# CS2 Current Technologies note 5

# Why we need databases

**Peter Buneman**

## 5.1 Introduction

Imagine, if you will, that XML had been invented long before database management systems. Would we now bother with databases? In fact our hypothesis is almost true. The language Lisp was invented and widely used long before relational databases were thought of. Lisp, among other things embodies a data format that is simpler and cleaner than XML, but it didn't catch on either as a means of representing data or as a format for data transmission. There are three reasons.

- Efficiency. All the XML tools you have been using require the whole XML document to fit into main memory. They will simply break on large documents. There are some attempts to build general-purpose XML query systems that will work on very large documents, but these are all highly experimental. By contrast, relational database systems, when properly "tuned" are quite happy to deal with terabytes of data in secondary storage. We are not going to deal with efficiency in any detail in these lectures, but you should remember that the way we organise secondary storage data for efficient processing is completely different for what we do in main memory.

- Structure. XML is designed as an interchange format for data, not as a language such as Java for designing new data types. In XML the serial structure of data is important; in programming languages we regard the way the language represents our data in memory as "implementation detail" and often expect that the language will do something for us (e.g. in producing the memory representation for instances of a Java class definition.) In fact, XML sits a bit uncomfortably between a serial format and a data specification language. Even the designers of XML argue this point! Designing databases is extremely important, and we shall discuss it in this thread.

- Updates and transactions, XML was not really designed to be updated; if you want to change anything in an XML document, you simply creat a new docement. Hardly a good way of doing things if it takes you hours of disk-boound i/o time to create a new document!

  Even so, let's suppose your bank implemented their database using XML. Somewhere there would be an XML file containing your account identifier and your balance (among other things). Suppose you deposit a cheque for £100 by mail and sometime later withdraw £50 from a cash machine. It might happen that two processes, *deposit* and *withdraw,* are simultaneously called to change the XML file. The sequence might look like this:

1. *withdraw* opens the file and reads it into memory.
2. *deposit* opens the file and reads it into memory.
3. *withdraw* finds your balance and subtracts £50 from it.
4. *deposit* finds your balance and adds £100 to it.
5. *deposit* writes out the file.
6. *withdraw* writes out the file.

Would you be happy? You have probably already seen that there are mechanisms for "locking" files so that only one process can use it at once. That might keep you happy, but would it keep your bank happy? Your bank may be processing tens or hundreds of deposits and withdrawals a second. You can't read thorugh a file of any size a hundred times a second. Try it! Clearly something else is needed.

This last point indicates that database systems give us more than query languages. Many real databases are small enough to fit into main memory, so we could simply program them as data structures in our favorite programming language, but this would *not* solve the problem of concurrent access.

In this thread we are going to be mostly concerned with the basics of database design and querying.

## 5.2 The relational data model

.

Let's start with and example of something you are – or should be – acutely familar: the XML data set of the current homework.

```
<doa>
  <committee>
    <title> Very Important </title>
    <chair> <name> A. Stoat </name> <email> ast@doa </email> ... </chair>
    <secretary> <name> W. Weasel </name> <email> ww@doa </email> ... </secretary>
    <members>
      <member> <name> T. Feret </name> <email> feret@doa </email> ... </member>
      <member> <name> T.H. Toad </name> <email> toad@doa </email> ... </member>
    </members>
    <subcommittees>
      <committee>
        <title> Administration </title>
        <chair> <name> T. Feret </name> <email> feret@doa </email> ... </chair>
        <secretary> <name> T.H. Toad </name> <email> toad@doa </email> ... </secretary>
        <members>
          <member> <name> M.R. Fish </name> <email> mrf@doa </email> ... </member>
          <member> <name> M. Magpie </name> <email> mm@doa </email> ... </member>
        </members>
        <subcommittees> </subcommittees>
      </committee>
    </subcommittees>
  </committee>
  <committee>
```

```
  ...
</doa>
```

This document was great for extracting information about committees. A simple stylesheet was all that was required to make this information readable. It was not so good for extracting information about people – we had to do more work using a query language to get that. However, when we come to SQL, we are going to be doing this kind of work in any case to get answers, so maybe we shouldn't object – at least not on the grounds of database design.

So, returning to the second point above, what is wrong with the XML representation? There are two very serious problems.

- *Redundancy*. Information on people is kept redundantly. T. Feret's email and office are repeated everywhere T. Feret appears. This is not only wasteful of space; it is dangerous. If we want to change T. Feret's email, we have to find all the places in the document we need to do this.
- *Lost information.* This is the converse of redundancy. The XML document contains information about committees and people, but unless someone is on at least one committee, we don't have any information on them. This seems wrong. Does a person cease to exist if they are no longer on a committee? Maybe in the Department of Administration ...

What do we do about this? One way out is to create *two* XML documents – one for committees and one for people. Or, equivalently, one document with an element for people and and another for committees. Something like this:

```
<doa>
  <committees>
    <committee>
      <title> Very Important </title>
      <chair> A. Stoat </chair> <secretary> W. Weasel  </secretary>
      <members> <member> T. Feret </member> <member> T.H. Toad </member>
      </members>
      <subcommittees>
        <committee>
          <title> Administration </title>
          <chair> T. Feret </chair>
          <secretary> T.H. Toad </secretary>
          <members> <member> M.R. Fish </member> <member> M. Magpie </member> </members>
          <subcommittees> </subcommittees>
        </committee>
      </subcommittees>
    </committee>
    ...
  </committees>
  <people>
    <person>
      <name> A. Stoat </name> <email> ast@doa </email> <office> 1A </office>
    </person>
    <person>
      <name> W. Weasel </name> <email> ww@doa </email> <office> 123 </office>
    </person>
```

```
    <person>
      <name> T. Feret </name> <email> feret@doa </email> <office> 12B </office>
    </person>
    <person>
      <name> T.H. Toad </name> <email> toad@doa </email> <office> 14A </office>
    </person>
    <person>
      <name> M.R. Fish </name> <email> mrf@doa </email> <office> 12B </office>
    </person>
    <person>
      <name> M. Magpie </name> <email> mm@doa </email> <office> 24A </office>
    </person>
    ...
  </people>
</doa>
```

What we have done here is partially to "flatten" the document. In fact, the design is now rather satisfactory. We'd like to add the constraint that any `name` in the commitees section occurs in the `people` section, but that's about it.

However, we might observe that the `people` element has a higly regular structure. It is essentially a table – a set of records or tuples:

`people`:

| name | email | office |
|---|---|---|
| A. Stoat | ast@doa | 1A |
| W. Weasel | ww@doa | 123 |
| T. Feret | feret@doa | 12B |
| T.H. Toad | toad@doa | 14A |
| M.R. Fish | mrf@doa | 12B |
| M. Magpie | mm@doa | 24A |
| ... | ... | ... |

The idea behind the relational model is to represent *everything* as a table. So we should perhaps start with the other main kind of "thing" in our data – the committee. The rule is that the only things we can put in the tables are "base" values, strings, integers, etc. We cannot put lists or other tables into a table. We can start on committees as follows:

`committees`:

| title | chair | secretary |
|---|---|---|
| Very Important | A. Stoat | W. Weasel |
| Administration | T. Feret | T.H. Toad |
| ... | ... | ... |

However, this has now lost us information about members and subcommittees. To recover this information we add two more tables:

members:

| committee | member |
|---|---|
| Very Important | T. Feret |
| Very Important | T.H. Toad |
| Administration | M.R. Fish |
| Administration | M. Magpie |
| ... | ... |

subcommitees:

| committee | subcommittee |
|---|---|
| Very Important | Administration |
| ... | ... |

This, then, is the idea behind relational databases – reduce everything to tables. It is clear that the separation of a people table from the rest of the hierarchically structured document was a good idea, but did we need to go the whole way and "flatten out" everything, This remains a debate. Object-oriented and object-relational systems do not demand this degree of flattening, but relational database systems have dominated the practical database market for over 20 years. Why is this?

- They are extremely simple to understand.
- Query languages for them are well understood. There is an interesting and useful connection between relational database query languages and first-order logic.
- Query languages are *optimisable*. There is well-developed technology for this. Optimising queries for hierarchical databases and XML is much less well-understood.
- Updates and transaction processing are easier to understand and implement for relational databases.

We should also be clear about what we mean by a table. A table consists of *rows*, often called *tuples*. There is a clear similarity between a row and a record in a programming language. A table is a *set* of rows. This means that the order of the rows is unimportant and that there are no duplicated rows. The fact that tables are sets rather than lists is good because it allows us to do more optimisation (we don't have to preserve any order). It is bad because we often want to represent order and it is clumsy to do it with tables. The best way is to add a column of integers that indicates priority.

## 5.3. Database Architecture

There is a traditional "three level" approach to describing database architecture. Although it isn't accurate, it's a good starting point.

**The Physical Level** This is how our tables are represented as data structures in computer memory – main memory or secondary storage. Interestingly, the physical representation does not look particularly "tabular". All sorts of structures like linked lists are used at this level. It also includes index structures (search trees and hash tables) that are used to optimise queries.

**The Logical Level** This is a particularly annoying term. It indicates that the physical level is "illogical". There's enough bad database design around to suggest that the logical level is often anything but logical. A much better term would be the *abstract* level – the interface we have to the database system, which is usually through a query language like SQL. This is the level at which we understand the data to be tabular.

**User Views** Real-world databases typically contain hundreds, if not thousands of tables. If only to simplify the database for the benefit of people who use only parts of it, it is useful to give them access to only a portion of the data. Another reason is security, there may be restrictions on who is allowed to see certain parts of the database. A *view* dictates what a user or group of users can see. It is not just a subset of tables, it can be the result of a query.

It is important to remember that databases are not built with a fixed set of applications in mind. They are there to support a large number of applications, many of which are unknown when the database is designed. In building a database one is usually trying to "model" some part of the real world – people flying on aeroplanes, people ordering books, what is known about the human genome, etc. All of these are extremely useful. What one does is to try to understand fully the part of the real world that is important for a set of tasks or people and to come up with a description – at the logical level – of the data of interest. This is an extremely challenging task. It often happens that more is spent on designing a datbase than building it. The design is crucial. Databases frequently break down as a result of bad design, and these breakdowns are enormously costly.