

When you generate a computer program, you often start by structuring the data in a database. For instance in Access, Paradox, Interbase, MySQL, SQL Server, Blackfish or Oracle – all examples of relational databases. With these databases, the data are put into tables. Some thirty years ago a theory was developed about how to best arrange these tables: normalisation of relational databases.

Hierarchic or relational

Data in a database can be structured in several ways. Two commonly used forms are hierarchic and relational structures. An example of a hierarchic structure is the ordering of plants and animals. For instance, the species 'lion' and 'cat' are in the cat family, which in turn are mammals. Using XML, this hierarchy could be represented by nesting the less abstract term in the more abstract ones:

```
<mammals>
  <cat_family>
    <species>lion</species>
    <species>cat</species>
  </cat_family>
</mammals>
```

In a relational database, the data are put into tables that are related. The data above could be put into a table as follows:

Table: mammals

| species | family |
|---------|--------|
| lion | cat |
| cat | cat |

A big advantage of saving data in tables is that data retrieval is very easy and fast, without any need for the software to first traverse all kinds of tree structures. A disadvantage is that many sorts of intuitive, often hierarchic relationships are less clearly presented. The relational model is presently the most commonly used data model. But how can you best arrange the data in the various tables?

1st normal form: atomic fields

Suppose an institution organises courses in computer science. The data to be recorded are which students are taking which courses. You could create a table with two columns (fields): the course name and the names and addresses of the students. This would produce something like:

Table: courses

| course name | Names_and_Addresses_of_Students |
|-------------|--|
| Programming | Fatima Guermat, Streetway 16, Birmingham; Lucky Luck, Bingoroad10, Fortune City; Chris... etc. |
| Web Design | John Shepard, Wooldrive 33-a, Heather; Habiba Hadith, Beachpark 14, Bristol; Simone... etc. |
| Databases | Here again a list of names and addresses separated by semicolons. |

With this arrangement, however, it is difficult to select all students from a certain city or change the address of a particular person. This would be much easier if only indivisible values were put in the fields. For instance: the name and address of only one student in the student names and address field, instead of data for several students, separated by semicolons. This would give us something like this:

Table: courses

| Course_Name | Names_and_Addresses_of_Students |
|-------------|--|
| Programming | Fatima Guermat, Streetway 16, Birmingham |
| Programming | Lucky Luck, Bingoroad10, Fortune City |
| Programming | Chris... etc. |
| Web Design | John Shepard, Wooldrive 33-a, Heather |
| Web Design | Habiba Hadith, Beachpark 14, Bristol |
| Web Design | Simone... etc |
| Databases | And also here just one name and address |
| Databases | Every student has her own row (record) |

Depending on how the data will be used, you can split the structure into more fields. For instance: name, address, city. Or even further, for instance: first name, middle name, family name, street, number, number suffix, postal code, city. The degree of splitting depends entirely on the expected use of the data, because that is the only basis for deciding which value are atomic (indivisible). This kind of analysis is therefore not a mechanical matter; there will always be some interpretation involved. In the case of names and addresses, however, it is generally good practice to at least split the data into the first name or initials, family name, address, postal code, and city.

Avoid redundancy

Up to now we have split the data into different fields and rows (records) in a single table. Next let's look at how we can distribute the data over several tables. Suppose that a student in our example is taking several courses. In this case, his or her address will also appear in the table more than once. This could cause a lot of trouble: for instance, if by accident the address is only updated in one location but not in the others. Then we would be faced with two different addresses... which one is correct? This kind of duplicate (and thus superfluous) data is called 'redundant'. Another example: if more data will be recorded about the courses (for instance: the date, location and fee), the same data will be repeated over and over again. The solution is to create several different tables: one containing data about students and addresses, another one with course data, and yet another one that couples courses with students. In schematic form, this would be something like:

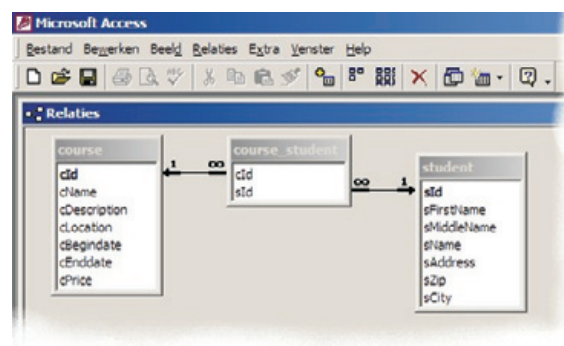


Figure1: Relationship diagram in MS Access

A scheme like this is called an entity relationship diagram (in this case generated in Access). The name of the table is put at the head of the box. The field names are listed below, with the key field in bold or underlined. Each record in the course table is uniquely identified by the cid field, and each record in the student table is identified by the sid field. This sort of unique record identifier is called a key, and

Practical database normalisation (2)



In this way a student can attend different courses and a course can have more than one student (we even hope that will be the case). This is called a many-to-many relationship between courses and students. This many-to-many relationship can be split into two one-to-many relationships: the new `course_student` table shows that a student can take more than one course AND that a course can have more than one student. In the relationship diagram, an arrow is drawn on the 1-side.

Splitting tables: 2nd and 3rd normal forms

The theory of database normalisation gives us formal rules on how and when tables should be split. Often an intuitive approach takes us quite a way in the right direction:

- Put distinct 'things' (entities) in their own table.
- Repetition of the same data is often an indication that something should be split.
- The same holds for columns with many null entries.
- Look for one-to-many relationships in a table and split them into two tables.
- Look for many-to-many relationships in a table and split the table into three separate tables, as in the example above.

Each record in a table should have a unique key to identify the record, and possibly a set of mutually independent fields (attributes) that further describe the record. This concept is formalised in the theory of normalisation (relational calculus), but in everyday practice the procedure described below will suffice for gradual transformation from the 1st normal form to further normalisation.

2nd normal form

(2NF): Put all fields that are not dependent on the entire key in a separate table.

Suppose we put the locations for our courses in a table with the following fields: `building`, `room`, and `address`. The key (unique record identifier) is a combination of the fields 'building' and 'room'. You can write this as follows:

```
locations (building, room, address)
```

However, if each building has only one address, then the address field is dependent on only part of the key. This could be split into two tables as follows:

```
building (bId, bName, bAddress)
locations (bId, room)
```

Here we created a new unique key (`bId`) in the 'building' table, so that we can also put different buildings with the same name in the table ('Crossroads' in Brighton and Lancashire, for instance). Usually an auto-increment field is used for such a key. A short key like this is easier to use as a reference in other tables. In the `locations` table, such an identification number will also always be used in practice, and it provides an easy way to reference a location in another table. As a result of using an extra field as key, there are now two possible keys to uniquely designate a record: the new identification number, which is in fact used as the key (the primary key), and the combination of the fields 'bId' (building identification) and 'room'. All possible keys are called candidate keys, and the key that is actually used is called the primary key.

3rd normal form

(3NF): Put transitive dependent attributes in another table. This means fields that are also dependent on a non-key field.

This usually implies a one-to-one relationship. Suppose our example institution uses the following database table, consisting of three 'foreign' keys, to reference records in other tables:

```
course_location_student (courseId, locationId, studentId)
```

With this table, we can record which courses are given where and which students take which courses. If each course is always given in only one specific location and this is not expected to change, it is not necessary to state the location identification for every occurrence in this table. In this case, it would be better to move it to the course table. This again is a matter of estimating how things will develop in the future – more a subtle art than a mechanical procedure.

There is another normal form that combines 2nd and 3rd normal forms and takes into account that there can be more than one (candidate) key:

- Boyce/Codd normal form (BCNF)
- Ensure that all fields in a table are only determined by a candidate key.


If a table is in 3NF (or BCNF), you can say that it is normalised. This means that the major redundancy traps will be avoided. There are also fourth and fifth normal forms, but they are simply extensions of the dependency concepts of the first three normal forms for use in relatively uncommon situations.

'Every advantage has its disadvantage' (Élk voordeel heb z'n nadeel')

This is a literal translation of a well-known statement by the famous Dutch football player Johan Cruyff.

Splitting the data into several tables has the advantage of reducing redundancy, but it also makes table management more difficult. For instance, tables have to be joined temporarily to obtain any useful information from them – a pile of identification numbers of courses and students doesn't say much. You also have to be careful not to discard necessary data. For instance, if a location identification number is used somewhere in a table, this information must remain available in the location table. The latter restriction is called 'referential integrity', and it can be enforced automatically in most database management systems.

Literature

- Almost every book about using databases has something to say about normalisation. Tons of information can also be found on the internet, e.g.:
http://en.wikipedia.org/wiki/Database_normalization , 
www.serverwatch.com/tutorials/article.php/1549781 ,
<http://dev.mysql.com/tech-resources/articles/intro-to-normalization.html>
or
www.databasedev.co.uk/database_normalization_process.html
- A classic book, first published 1975 but still beingis: C. J. Date, An Introduction to Database Systems (Addison-Wesley, 8th edition, July 2003). There are more publications from the same author the relational model. Date worked together with Codd, the founder of normalisation theory.

Herman Peeren, Rotterdam, December 2007.

herman@yep.r.nl





Readers

Please send your questions and comments to:

email office@blaisepascal.eu

or to: Edelstenenbaan 21 3402 XA IJsselstein, The Netherlands

Tel.: + 31 (0) 30 68.76.981 Mobile: + 31 (0) 6 21.23.62.68