Databases - Introduction

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

What is data?

- A *datum* is a single fact about some *universe of discourse*.

  Jack Johnson has student number 20723081

- *Data* is the plural of datum, hence refers to a collection of facts.

  By itself, data has little use or meaning:

- *Information* is the *interpretation* of data — attributing *meaning* to the data.

  As one writer has said:

  Information is what you want; *data* is what you've got.

Databases

A *database* is an organized collection of data, often representing a model of the activity of some business or other organization, large or small.

For example, a database might contain data about:

- Students, Courses, Units and Grades
- Customers, Products, Orders and Deliveries
- Doctors, Patients, Prescriptions and Drugs
- Students, Books, Periodicals and Loans

The fundamental role of *database technology* is to allow users, typically organizational users, to extract information from their data.

Database Management Systems

A *database management system* (DBMS) is any system that allows users to manage their data, although nowadays the term is used almost exclusively for *software* rather than manual systems.

The *amount* of data that can be collected automatically is growing exponentially, and so there is a strong demand for database professionals, particularly *database administrators* (DBAs) who *design*, *implement* and *maintain* databases.

There are many different types of DBMS, but *relational* database management systems (RDBMS) have been the dominant paradigm for several decades.

An old-fashioned DBMS

Before computers became ubiquitous, libraries used *card catalogue systems*, which are just manual databases.

There are many variations, but usually they would at least have:

- A large *card catalogue* listing all books in the library
- *Check-out* slips stored in little envelopes in each book
- An *"on loan"* list formed from the check-out slips of the books on loan

An old-fashioned DBMS

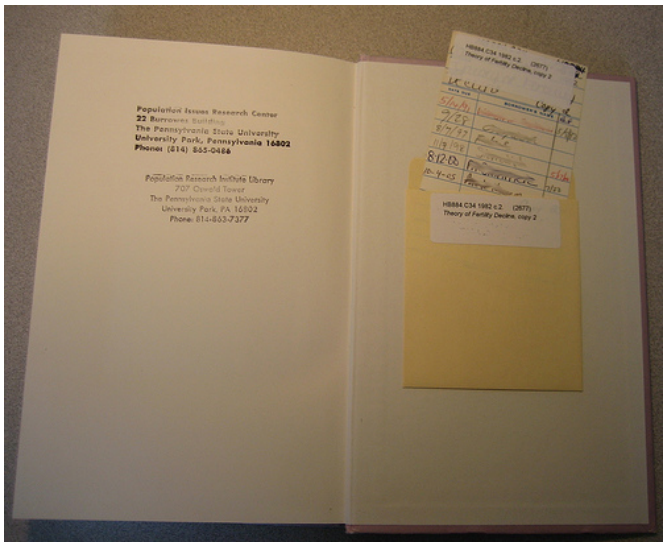
A card catalogue consisted of wooden cabinets with large numbers of drawers, each containing an index card relating to a single book.

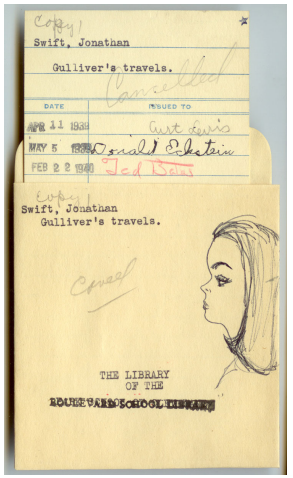
The drawers, and cards, were *alphabetically sorted*, usually by author name, to allow readers to find which *titles* the library stocks.

"2010 Manchester UK 4467481691" by Ricardo from Manchester, UK - Manchester Central Library, March 2010

Check-out cards

Each book had a little envelope with a *checkout-card* that was removed and filled-in with the due-date and the borrower's name, while the due-date was stamped onto a slip of paper pasted into the book.





The check-out cards were stored in *due-date* order, allowing library staff to quickly access the slips when books were returned, and to determine each day which books had become overdue.

Features of this DB

Despite being manual, the library system nevertheless displays some of the features of a modern database:

It had separate listings (catalogues) for *books* and *borrowers* with the check-out cards forming a third list *connecting* specific books to specific borrowers.

Each of the lists is *sorted* in order to permit rapid searches — in modern terminology, we would say that the lists are *indexed*.

Why use a DBMS?

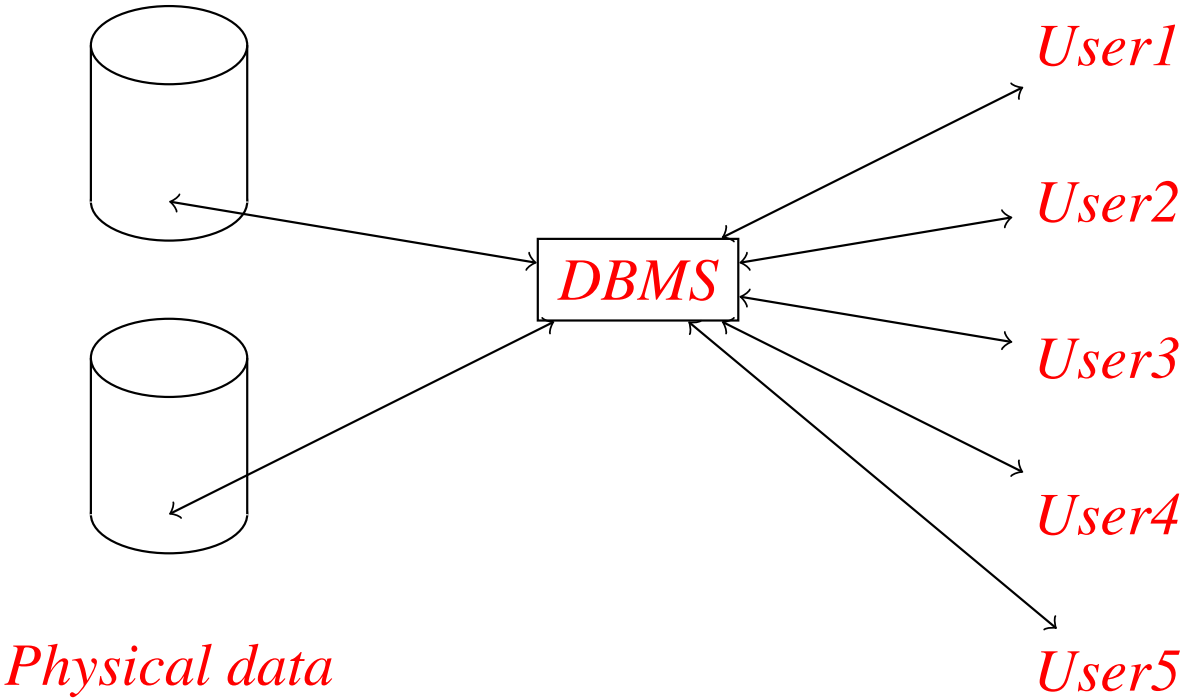
Many organisations can (and do) manage their data simply as a collection of files on a shared file system.

Any user with sufficient file system privileges can search, open and edit the files.

This is conceptually simple, but has many disadvantages in terms of flexibility, reliability, scalability and data integrity.

A DBMS sits *between* the users and the data, and *manages the interactions* between them.

The DB model



What does a DBMS provide?

There is a free online course *"Introduction to Databases"* on *Coursera* (`coursera.org`), presented by Jennifer Widom from Stanford University.

In this, she describes a DBMS as providing *"efficient, reliable, convenient, and safe multi-user storage of – and access to – massive amounts of persistent data"*.

(In general, the Coursera video lectures and supporting materials from this course are a useful resource for this unit.)

In more detail

We'll consider some of these in more detail.

- Data Independence
- Efficiency
- Data Integrity
- Data Administration
- Concurrency Control
- Application Development

Data Independence I

Data independence provides analogous benefits to the *encapsulation* found in object-oriented programming languages:

- Users and applications use a *logical model* of the underlying data, rather than directly manipulating the physical files storing the data.
- *Implementation* of physical storage can be altered or improved without affecting client code.
- Physical storage can be remote, or distributed, or both, with no alteration in client code.

Data Independence II

If the user wants to know the name of the student with student number `22041020` then they *ask the DBMS*, using something like the following SQL (Structured Query Language) command[1].

```
SELECT name FROM Student WHERE snum = 22041020;
```

The user does not need to know *where* this information is stored, what machine it is on, what files it is in and so on — the mechanics of locating and extracting the information is left to the DBMS.

This is a *declarative* statement rather than an *imperative* statement — the user is *asking* for the desired information, rather than *telling* the computer exactly what to do.

[1] Do not worry if you do not understand this yet!

Efficiency

A DBMS can implement a number of storage strategies and optimizations to make the most common operations as fast as possible.

In particular, the DBMS can maintain various *indexes* to the data to make querying the database quick; the user can control which indexes are present, but need not know how they are implemented.

Database storage and indexing strategies are extremely sophisticated applications of data structures techniques.

Data Integrity

A critical role of a DBMS is to ensure that the entire collection of data is maintained in a *consistent* state.

At its simplest, this means that each type of data should only be stored once — if several parts of an organization use personal data about an employee, and each stores that data separately, then it is difficult to enforce integrity.

More signficantly, a change in one data item often has a "ripple effect" of consequences for data in other areas: a DBMS can ensure that all of these consequences occur and disallow an operation that would leave the database corrupt.

This is often described as the DBMS enforcing *integrity constraints*.

Data Administration

A DBMS allows the organization a fine degree of control over who is permitted various levels of access to the database.

In most operating systems (Unix, Windows etc) users can either read an entire file or none of it, but a DBMS can present different *views* of the same data to different groups of users.

For example, a lecturer may be able to look up a student's academic record, but not their personal or financial details, while only certain staff will be able to *alter* their academic record.

Concurrency Control

In a large organization, there will often be several people accessing the same data item at the same time.

While this is not a problem if all users are simply *viewing* the data, it becomes a major problem if some of the users need to *update* the data.

For example, an airline reservation system may have several travel agents viewing availability at the same time, but the DBMS must prevent two agents from booking the same seat at the same time.

Application Development

Analysing data may require more sophisticated and application-dependent programs than a general-purpose DBMS can provide.

This can be accomplished by having a general purpose programming language such as Java and C accessing the data through the DBMS and then performing additional processing with the results.

This combination permits the developer to focus on the "business logic" of the application.

The power and success of this form of application development can be seen by the fact that essentially every large dynamic website is a database-backed application.

A large subject

Each of the topics listed above has enough theory, practice and technology associated with it to form an entire unit that could legitimately be called *Databases*.

Ramakrishnan & Gehrke[2] identify two major approaches:

- Systems Emphasis

  Building database systems — the nuts and bolts of storage, indexing, query optimization, transaction management.
- Applications Emphasis
  Using database systems — data modelling, data query languages, database-backed applications.

[2] `http://pages.cs.wisc.edu/~dbbook/`

This unit

We will take the *applications emphasis* covering

- The relational data model
- Database design using entity-relationship (ER) diagrams
- SQL (the standard) and MySQL (the implementation)
- Database use in practice
- Database-backed applications

There are a number of *different data models*, but the *relational model* is the most entrenched, developed and heavily used, so we will focus almost entirely on that.

Terminology

Theoretically, the relational model is based on *formal mathematical concepts*, so uses mathematical terminology in a precise way, e.g.

- Relation, instance, tuple, attribute, field.

In practice, implementations of databases are only loosely based on the formal model, so a more informal terminology is used, e.g.

- Table, row, column, heading.

It is important to recognise both the formal terms and the informal terms!

Tables / relations

The *table*, or *relation* (the dark green words are the formal terms), is the fundamental object manipulated by a *relational database*.

Here is a toy example of a table / relation called `Book`.

Author	Title	Date
Tolstoy	Anna Karenina	1873
Steinbeck	Cannery Row	1945
Wharton	Ethan Frome	1911
Conrad	Lord Jim	1900
Kerouac	Big Sur	1962

This is a poorly-designed table, but we'll dissect it to learn the terminology, and sneak in some basic SQL commands as well.

Tables / Relations

In an RDBMS a *table* represents the *objects* or the *relationships* between objects based on analysis of the application.

A table has

- A *name* allowing the designer and user to refer to it
- A *header* giving names to the *columns*, each of which has a specified *type*
- Zero or more *rows*, each representing one object

The rows

Each *row* contains the data for one book title.

Author	Title	Date
Tolstoy	Anna Karenina	1873
Steinbeck	Cannery Row	1945
Wharton	Ethan Frome	1911
Conrad	Lord Jim	1900
Kerouac	Big Sur	1962

The highlighted row refers to the book "*Cannery Row*" written by Steinbeck and first published in 1945.

In the more formal language, a *row* is called a *tuple*.

The columns

The *header* of each column names a *general property* of the objects being stored in the table.

A *column* of the table stores the *values* of these properties, one value per object (i.e. per row).

Author	Title	Date
Tolstoy	Anna Karenina	1873
Steinbeck	Cannery Row	1945
Wharton	Ethan Frome	1911
Conrad	Lord Jim	1900
Kerouac	Big Sur	1962

Thus each book has a *publication date* — a *general property* of books, but each individual book has *its own* publication date, and so the *value* of this attribute differs between books.

Attributes

The formal name for a "*general property*" is an *attribute*.

At the design stage, the database designer has to decide *which attributes* of the objects need to be modelled (stored) — usually by thinking about the real-life objects, and distinguishing the important from the unimportant.

For example, the following are all attributes of students — but which are relevant to designing a table `Student` for a university database?

- Name, Address, Date of Birth, Student Number
- Height, Eye Colour, Favourite Movie, Nickname

Types I

Each attribute also has a *type*, which refers to the kind of values that can be stored in that column.

Some common types are:

- Various *numeric* types, including;
  `INT`, `BIGINT`, `FLOAT`, `DOUBLE`
- Various *time-related* types, including:
  `YEAR`, `DATE`, `DATETIME`
- Various *text-related* types, including:
  `CHAR`, `VARCHAR`, `TEXT`
- Various other special-purpose types, including
  `BLOB`, `ENUM`, `BINARY`

If a column is declared to be of a particular type, then it can only be assigned legitimate values of that type.

Types II

The different types are useful in several different ways:

- They help the DBMS use storage efficiently by letting it know how much space to allocate for all fixed-size types
  For example, in MySQL, an `INT` *occupies 4 bytes of space*
- They provide the first layer of protection of data integrity by avoiding obvious data entry errors
  For example, if a column has type `DATE`*, then it will only accept only values given in the correct format for a date.*
- The DBMS can interpret commands differently according to type
  For example, the command that adds 1 to a value will use integer arithmetic for an `INT`*, but adds* one day *if the variable is of type* `DATE`.

Tuples I

The number of columns of a particular relation (table) is known as the *arity* of the relation. There are special words *unary*, *binary* and *ternary* relation for relations of arity 1, 2 or 3 respectively.

The `Book` relation has arity 3, so each row is a *3-tuple*, or an *ordered triple*.

Mathematically, a tuple is given as a comma-separated list of values between brackets. Thus we can say that

```
(Dickens, David Copperfield, 1850)
```

is a *legal tuple* for the relation `Book`.

Tuples II

Some tuples are simply not legal, either because they have the wrong *arity* or a component has the wrong *type*. So

```
(Dickens, 1850, David Copperfield)
```

is not a legal tuple because the third component does not represent a date, and

```
(Dickens, David Copperfield)
```

is not a legal tuple because it has the wrong arity.

On the other hand,

```
(Dickens, The Da Vinci Code, 2003)
```

is legal, but just incorrect.

A subtlety

It is sometimes necessary to distinguish between the *contents* of a table, and the *structure* of the table, and for this, the more formal language is needed.

- The structure of the table — that is, the names and types of the attributes — is called the *schema* of the relation.
  The schema of a relation is normally carefully designed and changes infrequently, usually in response to some structural change in the business environment
- The contents of the table at any particular point in time is called an *instance* of the relation
  Under normal usage, rows may be added to a table, then altered and finally deleted, and so the relation instance *is frequently changing*

Structured Query Language

Structured Query Language or SQL is an *ISO standard* specifying the *syntax* and *semantics* of a declarative language for accessing a relational database.

- Syntax — The syntax determines which statements are *legal expressions* in the language
- Semantics — The semantics determine *the meaning* of each of the legal expressions

However there is no compulsion on any database vendor to stick precisely to the standard, and so there are numerous *"flavours"* of SQL.

Every *actual* database system *omits* some SQL commands, but *includes* some non-standard extensions.