceptual and external views are well separated, so that changes in the conceptual schema do not cause existing application programs (all of which use one of the external schemas) to fail. It can be useful to change the conceptual schema either to add new sorts of information to the system, or to add extra constraints that police consistency conditions that had previously been ignored, or to remove or rearrange the storage of information that is now no longer frequently accessed.

The main consequence of the ANSI/SPARC architecture and its application to data independence is that there have to be explicit mappings that transform queries from the language used at one level to that needed by the next.

A secondary matter (well, historically it was not secondary at all!) is that a strict adherence to data independence makes it hard for one of the higher level

schemas to provide the lower level ones with hints that might make a big difference to performance. For instance if following the tenets of data independent design, a conceptual schema will utterly ignore such issues as the size of sectors or tracks on the discs that will be used at the internal level, and so will sometimes specify records that would (in a naive implementation) grate against the hardware. Bigger and better database management software systems can now deduce or reconstruct most of what they need to deliver reasonable performance, and because many databases will last for many years, designs optimised for one generation of equipment may look rather silly next year when new computers and discs are installed.

The conceptual level of a database is defined in some notation that will be referred to as a **Data Definition Language** or **DDL**. The term **Storage Definition Language** or **SDL** relates to a notation used to define the internal schema. The ways that external users see the data are sometimes known as **views**, and one can have a **VDL** to describe one. At the external level there will also need to be a **Data Manipulation Language** (**DML**) in which operations on the data are expressed. Often the DML will be embedded within some ordinary programming language, either using language-extension keywords, wrapped up as a collection of library routines, or supported by some way in which ordinary programs can exchange messages with the Database Manager (DBMS).

5 Databases and Reality

A quite unexpected problem with databases is that it is usually not at all easy to design them so that they are proper mirrors of the real world. While starting to plan this course I came across a concrete example of such a problem, and one that relates to what is really quite a small amount of data. Trinity College was keeping separate records relating to students in several different offices, and it turned out that even after putting in real work chasing oddities it proved impossible to discover exactly how many undergraduates were around! The numbers as shown by the different lists differed by two or three, and the differences seemed not to be simple to reconcile. Some of the difficulty is that the term “undergraduate” ends up not being quite precise enough:

1. The Admissions office might count the number of applicants who had been made offers, had achieved the relevant grades and had not indicated that they were going to withdraw;

2. The Tutorial offices might count exam entries;

3. The Junior Bursar looks at students who need accommodation in the undergraduate room ballot;

4. The College office needs to report how many students are getting grants from their local authorities as undergraduates, and how many are being funded in some other way.

In each case there can be odd or marginal cases. These include students who are away from Cambridge for a year hoping to recover from illness. Also ones studying abroad for a year on some exchange scheme. Some students drop out completely, but at odd times of year, for academic, financial or other reasons. Exam entries get muddled, and it is even the case that a few students will be entered for two sets of examinations in different subjects in one year. In even a medium sized College one can get two different students each called J Smith and both reading the same subject — and thereby causing confusion. I have known of students changing their names (by deed poll) partway through their stay here, further muddling things. Frequently in curious cases the exact status of a maybe-student will be unclear for quite a long time, and it will not be at all clear cut when their status in any particular batch of records should be updated.

Of course the funny cases are each individually fairly uncommon, but with about 600 undergraduates (hah — for some purposes a Part III mathematician is an undergraduate too…, and where shall we list affiliated students?) some uncertainty remains.

A further indication of how hard it is to collect nicely structured data that nevertheless ties in with the real world is the large incidence of grossly inappropriate questions on forms that have to be completed. They often indicate assumptions that the designer of the form made when deciding what to ask, and how such assumptions can fail. Kent's book[3] is by now pretty old, and its typography is a reminder of a few unhappy years when some authors were invited to hand golf-ball-typewriter typed final copy to their publishers. But it explores at length the relationship between the tidy virtual world of electronically stored data and the reality that it is intended to model.

The message that this gives us is that an unthinking acceptance of words that are in common use as adequate descriptions for classes of entities to be installed in databases can lead to trouble: more careful thought is needed.

6 Entities and Relationships

Two vital words associated with database design are **entity** and **relation**[1]. An entity is just any object that a database will want to include reference to. Date gives an example where a database will contain entities that are suppliers, projects,

[1]Later on we will cover a database strategy that is known as the *relational model*, but for now the term "relation" is not implying that we are using that particular approach.

parts, warehouses, locations, employees and departments. When starting to design a database identifying the entities that are needed will be a very early step. A relation links several entities, and although many relations concern themselves with just pairs of entities it is perfectly possible to have higher order relations.

The purpose of identifying entities and relations is to gain some degree of control over the semantics of the database that is eventually designed. Note that when identifying entities and relations present in some scenario that is to be modelled by a database you are concerning yourself with the conceptual level and are not pre-judging any internal details.

When listing the sorts of entities that will be used it will also be useful to think about their types (for instance some numeric data will represent amounts of money, other data that may end up stored using exactly the same numeric representation may indicate size rather than value) and to decide what constraints apply (so the age of a person might reasonably be constrained to lie between 0 and 969 (the age that Methuselah is reported as having lasted to)).

7 The network model

The "network model" for databases is hardly even mentioned in the latest edition of Date's book, but you can read about it in Elmasri and Navathe[2]. Or if you go back to a 5th edition of Date you will find a lengthy appendix about it, with yet earlier editions giving it even greater prominence. Today there is much more emphasis on "relational" databases (discussed later). The network model emerged from the CODASYL (Conference on Data System Languages) Data Base Task Group (DBTG) in 1971. This is (in computer terms) an enormously long time ago, so why still discuss it? There are two major reasons:

1. Databases are often very long lasting entities. Many large databases will last **much** longer than the particular computer systems on which they are mounted. Thus there will be databases initially established in the 1970s that are still in use, and where altering the associated models will be very hard. The existence of such "legacy" systems means that it is still useful to understand the previous major generation of database techniques.

2. Understanding the places where the network model was awkward to use can help justify the relational replacement. In the same way it will be necessary to study relational databases even when they have been replaced by the next big 20-year wave of ideas (which *may* be object orientated databases).

One of the most important examples of a network-based DBMS is a product called IDMS, and some of this section may relate specifically to that, while other

remarks will be true of network-based databases in general. It does not support the full three levels of the ANSI/SPARC suggestion, but still tries to keep some separation of concerns. It provides three components:

1. A "Schema DDL", which is used to define the global structure of the material that is to be stored. It slightly mixes together concerns that ANSI/SPARC separate into their conceptual and internal schemas;

2. A "Sub-Schema DDL" that defines external views of the data (i.e. ones that will be used by individual applications);

3. A DML (data manipulation language) which is a set of record-by-record operations that work on the structures defined by the two DDLs. This will be embedded in some existing programming language — and for most real database applications over the time-frame where this style of work was most common, this would mean COBOL. (Although a FORTRAN DML is also supported through the use of a database library to extend the language.) The access is batch oriented and not all convenient for *ad hoc* queries.

The presentation here will **not** reveal too much about how network databases are mapped onto physical discs, or how multiple simultaneous users are supported, or what is done to ensure recovery following hardware failure. I will provide a logical explanation of what is done. The entire database will be made up out of *records* and *links*. There will be types associated with both records and links. Each link type will associate one record from a *parent* (or *owner*) type with an ordered collection of records from a *child* (or *member*) type. Any particular record of the parent type has exactly one link (and thus one associated set of child records) associated with it. Each child record is in at most one link (of specified link type). The individual records will comprise fields that contain whatever strings, numbers or other material the database is concerning itself with. The detailed representation of links is a business for the DBMS to worry itself with, not for us. Even if, internal to the database, some quite different indexing scheme is used, it is normal (and convenient) to think of a parent record containing a pointer to its first child, a pointer chain following through there to further children, and ending by referring back to the parent. This forms a loop or ring. We can therefore reasonably expect that given an instance of a parent, we can find the first child, given a child we can find the next child (or have a LAST indicator returned), and given a child we can find its parent, and this is indeed the case.

A somewhat more detailed presentation of the various components of the CODASYL DBTG proposals will now be given (but some of the goriness has been suppressed).

7.1 Schema DDL

There are four main parts to the Schema DDL:

- Schema Entry
- Area Entry
- Record Entry
  - Record Subentry
  - Data Subentry
- Set Entry
  - Set Subentry
  - Member Subentry

The Schema Entry NAMES the data model (database), and can also specify procedures to be called for the validation of access rights (logon procedure, password protection), and on error. Thus there is overall provision for privacy, security and integrity in the proposals.

The Area Entry describes the LOGICAL data storage regions (thus enabling records to be stored on different discs dependent on some attribute value), and specifies procedures to be called to effect OPEN and CLOSE as well as when errors occur. Access is specified on OPENing and is usually EXCLUSIVE for UPDATE and PROTECTED for RETRIEVAL. Areas can also be specified as TEMPORARY in order to define a scratchpad which will persist only from OPEN to CLOSE.

In the Record Entry, there must be one entry for each RECORD TYPE. The Record Subentry is about accessing the record as a whole and will define the type name, with possibly KEY information including control over duplicates and ordering, and a reference to the AREA or AREAs on which the record occurrences are to be stored. An important clause covers the placement of record occurrences when they are created — LOCATION MODE IS — and this can be set to DIRECT (near a given DBKEY — a physical address), CALC (e.g. hash, index), VIA set-name SET (logical level version of direct placement) or SYSTEM-DEFAULT (up to the DBMS). This clause mixes concerns of the internal level with the conceptual level, and is contrary to the ideals of data independence. The Data Subentry gives the 'record layout' by FIELD, defining field-name, type and length attribute by attribute for each record type.

The Set Entry defines the relationships in terms of parent and child records that exist in the data model. For each association (or set) of child records with parent

records, an OWNER type must be specified, and for each SET occurrence (each ring structure) the ordering of MEMBERs and the insertion strategy must be specified. For most sets, MEMBER types must be defined in the Member Subentry to determine whether for each MEMBER type membership in a set is MANDATORY (must have a proper OWNER), or OPTIONAL. This Subentry will also give details of KEYs for ordering set occurrences, locating member records etc. If membership is MANDATORY, then the actual insertion may be MANUAL or AUTOMATIC and in the latter case the occurrence of the set to which the newly created MEMBER record is to be appended is explicitly defined.

7.2 The Sub-Schema DDL

On the whole, this provides a restricted view of the data model. The structure of the sub-schema definition closely follows the pattern set in the Schema DDL. However AREAs are now called REALMs. It is also the case that not all SET and RECORD types need be visible (but nothing new can be added). The Data Subentry for records can now specify changes of data type, calculate new units etc. New privacy controls may be introduced at sub-schema, realm, record or set level.

7.3 Data Manipulation Language

The access to the database is always via some subschema. An application programmer will compile in an environment containing subschema information, and will be able to create work areas for each record type with the same names and components as are specified in the subschema through using the INVOKE <subschema> statement.

The security of data as well as the control of concurrent access is effected by the OPEN and CLOSE "verbs" (COBOL terminology for a command), and OPEN can specify either the SETs (high-level) or REALMs (more physical) that are to be manipulated. Query processing would be run as 'PROTECTED RETRIEVAL' whereas update processing would be made via 'EXCLUSIVE UPDATE'. This is not enforced and is left to the discretion of the application programmer.

Once a process has OPENed SETs or REALMs, data access is possible. Records of given type are read into the work areas defined by INVOKE. The program has available the following CURRENT records:

- CRU — the Current-of-Run Unit which is the record currently being looked at (regardless of type)
- CURRENT of each record TYPE — the last record of each specific type to be looked at

- CURRENT of each SET — which is the OWNER or MEMBER last looked at ('touched') in that set
- CURRENT of each AREA/REALM

The programmer "navigates" around the database transferring attention from record to record by RECORD SELECTION EXPRESSIONs (RSEs). Normally if a new record is selected, each of the currency indicators will change — although it is possible to suppress all changes except those to the CRU. To establish a particular record currency, the FIND verb is used and the RSE can select in a number of ways:

- via DBKEY
- via KEY and mode CALC (hash table or index access)
- NEXT, PRIOR, FIRST, LAST in the current occurrence of given SET
- the OWNER (parent) in given set using any currency indicator
- by making CURRENT of RECORD TYPE, REALM or SET the CRU

The verb GET actually transfers data from the CRU to the work area, and in IDMS the verb OBTAIN $\equiv$ FIND + GET.

There are a number of other verbs available such as STORE (to the database!) which creates a new record of specified type from data in the corresponding work area, and also inserts the record into the appropriate occurrence of any set in which its membership is AUTOMATIC. The verb MODIFY may be used to alter data values in the CRU obtaining the new values from the work area (and can sometimes alter its set membership from one set occurrence to another). The verb INSERT (or CONNECT) places the CRU in a selected occurrence of one or more sets, whereas REMOVE (or DISCONNECT) has the opposite effect. The verb DELETE (or ERASE) erases the CRU and may delete members of sets that are owned by the CRU (i.e. any of its child records). The actual result of a DELETE invocation depends on which qualifiers are used, but an injudicious use could cheerfully delete the entire database. Currency and navigation are a complex and dangerous way of moving around a database.

After all that, it is no wonder that the relational database model has won many adherents.

8 The Relational Approach

The Relational Model was founded in 1970 by a paper from E.F. Codd (then of IBM Thomas J. Watson Research Center). It aims to provide a simple and clear

mathematical representation to aid database design. The data model description is at a high level, and the nature of the proposed DMLs is such that they will ensure data independence. There is no provision in the proposals for the storage level specification and so this is left entirely to the DBMS implementor.

To understand and discuss relational data models, we must recall a few terms and define a few crucial ideas:

- An **entity** is an object which exists in the enterprise and which is distinguishable from other objects in the enterprise.
- An **attribute** is a function which maps from an entity set into a domain (set of permitted values) such that every entity can be described by a set of (attribute, data value) pairs with one pair for each attribute of the entity set.
- An **attribute value set** is the set of values that may be taken by a given attribute taking account of **all** possible populations of the database.
- A **relationship** is a n-ary association arising naturally between entities.
- **Populations** : the population is the collection of values **currently** derived from a real-world entity set.
- **Relations (not relationships)** : are current entity descriptions or current entity associations. (Both entity descriptions and relationships may be represented in relations.)
- **Functions**: Those relations which are necessarily functional (whatever that means) in any achievable real world state.

The relational model is founded on the idea of a **domain** or value set which is specified by an underlying simple (scalar) data type which is the smallest semantic unit of data.[2] The domain is essentially a datatype providing a set of scalar values from which the actual values of the various attributes defined in terms of the domain are drawn.

A record in the relational model is known as a **tuple** which is a set of attribute values d_i taken from domains D_i (where $1 \leq i \leq n$). Thus we write (d_1, d_2, ..., d_n) where n is the **degree** of the tuple (ie number of **attributes**). Tuples defined over the same domains D_i and with the same *meanings* are grouped to form **relations**. The number of domains then becomes the degree of the relation. The number of tuples in a relation is known as the **cardinality** of the relation and varies with time. It should be noted that different attributes in a relation may share the same underlying domain (but the intended semantics of the attributes will be different)

[2]This criterion for nondecomposition of domains is First Normal Form — explained later.

and so it is necessary to name both the underlying domain and the rôle played by a given attribute in a relation.

A relation can be thought of as a table with the columns representing the attributes and the rows representing the tuples, but this is an approximation only for three very important reasons:

1. There are no duplicate tuples in a relation (follows from fact that relation is a mathematical set of tuples). Thus a proper implementation of a relation should not permit duplicate tuples to be entered. (SQL unfortunately does allow this.)

2. Tuples are unordered (top to bottom) and this is also a set definition consequence. Thus there is no such concept as positional addressing or nextness *in the model*.

3. Tuples are unordered (left to right) and this is again a set definition consequence. Attributes should only be referenced by name, and there should in particular be no importance attached to the first attribute (i.e. it does not have to be the primary key (explained now)).

In order to address a tuple in a relational database, we need to choose a **primary key** for each relation. As a result of the no duplicate tuple constraint, it will always be possible to accomplish this. The process of selecting a primary key involves determining the **candidate keys** of the relation. A candidate key for a relation R is a subset K of the set of attributes of R where K possesses the properties of **uniqueness** and **irreducibility**. Uniqueness means that no two distinct tuples of R will have the same value of K, and irreducibility means that no proper subset of K has the uniqueness property. If there is more than one candidate key for a given relation then which one becomes the primary key must be chosen. (The other candidate keys are then called **alternate** keys)

8.1 The Relational DML

Two formalisms are available to serve as the basis for relational DMLs, and both have been used in that capacity. Both formalisms are equivalent in that any expression in one formalism can be reduced to an expression in the other. The two formalisms in question are the relational algebra and the relational calculus. The Relational Algebra is efficient at providing query operations at the relation level, whereas the Calculus operates at the tuple level in order to provide efficient update (which would be cumbersome at relation level). I will content myself with presenting only the relational algebra. (Concrete examples of the algebra will (probably) be given in the lectures.)

The Algebra allows **set** operations to be performed on pairs of relations R,S having the same domains underlying their corresponding columns (and hence also the same degree; the two relations should also have the same meanings). Thus tuples from R,S are comparable and the standard Boolean operations can be meaningfully applied. The Algebra supports:

Intersection: R $\cap$ S — a relation that contains those tuples common to R and S (Notation R·S)

Union: R $\cup$ S — a relation that contains all the tuples from R and S (Notation R+S)

Difference: R $\setminus$ S — set of tuples that belong to R but NOT to S (Notation R-S)

Quadratic Join: If R is a relation over domains (D_1, . . ., D_m) and S is a relation over domains (D'_1,. . ., D'_n), then the quadratic join R $\times$ S has the domains (D_1,. . .,D_m,D'_1,. . .,D'_n), and therefore it consists of concatenations (d_1,. . .,d_m,d'_1,. . .,d'_n) where (d_1,. . .,d_m) ε R and (d'_1,. . .,d'_n) ε S. As the notation suggests, this is essentially the Cartesian product of the two relations and so the resulting cardinality of R $\times$ S is the product of the cardinalities of R and S.

There are three data manipulation operations specific to the relational data model, namely **selection**, **projection** and **equi-join**.

Selection expressions define filters that accept or reject the tuples of a relation. Exactly what can be specified in a selection expression depends on the underlying DBMS. The simplest selection expressions restrict by constraining the values in a particular attribute. Most systems permit Boolean combinations of such elementary selectors. More generally if two attributes (C_i,C_j) in a relation are defined over the same domain D, it should be possible to SELECT via expressions involving the domain values d_i,d_j. The most general case is of selection by a tuple-predicate f(d_1,. . .,d_n) that returns the boolean true or false to indicate whether tuple (d_1,. . .,d_n) is to be accepted or rejected.

Projection serves two functions. Firstly it permutes a subset of the columns (rôles) of a relation: secondly it permits redefinition of rôle identifiers. The latter is needed in some DBMS to enable correct data linkage via the equi-join, and may aid sensible tabulation. Projection (because it specifies a subset of a relation), like selection, can only decrease or maintain the cardinality.

Equi-join provides the cross-reference by value between two relations, and requires good tactics as it is the most costly operation. Assume two relations R defined by columns (C_1,. . .,C_m) with domain D_i underlying C_i, and S defined by columns (C'_1,. . .,C'_n) with domain D'_j underlying C'_j. A common requirement is

to combine data from tuples in R and S that match on some column or columns. Thus if columns C_i in R, C'_j in S share the domain $D_i = D'_j$ R and S can be joined "WHERE $C_i = C'_j$". Similarly if there is more than one pair of columns that share an underlying domain. In the worst case (when all tuples in each factor in the JOIN share the same single value of the "join key") the resulting relation can have cardinality that is the product of the cardinalities of the two factors.

The set operations together with selection, projection and equi-join may be combined to form relational expressions in order to pose more complex queries.

8.2 SQL — Structured Query Language

The first prototype of this language appeared in 1974-75, and it has been revised since then *culminating* in an ISO/ANSI standard known as "SQL/92" (or International Standard Database Language SQL (1992)). The standards document is over 600 pages long, and so I will not even pretend to present SQL in any depth. The fact that SQL/92 has diverged greatly from being a close implementation of the relational model is another reason why I will not spend much time on it.

SQL/92 consists of three components: a data definition language, a data manipulation language, and a view definition. The data definition language possesses a schema definition and a table definition (table is SQL-speak for relation). The data manipulation language is based on relational calculus with relation-valued queries. The view definition consists of named relational expressions. An SQL database consists of one or more schemas (database areas belonging to some individual user), with transaction support in terms of "commit" and "rollback".

SQL data definition consists of issuing a CREATE SCHEMA invocation followed by an AUTHORIZATION $<$ user $>$ in order that the user can then GRANT privileges. The principal task after this is to define the tables that will store the data. This is done via the CREATE TABLE command:

```
CREATE TABLE Cities
 ( CITY CHAR (15) NOT NULL,
   POP  DECIMAL (10),
   PRIMARY KEY (CITY)   )
```

The above example defines a table with two columns (SQLuses the term "column" where the relational model talks of an "attribute") where the CITY column will be used as the primary key. The domains as allowed by SQL are not user-defined types in any real sense but merely a slightly extended set of primitive built-in data types that one would expect to find in any ordinary programming language. No support is available for strong typing or inheritance. SQL tables are allowed to have duplicate rows (ie tuples) and the tables are further considered to have a left-to-right column ordering. Tables can be ALTERed at any time

to insert/delete a column or to insert/delete a column default value (NULL is the "default default").

Data manipulation comprises four principal statements SELECT, INSERT, UPDATE and DELETE. I will content myself with giving some simple examples for these operations which should be (fairly) self-explanatory but will not demonstrate the full power of the language.

```
INSERT
INTO Cities (CITY, POP)
VALUES ('Cambridge', 75000) ;

UPDATE Cities
SET POP = 85000
WHERE Cities.CITY = 'Cambridge' ;

DELETE
FROM Cities
WHERE Cities.CITY = 'Oxford' ;

SELECT Cities.CITY , Cities.POP
  FROM Cities
  WHERE Cities.POP > 50000
  ORDER BY Cities.CITY
```

The SELECT statement should not be confused with the *selection* operator from the relational algebra as the use of various optional clauses in a SELECT statement can enable it to perform a complex series of selections, projections and joins.

SQL supports views via the CREATE VIEW statement. It should be noted that INSERTs and UPDATEs on a view will only fail if they violate the view-defining conditions *and if a special optional clause has been included*, otherwise they will not fail. I hope that you all realise that this is logically *wrong*. A view will typically be accesible via the standard data manipulation statements as given above.

9 Redundancy and Normal Forms

Codd's original paper on the relational model presented a methodology for helping to maintain functional dependencies in the form of criteria for schema definitions that store information on a "ONE FACT, ONE PLACE" basis. The normalisation strategies that I will present are based on removing redundancy from the schema

via **nonloss decomposition** where no information is lost in the process of breaking up relations into smaller ones.

Firstly, a definition of what it means to be functionally dependent:

Let R be a relation, and let X and Y be arbitrary subsets of the set of attributes of R. Then Y is functionally dependent on X (or X functionally determines Y (X $\longrightarrow$ Y)) iff each X-value in R has associated with it precisely one Y-value in R.

Thus whenever two tuples in R agree on their X-value, they also agree on their Y-value. It should be noted that the functional dependencies of interest here will be ones that relate to the real-world semantics associated with the data being represented. Thus dependencies that just happen to be true for all the data currently stored but which could potentially be broken next time the database is updated will not count. This is a major reason why the delicacies of modelling real-world data have an important effect on database design.

First Normal Form A relation is in 1NF if and only if all underlying domains contain scalar values only.

1st Normal Form Example						
MAKER	MODEL	DOORS	C.C.	STYLE	DEALER	TEL
AUDI	100 CD	4	2200	SALOON	SMITH	8331
MG	MAESTRO	4	1600	H'BACK	JONES	6221
MG	METRO	2	1300	H'BACK	JONES	6221
ROVER	VITESSE	4	3500	H'BACK	JONES	6221
ROVER	2000	4	2000	H'BACK	JONES	6221
VW	GOLF GTI	2	1800	H'BACK	SMITH	8331

Second Normal Form A relation is in 2NF if and only if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key.

Without 2NF, the functional dependency of a non-key attribute on a subset of the key (in our example, main dealer is only dependent on maker and not (maker and model)) can be broken. This problem can only arise if the primary key is composite in nature (multi-attribute).

Third Normal Form A relation is in 3NF if and only if it is in 2NF and there are no functional dependencies between non-key attributes.

Otherwise we could break such a functional dependency by introducing an additional tuple into the relation. Here is the previous example converted into 3NF.

MODELS				
MAKER	MODEL	DOORS	C.C.	STYLE
AUDI	100 CD	4	2200	SALOON
VW	GOLF GTI	2	1800	H'BACK
MG	METRO	2	1300	H'BACK
MG	MAESTRO	4	1600	H'BACK
ROVER	2000	4	2000	H'BACK
ROVER	VITESSE	4	3500	H'BACK

DEALERS FOR MAKERS	
MAKER	DEALER
AUDI	SMITH
VW	SMITH
MG	JONES
ROVER	JONES

NUMBERS OF DEALERS	
DEALER	TEL
JONES	6221
SMITH	8331

Boyce-Codd Normal Form Let R be a relation of degree n defined over domains D_i ($1 \leq$ i $\leq$ n). A proper subset of k $\leq$ n domains forms a **determinant** if some other attribute value in R is functionally dependent on the values taken in these k domains. Then a relation is in BCNF if and only if the only determinants are candidate keys.

The intuition behind BCNF is that if something determines anything then it should determine everything, hence it should appear only once. It should be noted that BCNF is strictly stronger than 3NF and is conceptually simpler in that it does not refer to more liberal normal forms as such. Breaking BCNF means that we run the risk of storing a determined value in more than one place.

In order to demonstrate BCNF and the next level of normalisation (4NF) we will exhibit a relation in BCNF.

Qantas Airways run a fleet of Boeing 747's. Individual aircraft differ in payload, seating capacity and range, and so each aircraft only flies some of the routes, being operated by particular crews. Spare parts are held at major airports visited by Qantas aircraft, but only those required for models visiting that airport. (The relation we exhibit is all key, and therefore in BCNF, but clearly information is redundantly stored.)

BCNF EXAMPLE		
Aircraft	Crew	Spares Depot
City of Brisbane	Capt Thomas	Auckland
City of Brisbane	Capt Thomas	Tullamarine
City of Brisbane	Capt West	Auckland
City of Brisbane	Capt West	Tullamarine
City of Melbourne	Capt West	Amsterdam
City of Melbourne	Capt West	Singapore
City of Melbourne	Capt West	Tullamarine
City of Darwin	Capt Smith	Auckland
City of Darwin	Capt Smith	Tokyo
City of Darwin	Capt Smith	Tullamarine
City of Darwin	Capt Thomas	Auckland
City of Darwin	Capt Thomas	Tokyo
City of Darwin	Capt Thomas	Tullamarine

Fourth Normal Form Given a relation R with (sets of) attributes A, B, and C, the **multi-valued dependence** A $\rightarrow\!\!\!\rightarrow$ B holds in R iff the set of B-values occurring for a given (A-value, C-value) pair is independent of the C-value. Then a relation is in 4NF if and only if, whenever there is a multi-valued dependency in R, say A $\rightarrow\!\!\!\rightarrow$ B, then all attributes of R are functionally dependent on A. (Equivalently: R is in 4NF if R is in BCNF and all multi-valued dependencies in R are in fact functional dependencies.)

I will now exhibit the Qantas aircraft schema in 4NF.

AIRCRAFT	CREW
City of Brisbane	Capt Thomas
City of Brisbane	Capt West
City of Melbourne	Capt West
City of Darwin	Capt Smith
City of Darwin	Capt Thomas

AIRCRAFT	SPARES DEPOT
City of Brisbane	Auckland
City of Brisbane	Tullamarine
City of Melbourne	Amsterdam
City of Melbourne	Singapore
City of Melbourne	Tullamarine
City of Darwin	Auckland
City of Darwin	Tokyo
City of Darwin	Tullamarine

Fifth Normal Form Let R be a relation, and let A, B,. . ., Z be arbitrary subsets of the set of attributes of R. Then R satisfies the **join dependency** * (A, B, . . ., Z) if and only if R is equal to the join of its projections on A, B, . . ., Z. Thus a relation is in 5NF (also called **projection-join normal form** (PJ/NF) iff every join dependency in R is implied by the candidate keys of R.

So are Normal Forms a wholly 'good thing'? They do remove a number of serious problems resulting from redundant information, but they also have their problems in that decomposition may lead to poor performance (because of the need to run round tables), decomposition may make it easy to break semantic constraints, and finally one sort of normalisation may cause another to be broken. Constraints of referential integrity can aid us, but I do not have the time to go into the nature of these constraints.

10 Missing Values and other problems

10.1 The Problem of the Missing Values

In the real world, it is often the case that you don't actually know the answer to a question, or the question is in fact inappropriate or irrelevant in your particular circumstances. The Problem of the Missing Values is that we must decide how the database shall represent such real world situations. This area of concern has sparked a tremendous controversy in the literature and there is still no consensus. I shall outline the two opposing "solutions" but I stress that neither of them should be considered to be the *right* answer as Missing Values are a really nasty problem.

The first "solution" is to use **nulls** to represent missing information. Thus in a historical database storing genealogical information, we would insert a special marker (a null) in any tuple where we didn't know the date of birth for instance. Thus we insert a null in any attribute position in order to record the fact that we don't know the given fact — i.e. that the requested value is UNKNOWN. A null is not a blank and it does not equate to zero, it is simply an undetermined value. It may well be the case that certain attributes had better not contain nulls and so some current relational database software includes DDL extensions to allow the database designer to specify whether a given attribute may be assigned nulls or not. The problem with using nulls is a consequence of nulls being based on three valued logic (true, false, and unknown). Thus when we ask queries like "How many people were born before 1st Jan 1900?" of our hypothetical historical database, we cannot be sure what the answer really means, and expected identities such as (Number Born Prior to 1st Jan 1990) + (Number Born After 31st Dec 1899) = (Total Number of Entries in Database) will also not hold. For this and similar reasons, some researchers hold that null values have no place whatsoever

in a mathematically based model such as the relational model, and that their very introduction destroys the mathematical validity of the model.

The alternative "solution" is to use **defaults** instead of nulls. Thus wherever we have to represent missing information, we insert the default value associated with the corresponding domain. This has a number of consequences. For instance, on the insertion of a new tuple into the relation, the user must supply a value for each attribute where a default value would be illegal (such as an attribute which by itself or in tandem with others forms a candidate key), and the system will supply default values for any other attribute where the user is unable to specify a value. There is a need to support a builtin function to return the default value associated with a particular domain. When applying aggregate functions to the relation (say to average the wage earned by each employee), there is an explicit need to ignore default values. There is a certain amount of trouble caused in cases where every value in the domain is a possible real (in the sense of being nondefault) value, and so explicit user support is required to handle these cases. The use of defaults is considered not to result in the normalisation breakdown caused by the use of nulls. Default values are a somewhat inelegant solution requiring significant user interaction, but their proponents claim that default values are intuitively closer to what we use in the real world.

10.2 The Problems of Aggregates and Nonscalar Domains

It is often the case that we wish to manipulate groups of records in order to answer such queries as "What is the monthly wages bill for the quality control section?" This cannot be accomplished using selections, projections and joins. In SQL, this sort of query can be satisfied using the GROUP BY field and SUM builtins, but this is an SQL fudge to provide this sort of **aggregate** function. There is no generalised solution in the basic relational model for handling sets of tuples as first-class values.

One extension which provides some help towards solving aggregate requirements and also support for more complex datatypes is to abandon the insistence that all domains must correspond to scalar types. This can be accomplished by a method known as the **nested relational model** (aka **NF2 — Non First Normal Form**). Thus in a database comprising details on scientific papers we are now allowed to have the attribute authorlist which is a simple set of authornames and so the domain of authorlist is relation valued. This (one-level) nesting of relations enables us to represent dates as an attribute whose domain is the relation consisting of attributes Day, Month and Year. Care must be taken when using nested relations in this way as the model is too liberal in terms of values that would be valid members of the domain underpinning Month say but not valid components of a calendar date. Extensions must be made to the DML in terms of NEST and

UNNEST (flatten) operators in order to provide this functionality.

One concern raised about using NF2 is that it endangers the validity of normalisation strategies, and introduces additional possibilities for needing nulls. A variation has been proposed to use encapsulated relation-valued attributes where you are unable to look simultaneously at both the inner structure of an attribute and the outer face it presents at the higher level of the relation.

Recursively nested relational models have been suggested (i.e. multiple levels of nesting) but there is little experience of using these in the real world.

11 Object Orientated Databases

It *may* be the case that object-orientated databases will be the successors to relational databases, and so it is a thoroughly good idea to probe deeper than the hype. An object-orientated DBMS (OODBMS) comprises an object-orientated programming language combined with persistence and transaction support facilities. (You will all by now be Modula3 and C++ wizards and so abstract datatypes and class-based methods should hold no fear.)

Proponents of OODBMS tend to be rather fanatical in their views and so present *manifestos* (rather than polite proposals) as to which features should be mandatory in an OO database.

For a database to be truly object-orientated, it must allow object identifiers, user-defined types and at least simple inheritance. The objects must be encapsulated so that they can only be used via an interface. Support must also be provided for aggregates in terms of bulk types, sets and relations. Late-binding (for persistence) is mandatory in that new programs must be incorporated by name using dynamic lookup. The database must implement concurrency control and recovery procedures as necessary features for transaction support. Finally *ad hoc* queries (or 'database browsing') must be efficiently supported even in large databases. (Of course this is only one of many OO philosophies, OO manifestos come in more flavours than ice-cream — which is clearly evidence of an *active* research area.)

The key mantra to chant is "**Everything is an object**" (sometimes refined by high adepts to "Everything is a **first-class** object). Objects can be divided into two worlds — builtin, primitive, **immutable** objects (such as integers e.g. 666, character strings e.g. "Help!"), and complex, usually user-created, **mutable** objects (like Student, Vehicle). Every object must have a **class** (i.e. a type), and individual objects are often called **instances** in order to distinguish them from their defining class. Each class will have a set of **methods** which are functions and operators which can be applied to objects of that given class.

Objects are always **encapsulated** in that the internal structure of an object is

hidden from the users, and they must access the objects by making calls to the object's methods — the methods naturally can manipulate the internal structure. Key terms to bandy about are "private memory" (instance variables representing an object's internal state) and "public interface" (interface definitions — inputs and outputs to the various methods). Hence we have **encapsulation** $\longrightarrow$ **data independence**. The methods are invoked by **messages** which are function calls with a little extra syntactic sugar.

Each and every object has its very own unique identity called its "object ID" or OID. Immutable objects are said to be "self-identifying" in that they are their own OIDs, whereas mutable objects have (conceptual) addresses as their OIDs, which are then usable elsewhere in the database as (conceptual) pointers to reference the objects in question. Objects in OO databases do not therefore need to have user-defined candidate keys for entity identification and reference but as the OIDs are not directly visible to the user something has to be available for user interaction. It is claimed by some that being able to represent two or more distinct objects which are identical in all user-visible aspects (ie differing only in OID) is an advantage of the OO approach over the relational model. This is really dubious because how will the user be able to distinguish between the objects?

To create a new instance of a given class, it is necessary to send a **New** message. Objects can also contain OIDs pointing to other objects which means that the very same object can be shared by many objects, and such a shared object is said to belong to multiple **collection** objects. Naturally there is also scope for **class hierarchies** e.g. the object class Student is said to be a **subclass** of object class Human or equivalently object class Human is said to be a **superclass** of the object class Student if and only if every object of the class Student is necessarily an object of the class Human. Thus objects of a subclass can inherit the instance variables (structural inheritance) and the methods (behavioural inheritance) of their superclass. Some systems support **multiple inheritance**, where a given class can be a subclass of several superclasses simultaneously.

Clearly in an OO database, object instances must be the natural units of security, authorisation, recovery and concurrency.

We conclude this overview of the OO database model by voicing a few concerns about generally held (mis)conceptions regarding OO. It is claimed that OO simplifies database design and development due to its capability to provide system supported high-level modelling structures. The real problem is deciding for any given data whether it should be modelled as instance variables or in a procedural fashion as a method. It is also claimed that OO databases can model complex objects without the need for normalisation strategies so beloved of relational databases. Unfortunately the normalisation strategies have been developed to handle problems which are inherent in any data where functional dependencies exist, and not just relational models. OO is not a magic wand with which we can wave

away these very real problems. Finally many folk (including software vendors) will convey their belief that the relational model has had its day, and that the time has come for the OO model to conquer the world. They firmly expect that the relational model will be swept aside in the same way that the relational model swept aside the network model and the hierarchical model. There is however a significant difference between relational and pre-relational models in that the prerelational models were *ad hoc* schemes which worked by dint of convoluted low-level strategies and twisting the real-world data to fit the model's structure, whereas the relational model is founded foursquare on a solid theoretical basis which has been refined through years of research. (A full discussion of the need to marry the relational model and the OO model together can be found in Date[1], chapter 25).

References

[1] C J Date. *An Introduction to Database Systems*. Addison Wesley, 6 edition, 1994.

[2] R Elmasri and S Navathe. Addison Wesley, 2 edition, 1994.

[3] William Kent. *Data and Reality*. North Holland, 1978.