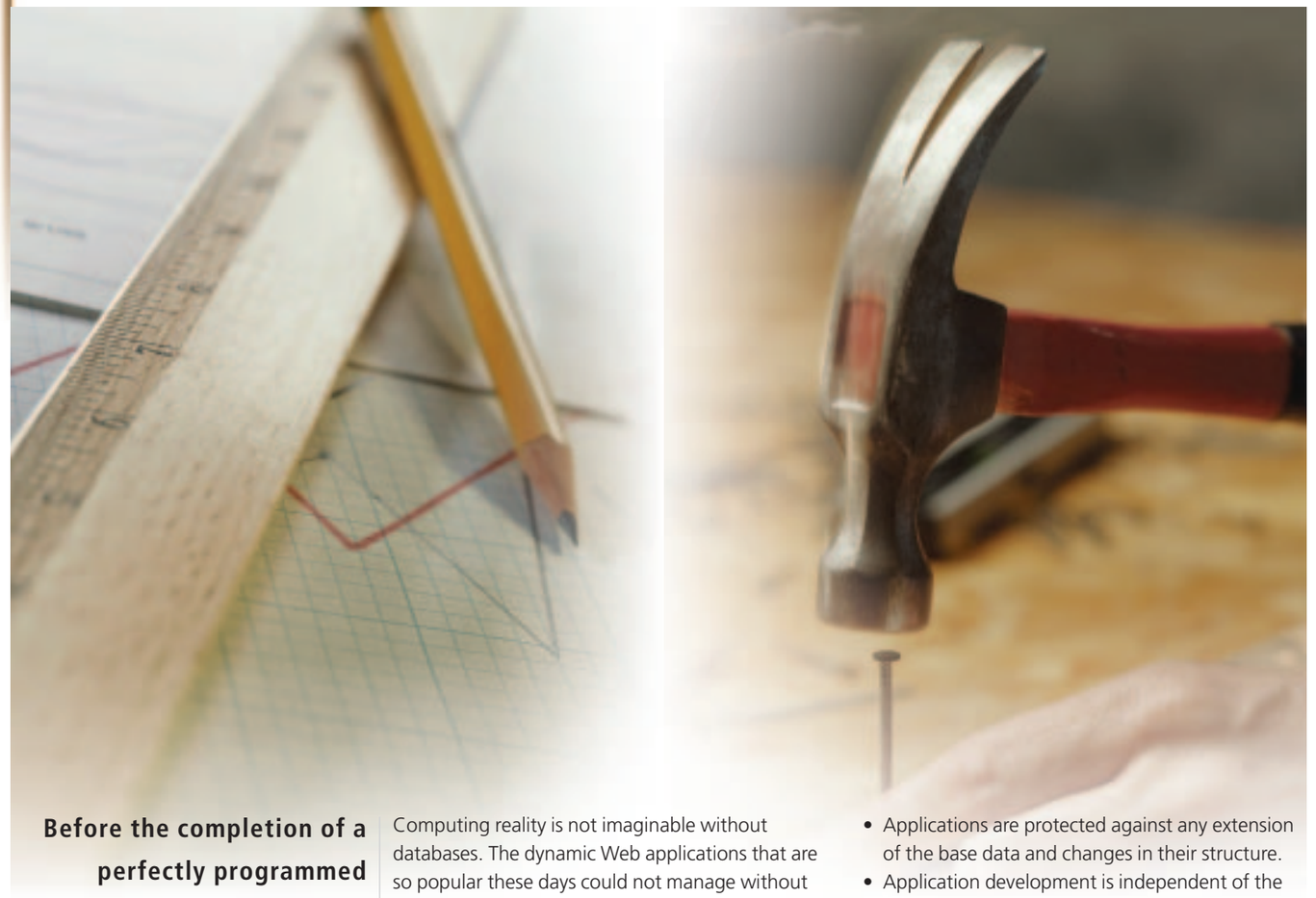
Designing Data Models

DATA WORKSHOP

BERNHARD ROHRIG



Before the completion of a perfectly programmed database we have the hard work of database modelling. This article shows ways and means of achieving a clean database design.

Computing reality is not imaginable without databases. The dynamic Web applications that are so popular these days could not manage without them. There is now a large selection of server software, tools and development environments in which to operate database systems under Linux. But before the implementation of a specific database server can be discussed, it is necessary to agree on the model that can be represented in the data structures.

Put simply, a database is a collection of non-redundant data that is used by different applications in parallel and simultaneously. Compared to classic data storage forms, where each application had its means of storing data, they offer the following advantages:

- Applications are protected against any extension of the base data and changes in their structure.
- Application development is independent of the way the data is organised and accessed.
- Applications are no longer linked in terms of changes to individual applications.
- Data consistency is provided centrally by the database itself.

This allows more flexible data utilisation, avoids wasting disk space and saves a lot of program maintenance and development time.

Down with redundancy

Why is it so important to eliminate redundancy? Redundancy in this context means that the same

information is stored several times in the same database. Should individual representations occur of the same fact that contradict each other (amendments carried out in one place, no amendment in another), we call this data inconsistency. Once a database has become inconsistent it takes a lot of effort to restore the desired integrity. By avoiding redundancy in the first place, inconsistencies can be ruled out from the start.

A considerable number of processes have been developed to eliminate data redundancy. Many of these are based on complicated mathematical procedures (relational algebra) and the resulting models are not always very clear. However, for practical purposes it is sufficient to know the most important methods with which to avoid a large proportion of redundancy-pitfalls, as these are not always apparent at first glance.

A central tool in this is entity relationship modelling on the one hand, and the use of normalisation theory on the other.

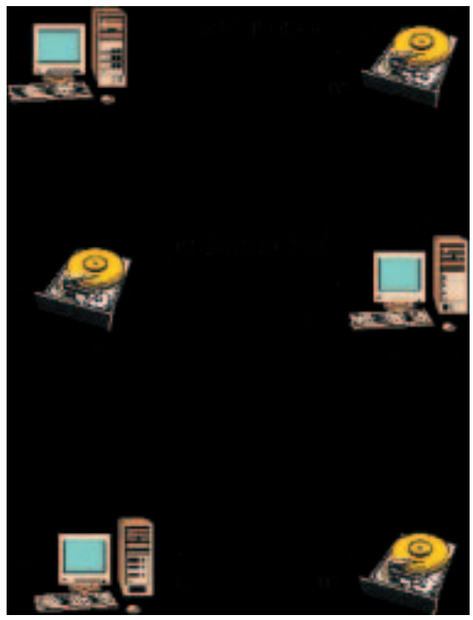


Figure 1: A simple structural model (entity relationship)

First approaches

As a database developer you are faced first of all with producing a more or less exact description of the section of reality which is going to be represented in the data structures you are developing. We shall use the administration of data relating to PCs and operating systems as an example.

Let's assume that you like to experiment with different Linux distributions and other operating systems, and that you own several PCs for this purpose. Since the built-in hard disks are normally not sufficient for such experiments some of the PCs have hot swap facilities, which allow the easy installation of hard disks. It is of course also possible to install several operating systems on one hard disk.

In order to represent real objects and their relationships to each other as a structured data set, the colloquial terms need to be classified. That is a first step on the way from chaotic diversity to a model based strictly on logic.

It is useful to classify according to object class, attribute and relationship. Table 1 shows a suggestion for our example. If you tried to draw up this sort of overview yourself, you might come up with different results, because the assignment cannot always be unambiguous.

If a given fact is classified as a relationship in one context, it can also appear as an object in another, just as a seemingly independent object can suddenly become merely a simple attribute of another object or vice versa. An example: on the one hand, the object *vendor* is an object class that has specific attributes (for instance *address*, *telephone number*). On the other hand it can sometimes be sufficient to regard *vendor* as an attribute of *PC*.

What this means is that the model doesn't have to be correct, it has to be consistent. This is not about truth, but rather functionality. The same or a roughly similar section of the real world can be modelled quite differently in two separate databases - and both models could still serve their purpose. In the rest of this article, the information from table 1 is used to draw up a rough structural diagram, the entity relationship model, or ER model for short.

Elements and relationships

An entity relationship is a technique that makes it possible to express the relationships between facts

Table 1: Classification of modelling objects

Real Term	Object Class	Attribute	Relationship
PC	x		
Hard Disk	x		
PC Type		x	
Hard Disk Type		x	
Vendor	x	x	
PC Hard Disk			x
Vendor Address		x	
Operating System	x		
Operating System Supplier		x	
Operating System on Hard Disk			x

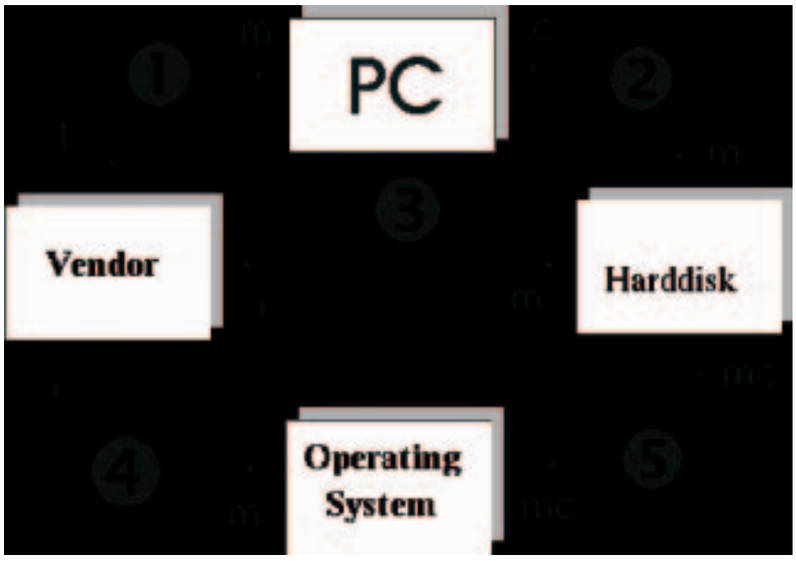


Figure 2: ER model for administration of PC/operating system

and transactions in the real world within a data model. The defined object classes with their attributes are regarded as units or entities, between which a certain relationship exists. This relationship is represented in ER-diagrams, where we distinguish between four classes of relationships, listed in table 2.

You can see the ER-diagrams for the relationships between the object classes *PC* and *vendor* in figure 1. The following relationships exist:

- One PC has one or more hard disks (m).
- Every hard disk is installed on a PC or lying in a cupboard (c).

Overall, a relationship of the type c : m (c to m) exists between hard disk and PC. Such relationships are then defined for all object classes of the data model and summarised in a structural ER-diagram (see figure 2).

Again, there is no definitive solution. Different database designers arrive at different layouts given the same facts. The only important thing is that the resulting model is suitable for the intended purpose and that any redundancy is avoided.

In this type of diagram you will find three main forms of interrelations:

- hose featuring relationship type 1.
- Those featuring relationship type c, but not relationship type 1.
- Those featuring only relationship types m and mc. Relationships 1, 3 and 4 in figure 2 belong to the first category, relationship 2 to the second and relationship 5 to the third. Their subsequent handling depends on the category. However, this requires us to know a bit more about the internal structure of the data model.

A look inside

In relational data models (SQL databases) the data of each object class are collected in a sort of table, called a relationship. The table rows, also called data tuples or data records, store the object details, while the columns - also called fields or attributes - contain the respective value for each attribute. This is similar for post-relational or object oriented databases, except that storage is not two but multi-dimensional.

It is important to note that the storage sequence of the rows and the field is of no external significance; they are also referred to as unordered tuples or unordered attributes.

Object attributes are identified through the relevant field names , objects through the primary key. A primary key is a sort of index that has a unique value or combination of values for each data record. Normally an additional attribute is introduced for this purpose, whose value is often automatically assigned by the database when a record is created. Using this primary key, each data record in a table can be uniquely identified. This also means that a primary key field must not contain a NULL marker, indicating no value, in any of the records.

Table 2: Association types in ER models

Symbol	Type of relationship	Example
1	to one	Each PC has one vendor.
c	to one or none	Each hard disk is in one PC or in none.
m	to at least one	Each PC has at least one hard disk.
mc	to none, one or several	An operating system is on no, one or several hard disks.

Table 3: Basic relation PC

Attribute	Content
PCID	PC identification, primary key
vendor	PC manufacturer
type	PC type
purdate	Purchase date

Table 4: Basic relation HD (Hard Disk)

Attribute	Content
HDID	Hard disk identification, primary key
vendor	Hard disk manufacturer
type	Hard disk type
capMB	Hard disk capacity in MB
PCID	Built into which PC?

Table 5: Basic relation OS (Operating System)

Attribute	Content
OSID	Operating system ID, primary key
vendor	Manufacturer/distributor
namev	Name/version of operating system
HDID	Installed on which hard disk?
date	Date of OS installation

Table 6: Basic relation Vendor

Attribute	Content
VID	Vendor ID, primary key
vname	Vendor company
vaddress	Address
vcity	Town
vzip	Postcode
vtel	Telephone number

The primary key also allows links with related data records in other tables using a foreign key. This is simply a reference to the primary key of one table through a corresponding value in a field of another table. The database has to know, of course, which field is related to which other field.

The individual SQL dialects offer appropriate facilities for this purpose. An important rule for all foreign key relationships is referential integrity, which stipulates that foreign key fields in a dependant table must only contain NULL values and values that already exist as primary key values in the referenced table, but never other values which might possibly appear there at some point in the future.

Any good database system will monitor the compliance with referential integrity independently, which frees application programmers from this task.

Basic relations in detail

The relationships between primary and foreign keys for our data objects can now be defined on the basis of the ER model. The first step is to list all object classes with their corresponding attributes according to tables 3 to 5. To start off, we are going to assume that the attribute *vendor* is common to the three object classes.

This results in a relatively simple data model. It is apparent that the relationships of the objects to each other can be represented by rather simple key relations. Each table contains a numeric data record ID as its unique primary key field. Hard disks are linked to the corresponding data records in the PC table through the PC-ID. This makes it possible to have more than one hard disk per PC and to represent this fact in the database. The relationship between the individual operating systems and the hard disks on which they are installed is organised in a similar way.

As pretty as this model is to look at, it does have its hidden dangers, which we are now going to expose and remove step by step.

Consolidating the model

Even at first glance we can spot one redundancy, which could cause a lot of aggravation later during the operation of the database. The attribute *vendor* for the manufacturer, distributor or supplier of a device or program is only unproblematic as long as its values are not repeated in one or more tables. However, this is bound to happen once the latest version of a much loved (or hated) operating system is released, if not before, for instance when the letters IBM appear in a second table.

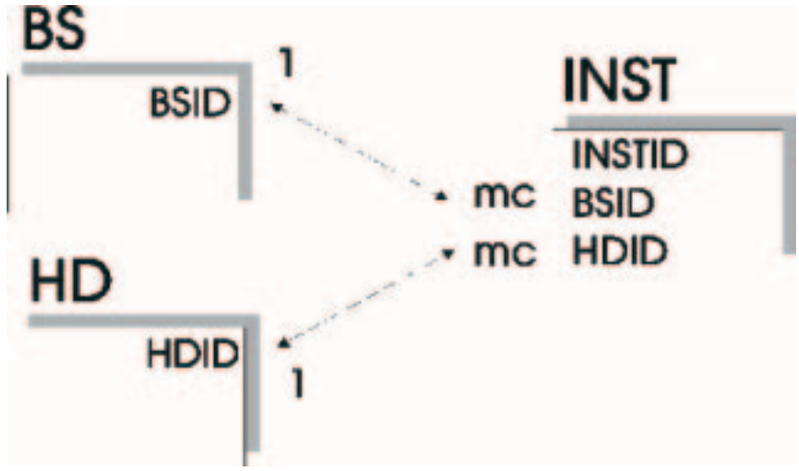


Figure 3: Resolving the mc:mc relationship in the ER model

What happens if a manufacturer's name or an address that is stored with it changes? The corresponding fields in all affected databases must be amended. If just one of them is forgotten, the data will be inconsistent. This redundancy is resolved relatively easily, by introducing an additional object class *vendor*. The corresponding basic relation is shown in table 6. The attribute *vendor* in the three other tables needs to be changed to the foreign key *VID* accordingly.

Other redundancy problems are not quite so easy to spot. Therefore we will fall back on methods from ER modelling and normalization theory. The ER model enables us to create clean primary/foreign key relations with three simple rules:

- Rule 1 applies to all 1:x relationships; in figure 2 those are relations 1, 3 and 4. This rule states that the primary key on the 1-side - here always *VID* from *vendor* - must be included as a foreign key attribute in the basic relation on the x-side, where we have either m, c or mc. NULL markers are not permitted for these foreign keys. We have already complied with this rule in the previous step.
- Rule 2 applies to all relationships that have c on at least one side, with the exception of 1:c relationships, which are already covered by rule 1. This rule states that the primary key attribute on the c-side will be included as a foreign key attribute into the basic relation on the other side (m, mc) of the relationship. NULL markers are permitted for these foreign key attributes. To comply with this rule we must allow the property *NULL marker permitted* for the attribute *PCID* in the basic relation *HD*. In MySQL, probably the most widespread SQL version under Linux, this is achieved by the following type of definition:

```
create table HD ... PCID int null;
```

Table 7: Additional basic relation INSTALLATION

Attribute	Content
INSTID	ID of the OS installation, primary key
OSID	foreign key, ID of the installed operating system
HDID	foreign key, ID of the hard disk, on which OS has been installed

- Rule 3 applies to the remaining relationships in the ER model, those that do not feature a 1 or c-side. An example is relation 5 in figure 2, a mc:mc relationship. Each hard disk can contain one or more operating systems, and sometimes it can be empty, while each operating system is installed on one or more hard disks or lying around unused.

This somewhat complicated relationship must be untangled a bit for a proper relational model, as it cannot be established quite so simply using primary and foreign keys. Rather, we require an additional interim table, which apart from its own primary key contains nothing except the combined primary keys of the two other tables. Let's resolve the mc:mc relationship into two 1:mc relationships by introducing a new basic relation *INSTALLATION* along the lines of table 7.

The relationship between the entities hard disk, operating system and installation can now be expressed as shown in figure 3. At the same time the attribute HDID must be removed from the operating system table, as there is no longer a direct link between an operating system and the hard disk on which it is installed.

How normal would you like it?

To put the finishing touches to our data model the individual basic relations have to be normalized. This means processing them according to certain formal mathematical criteria that have become known as normalization theory. These contribute to the detection of hidden redundancies within the model and thereby help to avoid inconsistencies in the underlying data. There are several normal forms of base tables. For practical application it is sufficient to check compliance with the first, second and third normal form for each of the tables involved. Normally the database designer will use a large quantity of test data supplied by the client. These should contain as many characteristic combinations of attribute values for the individual object properties as possible.

There is no space here for a more detailed description, just some methodological hints. A relation is considered to be in the first normal form when it does not contain any attributes with multiple values. These are fields in a table containing several values. It is possible to imagine a violation of the first normal form by table 3, if there are several PCs of different types from one vendor, and the corresponding entries are in one and the same row. This can be easily avoided by consistently assigning each PC its own row in the table.

The other data objects (tables) should be treated in a similar way to comply with the first normal form. This requirement may appear trivial, as it is usually adhered to instinctively by most database designers. However, it is useful to be aware of it in order to recognise a possible violation of the requirement in an actual set of data and to avoid it.

A table is in the second normal form if it has a primary key that consists of a single attribute. It is therefore useful always to work with such primary keys, as we have done in our example. You can find other possibilities in the literature. You can also find out there how to convert non-compliant tables into second normal form.

Compliance with the third normal form means that no attribute (table field) must be functionally dependent on any other attribute apart from the primary key. This is hard to determine without actual data.

The finished data model

A few comments about the tables in our example. Table 7 is in the third normal form from the beginning, due to its structure. Table 6 could lead to inconsistencies with large data volumes, as for instance the post code can be functionally dependant on the address. We can put a stop to that without any problems by simply splitting the table further.

In the same way it would be possible to create the facility to store several telephone numbers for one vendor (without violating the first normal form). Table 5 will be in the third normal form as soon as one vendor offers at least two operating systems - which has been known to happen.

It is just as simply to prove compliance with the third normal form for table 4. The only chance of a violation would be a fixed assignment between *PCID* and *HDID*; as the combination of the other attributes would then also seem to depend on *HDID*. However, this is ruled out by the assumption that we are using swappable hard disks in at least some of the PCs.

Finally, with the appropriate data, it would be possible to prove for table 3 that no attribute is dependant from any other apart from the *PCID*.

Thus our data model is solid and the way is clear for a conversion using one of the current databases. Admittedly things don't always go as smoothly as in our example. Solutions and suggestions for more complicated cases can be found in the literature. ■

Info

Dr. Bernhard Rohrig: Datenbanken mit Linux. C&L Verlag, Vaterstetten, 1998. ISBN 3-932311-32-9

■

The author

Dr. Bernhard Rohrig has written several books on Linux and Unix and can be contacted via the Internet under http://www.roehrig.com. You can also find out there what else he gets up to apart from writing books.