CS2 Current Technologies note 6

The Relational Data Model

Peter Buneman

6.1 Introduction

Last term we used XML for the representation of data and used XSLT for presenting the data and XQuery for transforming it. We were using XML in the same way that we would use a database. Now the usefulness of XML as a medium for *transmitting* data is beyond question; but whether XML will be used for *storing* data is still an open question. The investment of effort in designing query languages (like XQuery) and sophisticated structural descriptions (like XML-Schema – which we did not cover) indicates that there is some expectation that XML will be used – in some fashion – as the basis for data management, but this has yet to be established. One of the major issues is that of *efficiency*: XML query languages and APIs do not "scale". Very roughly speaking, current technology only allows us to manipulate XML documents that will fit comfortably into main memory.

In the previous set of notes (CS2 note 5) we discussed why an alternative model for data was needed. Let's briefly revisit that topic. We are now regressing from highly "current" technology – technology that has not yet established itself – to 30-year old technology that has very much stood the test of time. However it is important to understand the history of relational databases. Before E.F. Codd invented relational databases, people were dealing with a variety of representations of data, some of them quite close to XML. What was also a problem (as it now is with XML) no clear distinction was made between the physical implementation of data and the *abstraction* we use to think about it – sometimes called the "logical" view.

What Codd suggested is that we should use an extremely simple model for thinking about data. Everything is represented as tables. He also proposed an *algebra* for manipulating tables and established a connection between that algebra and predicate calculus – a connection that has proved invaluable for telling us what we can and cannot do with relational databases. We are going to make a brief study of that algebra, though not the connection with logic, and it is worth taking a few moments to understand why.

When Codd first described the relational data model it was regarded as theoretically elegant but practically unworkable. But it turned out that relational algebra is what was needed to make databases work. The first useful observation was that relational algebra uses a very small number (five – but that needs qualification) of operations. Implement those operations efficiently and you may have a good starting point for implementing queries. The second observation was that relational algebra comes – as it name suggests – with some rewriting rules or equations. In arithmetic, for example, we use the equation $ab + ac = a(b + c)$ to "simplify" our calculations.

That is, they require less work from us in evaluation. Similarly, in relational algebra, we use rules, some of them similar to $ab + ac = a(b + c)$ in order to simplify work for the computer.

I have been told that tens of billions of pounds *per annum* are spent on relational databases. Not being an economist, I have no idea what this means, but there is no doubt that this is a huge industry. The moral of the story is that one should never underestimate the usefulness of theory in computer science.

Let's look at what relational databases are and why we need them.

6.2 The Relational Data Model

It will not hurt us to repeat, using a different example, the justification for the relational model. Suppose we want to describe students, courses, and the marks they get in courses. This is a prosaic database example, and we'll make some hugely simplifying assumptions.

- Students take a course just once. Students have a number, which is used to identify them, a name, and an email address.
- A course has a name, one teacher and is taught in just one room (why can't UoE arrange this!)
- A student gets a single mark for a course he or she takes.

If we were representing this in XML, we might organise all this information around individual students as shown in Figure 1 (left) On the other hand, for the purpose of providing class lists, we might organise the information around courses.as in in Figure 1 (right)

These representations are both useful in specific contexts, and last term we saw how to transform one into another through the use of XQuery. However they are both problematic. They both contain *redundant* information. In the "student-centric" representation course information is repeated, and in the "course-centric" representation student information is repeated. Moreover, there is the problem that, in the "course-centric" representation, a student has to be taking at least one course in order to exist in the database.

The solution is, as we saw, to separate the student and course information. Neglecting the information about marks for the moment we get two relatively "flat" pieces of XML, which we can represent as *tables* as shown in the `students` and `courses` tables of Figure 2.

Unfortunately the information about marks, or, more generally what student is taking what course, is now lost, and it cannot be easily added to either the `students` or the `courses` tables. The only way round this is to add a new table, `takes` which describes which students were taking which courses and the marks they obtained. Note, however that we have now avoided the problems of redundancy and the dependence of student information on the presence of course information. Student and course information is localised, and there is no need for a student to be enrolled in a course for the student to appear in the database.

This, then is an example of a relational database. Some terminology:

```
⟨students⟩
  ⟨student⟩
    ⟨id⟩ S0123 ⟨/id⟩
    ⟨name⟩ White ⟨/name⟩
    ⟨email⟩ sw@dot.com ⟨/email⟩
    ⟨courses⟩
        ⟨course⟩
          ⟨cname⟩ CompSci3 ⟨/cname⟩
        ⟨teacher⟩ Bashful ⟨/teacher⟩
        ⟨room⟩ B34 ⟨/room⟩
        ⟨marks⟩ 75 ⟨/marks⟩
      ⟨/course⟩
      ⟨course⟩
        ⟨cname⟩ Hist4 ⟨/cname⟩
        ⟨teacher⟩ Grumpy ⟨/teacher⟩
        ⟨room⟩ C21 ⟨/room⟩
        ⟨marks⟩ 62 ⟨/marks⟩
      ⟨/course⟩
    ⟨/courses⟩
  ⟨/student⟩
  ⟨student⟩
    ⟨id⟩ S0456 ⟨/id⟩
    ⟨name⟩ Prince ⟨/name⟩
    ⟨email⟩ prince@dd.edu ⟨/email⟩
    ⟨courses⟩
      ⟨course⟩
        ⟨cname⟩ CompSci3 ⟨/cname⟩
        ⟨teacher⟩ Bashful ⟨/teacher⟩
        ⟨room⟩ B34 ⟨/room⟩
        ⟨marks⟩ 60 ⟨/marks⟩
      ⟨/course⟩
      ⟨course⟩
        ⟨cname⟩ French3 ⟨/cname⟩
        ⟨teacher⟩ Sneezy ⟨/teacher⟩
        ⟨room⟩ C17 ⟨/room⟩
        ⟨marks⟩ 70 ⟨/marks⟩
      ⟨/course⟩
    ⟨/courses⟩
    ...
⟨/db⟩
```

```
⟨courses⟩
  ⟨course⟩
    ⟨cname⟩ CompSci3 ⟨/cname⟩
    ⟨teacher⟩ Bashful ⟨/teacher⟩
    ⟨room⟩ B34 ⟨/room⟩
    ⟨students⟩
      ⟨student⟩
        ⟨id⟩ S0123 ⟨/id⟩
        ⟨name⟩ White ⟨/name⟩
        ⟨email⟩ sw@dot.com ⟨/email⟩
        ⟨marks⟩ 75 ⟨/marks⟩
      ⟨/student⟩
      ⟨student⟩
        ⟨id⟩ S0456 ⟨/id⟩
        ⟨name⟩ Prince ⟨/name⟩
        ⟨email⟩ prince@dd.edu ⟨/email⟩
        ⟨marks⟩ 60 ⟨/marks⟩
      ⟨/student⟩
      ...
    ⟨/students⟩
  ⟨/course⟩
  ⟨course⟩
    ⟨cname⟩ Hist4 ⟨/cname⟩
    ⟨teacher⟩ Grumpy ⟨/teacher⟩
    ⟨room⟩ C21 ⟨/room⟩
    ⟨students⟩
    ...
    ⟨/students⟩
  ⟨/course⟩
  ...  ⟨/courses⟩
```

Figure 1: Two XML representations of student-course data

students:

id	name	email
S0123	White	sw@dot.com
S0456	Prince	prince@dd.edu
...	...	...

courses:

cname	teacher	room
CompSci3	Bashful	B34
Hist4	Grumpy	C21
French3	Sneezy	C17
...	...	...

takes:

id	cname	marks
S0123	CompSci3	75
S0456	Hist4	62
S0123	French3	70
S0456	CompSci3	60
...	...	...

Figure 2: Relational representation of student-course data

- The tables are also called *relations*.
- The rows of the tables are also called *tuples*.
- The column names are also called *attributes* or sometimes *fields*.

More importantly we need to consider the precise meaning of a table. In the relational model it is assumed that the order of the rows is unimportant and that no two rows are identical. In other words, a table is a *set* of tuples. Also, the order in which the columns appear is unimportant. For a given table, each column should have a distinct name.

When we study SQL in detail, we shall see that some of these assumptions may not hold. But before looking at this query language, let us look a bit more at the *structure* of this database.

When databases are first designed, they start out life with no data in them. Nevertheless we need to describe them to the database management system. What we do is something like writing a class or type declaration for each table. For SQL this is done in something called a *data definition language* (DDL). Let us look at why the language is needed. Database people, when they write down a database design often give a rather terse description of the form:

```
students(id,name,email)
courses(cname,teacher,room)
takes(id,cname,marks)
```

From that simple description of the tables, one can start to "populate" or create an *instance* of the database. But can we put any data we want in the tables? The answer is no. To make sense there are a number of additional conditions that we need to enforce when we add tuples to, or modify, tables.

Types Each column has a type. They are familiar from programming languages. They are things like `INTEGER`, `CHAR(22)` for fixed-length character strings, `TEXT` of arbitrarily long strings, `DATE` etc.

Keys We have assumed that the tuples in the `student` table are uniquely identified by their `id` attribute and those in the `courses` table are uniquely identified by their `cname` attribute. If we didn't make this assumption it would be hard to decide what the `takes` table means.

When we want an attribute such as `id` uniquely to identify a `student` tuple we say `id` is a *key* for `student`. What this means is that at most one `student` tuple can have a given `id`.

What is the key for the `takes` table? Neither `id` nor `cname` alone can serve as a key. However the pair of attributes is what we need in order to determine a mark. This means that the more general definition of a key is a *set of attributes* whose value uniquely determines a tuple in a table.

Note that a key could consist of all the attributes of a table. If we were not interested in recording marks, but just which student was taking what course, we would have a table with structure `takes(id, cname)` and the set of attributes {`id, cname`} would be the key for the table.

Foreign Keys Another assumption we will probably want to make is that every `id` in the `takes` table also occurs as an `id` in the `student` table. If you like the analogy, `id`, because it is a key, acts as a "pointer" into the `student` table, and we don't want any "dangling" pointers from the `takes` table.

In SQL DDL we can specify all these constraints. This is what the DDL would look like:

```
CREATE TABLE students (  id CHAR(5),
                         name TEXT,
                         email TEXT,
                         PRIMARY KEY (id) )

CREATE TABLE courses (  cname TEXT,
                        teacher TEXT,
                        room TEXT,
                        PRIMARY KEY (cname) )

CREATE TABLE takes (  id CHAR(5),
                      cname TEXT,
                      marks INTEGER,
                      PRIMARY KEY (id, cname),
                      FOREIGN KEY (id) REFERENCES students(id),
                      FOREIGN KEY (cname) REFERENCES courses(cname) )
```

The syntax should be self-explanatory. The reason for a rather elaborate syntax for `FOREIGN KEY` declarations is that the column being referenced may not have the same name as the column that is doing the referencing, though in this case they are the same.

Another part of SQL is called the Data Manipulation Language (DML) which consists of expressions for querying and modifying the database. Whenever a modification of the database is attempted, SQL should check whether the constraints expressed in the DDL will be met *after* the update is made. If not, the update will fail.

Exercises.

- We want to augment our database with a table for `teachers` which looks like this:

tname	office	telephone
Dopey	D2	667 1234
Grumpy	D3	667 5678
Sneezy	B21	668 1212
Bashful	B22	668 9998

  Write down the additional SQL DDL.

- Suppose we change some assumptions. Courses are taught every year, and a course may have different teachers in different years (but just one for a given year). Also a student may take the same course in different years, and for each year the marks should be recorded.

  Rewrite the DDL.

- Consider updates that (a) delete a tuple and (b) add a tuple. Briefly describe how they could violate a key or a foreign key constraint.

Relational database design, which is what we have just been thinking about, is incredibly important. We shall return to it later, but first let us discuss how we query the data.