Normalisation

Introduction

ER modelling relies primarily on the intuition of the analyst to identify “significant” entities in the environment and represent them as tables. This is clearly not an exact science, and there are lots of opportunities for error. Because relational databases are inherently mathematical, most of these errors can be identified and resolved using some simple rules. In database terminology, this process is called *normalisation*, because a data structure can be improved by moving through a series of *normal forms*. Each normal form has certain characteristics which rule out particular types of problem.

Sometimes it can be difficult to understand the need for rules, because so much can be done intuitively, and the remaining problems are minor. Think of it like learning to drive: once you have been driving for a year or two, you instinctively do the right thing at a junction or when you want to overtake the car in front. When you first start, however, you learn to apply a lot of simple procedures such as *mirror-signal-manoeuvre* (ask a friend about this if you don't drive yourself). Normalisation is a little like this. Once you have built up some experience, you will have an instinctive knowledge of what makes a good database structure. Until then, you need some rules to help you reach a good quality result.

Normalisation is closely related to the idea of efficiency and the elimination of redundancy. This is something that we have mentioned briefly before, but now we will concentrate more on the details. To start with, we can identify three principles of efficient database structures:

- Closely related attributes are found in the same relation (table)
- Each relation contains a *minimum* number of attributes
- Each attribute value is stored a *minimum* number of times (ideally only once)

When they are written down, general principles often sound trivial; however, there will be many occasions on which you are faced with a practical question of database design and the answer is not immediately obvious. These general principles are your first line of defence and may allow you to resolve some of those design dilemmas easily.

Normalisation is also related to the idea of data integrity. Whatever the nature of the data being stored, it is important that it is correct, and that all relationships are correctly maintained. Because the job of designing database is very different from the later use of that database, there are certain types of structural error that are not obvious when you are only dealing with small amounts of test data. The rules of normalisation help you to avoid those errors, and to produce database structures that safeguard data integrity.

Data anomalies

The most fundamental reason for using normalisation is to eliminate undesirable redundancy in a data structure. Where redundancy exists, there is a risk that certain operations will create data anomalies. The term *anomaly* is used to describe something that is not in line with expectations. In a relational database, we expect that all of the data stored will be a) correct and b) internally consistent.

To illustrate what is meant by redundancy and data anomalies, let's imagine an example in which consultants are assigned to projects. The data about consultants and projects could be stored in a single table as shown below.

cons_proj(<u>cons_id</u>, <u>proj_code</u>, cons_name, grade, daily_rate, days, proj_name, start_date, end_date)

cons_id	cons_name	grade	daily_rate	days	proj_code	proj_name	start_date	end_date
123	McAlastair	Senior	750	42	AB66	Goldfish	01-OCT-2011	30-JUN-2012
143	McBeth	Executive	800	12	DD25	Silverbird	01-JAN-2012	31-MAY-2012
125	McCluskey	Junior	500	80	AB66	Goldfish	01-OCT-2011	30-JUN-2012
163	McDowell	Senior	750	65	GC31	Bronzecat	15-FEB-2012	15-JAN-2013
167	McEwan	Executive	800	10	GC31	Bronzecat	15-FEB-2012	15-JAN-2013
123	McAlastair	Senior	750	42	GC31	Bronzecat	15-FEB-2012	15-JAN-2013

From this table, we can see that two consultants are assigned to project Goldfish, one to Silverbird and three to Bronzecat. We can also see from looking at any record the details of the project that the consultant is assigned to. The data is therefore *correct*, *complete* and *consistent*. However, it is also clear that some data appears in the table more than once. For example, there are two sets of information for project Goldfish, one associated with McAlastair and the other with McCluskey.

Insertion anomalies

To insert the details of a new consultant into the CONS_PROJ table, the details of their project assignment must match those for other consultants on the same project. If we were to assign a new consultant to project Silverbird, for example, there is nothing to prevent us using a different project code, start date or end date. Thus there is the risk of an anomaly - that is to say, two inconsistent records containing conflicting data with no indication of which one is correct.

To insert the details of a new project into the CONS_PROJ table, we would need to use null values for the consultant data until a consultant is assigned. This is not possible since the cons_id column is the primary key.

Deletion anomalies

If McBeth leaves the company, we nee to delete the corresponding record from the table. However, since McBeth is the only consultant assigned to project Silverbird, that means deleting all the details about that project. We have then lost some data from the database, and it is no longer complete.

Modification anomalies

If the details of a project changes, we need to update the database. However, where there are several copies of project data, we need to be careful to update all of them so that the data remains consistent. If any related row is missed in the update, we are again left with inconsistent data.

Solution

The cure for all of these potential error situations is to eliminate the redundancy in the single table by splitting it up. In the tables shown below, the same data is stored, but this time each detail is stored in only one row, and consultant data and project data can be handled separately.

consultant(cons_id, cons_name, grade, daily_rate)
project(proj_code, proj_name, start_date, end_date)
assignment(*cons_id*, *proj_code*, days)

cons_id	cons_name	grade	daily_rate

proj_code	proj_name	start_date	end_date

123	McAlastair	Senior	750
143	McBeth	Executive	800
125	McCluskey	Junior	500
163	McDowell	Senior	750
167	McEwan	Executive	800
123	McAlastair	Senior	750

AB66	Goldfish	01-OCT-2011	30-JUN-2012
DD25	Silverbird	01-JAN-2012	31-MAY-2012
GC31	Bronzecat	15-FEB-2012	15-JAN-2013

cons_id	proj_code	days
123	AB66	42
143	DD25	12
125	AB66	80
163	GC31	65
167	GC31	10
123	GC31	42

In this case, both the problem and the solution are quite simple. You would probably have identified consultants and projects as separate entities from the beginning. However, more complicated cases arise in practice which are less obvious. It is these that the rules of normalisation can help to identify and resolve. Having said that, there is still a redundancy problem with one of the tables above. Before going on, try to identify the problem and suggest a solution.

Functional dependency

In week 2, the term *functional dependency* was introduced in the discussion on keys. For the rest of this set of notes, you need to be clear about the meaning of this term, so if you need to refresh your memory you should review the week 2 notes now.

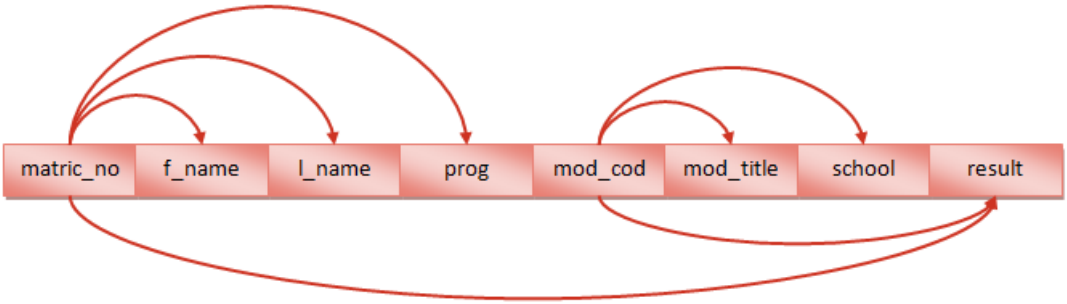
Recall that the purpose of a key is to uniquely identify a row in table. Choosing a good key is usually fairly easy for simple entities, but sometimes we may have a set of attributes that are not yet properly structured. Choosing a key in this situation is more difficult. It can help to draw out a dependency diagram so that you have a visual representation of the dependencies in your data. As an example, takes the data we would need to store about students and the grades they receive for their modules. The set of relevant attributes might be:

student_results(matric_no, first_name, last_name, programme, module_code, module_title, school, result)

Without splitting this set of attributes up, which could be used as a primary key? The technical answer is a minimal set of attributes that functionally determine the rest. In practice, it often helps to look at some example data to answer the question:

matric_no	first_name	last_name	programme	module_code	module_title	school	result
10001234	Ed	Edwards	BEng Computing	SET07101	Agile App Development	Computing	69
10001234	Ed	Edwards	BEng Computing	SET07102	Software Development 1	Computing	72
10010123	Jo	Jones	BIS	SET07101	Agile App Development	Computing	64
10010123	Jo	Jones	BIS	SOE07101	Business Skills	Business	70
10104321	Pete	Peters	Management	SOE07101	Business Skills	Business	70
10012222	Pete	Peters	English	JAC07109	Social Media	Arts	68

Working through the example data, you should quickly find that there is no single attribute that can uniquely identify a row. To select an appropriate combination we can set out the structure as shown below with the various functional dependencies made explicit.



Every attribute is dependent either on matric_no, on module_code or on a combination of the two. Therefore the combination of those two attributes would make a good choice for the primary key for this set of attributes. This can be written:

student_results(matric_no, first_name, last_name, programme, module_code, module_title, school, result)

First normal form

Normalisation proceeds through a series of refinement steps to arrive at a relational structure with the required properties. In the descriptions that follow, try to remember that normalisation is supposed to be a deterministic process with minimum reference to your own preconceptions about entities and relationships. Applied properly, you should find that you are deliberately ignoring certain facts that may be used in a later step. This is simply the nature of a rigorous process. It is always tempting to jump ahead; however, if you do that, you risk missing an important detail along the way.

The starting point for normalisation might be an unanalysed set of attributes that need to be split into separate relations with the appropriate relationships between them. This might be the case, for example, if you were automating an existing system based on paper records or spreadsheets. If the consultant and project data from the previous page were stored in a spreadsheet, for example, it might look like this:

	A	B	C	D	E	F	G	H	I
1	**Project code**	**Name**	**Start**	**End**	**Consultant id**	**Consultant name**	**Grade**	**Daily rate**	**Days**
2	AB66	Goldfish	01-Oct-11	30-Jun-12	123	McAlastair	Senior	750	42
3					125	McCluskey	Junior	500	80
4									
5	DD25	Silverbird	01-Jan-12	31-May-12	143	McBeth	Executive	800	12
6									
7	GC31	Bronzecat	15-Feb-12	15-Jan-13	163	McDowell	Senior	750	65
8					167	McEwan	Executive	800	10
9					123	McAlastair	Senior	750	42

We already have a list of the attributes provided by the column headings, which is useful. Notice however that to improve presentation the owner of the spreadsheet has only listed the project details once, and then added additional rows for further consultants. This is an example of a repeating group of attributes which is important in the refinement of the model to first normal form (1NF).

First normal form

The official definition of 1NF is a relation in which there are no repeating groups. Since every relation should also have a primary key, there are three requirements:

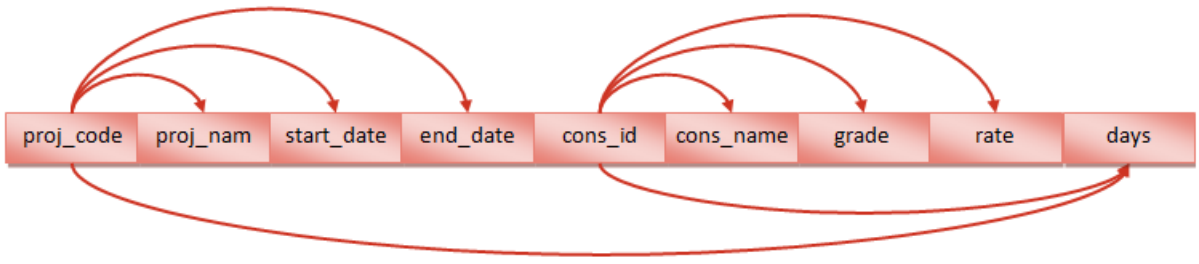
- There must be no repeating groups of attributes
- There must be a primary key
- All non-key attributes must be functionally dependent on the primary key

Check back to the notes on keys if you do not remember what functional dependency is.

The structure above could be written using the usual convention for a schema, but with brackets to indicate a repeating group:

cons_proj(proj_code, proj_name, start_date, end_date, (cons_id, cons_name, grade, daily_rate, days))

There are two common ways to eliminate repeating groups. The first is simply to fill in the blanks left in the spreadsheet. This would give the structure that we started with on the previous page. The second method is to immediately remove the repeating group along with the original primary key into a new relation. Both methods eventually reach the same result; however, the second assumes that you have already identified a primary key. This is not always easy, and the first method might therefore be easier to apply.



Once the blanks are filled in (sometimes referred to as *flattening the table*), we can think about an appropriate primary key. Examining the dependencies we find that the combination of cons_id and proj_code will uniquely identify a row and every other attribute is dependent on one or both of those attributes. We can therefore write the structure like this which is in first normal form:

cons_proj(proj_code, proj_name, start_date, end_date, cons_id, cons_name, grade, daily_rate, days)

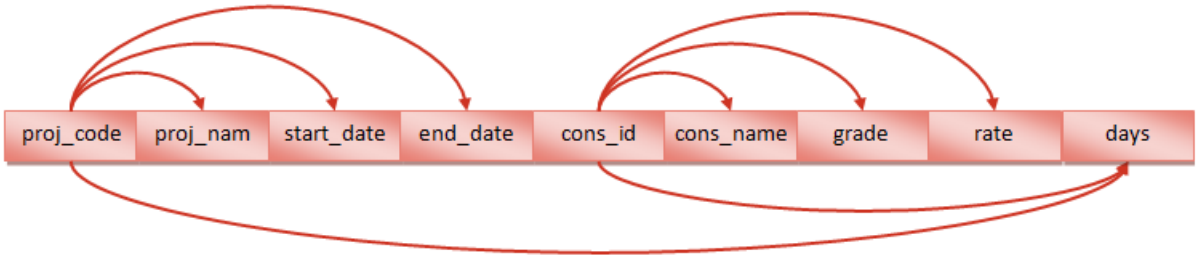
Second normal form

The next stage of structure refinement takes your relations from first normal form to second normal form (2NF). For a relation to be in 2NF it must also fulfil the criteria for 1NF. For this reason, the official definition often starts "*A relation is in 2NF if it is in 1NF and ...*". This can be a source of confusion, but it simply means that we already know the relation is in 1NF and now we are refining it further.

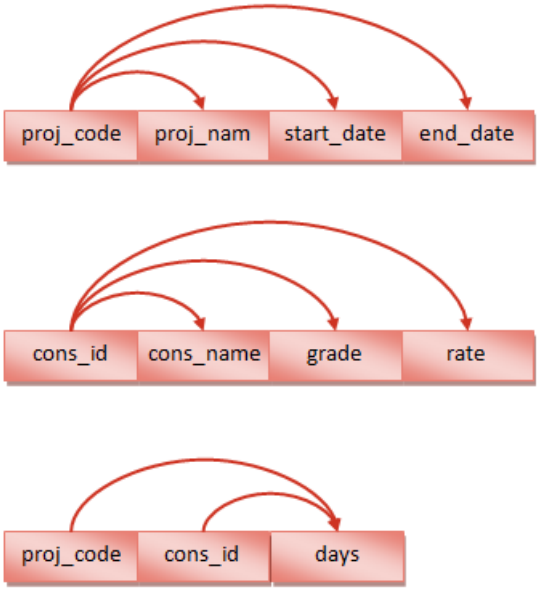
Verifying that your relations are in 2NF is concerned with *partial dependencies*. A partial dependency arises when there are attributes in the relation that are functionally dependent on part of a composite primary key but not all of it. This leads to the observation that if your 1NF relation does not have a composite primary key, then it is already in 2NF and there is nothing more to do at this stage.

In the example of consultants and projects, the primary key is composite, and we therefore need to check for partial dependencies. The dependency diagram can help with this:

cons_proj(proj_code, proj_name, start_date, end_date, cons_id, cons_name, grade, daily_rate, days)



Notice that the only attribute which is functionally dependent on both of the attributes in the primary key is *days*. All of the others are dependent on *either* proj_code *or* cons_id. The majority of dependencies in the diagram are therefore partial dependencies. Eliminating them is a question of creating a new set of relations according to the dependencies that have been identified. The result is shown below.

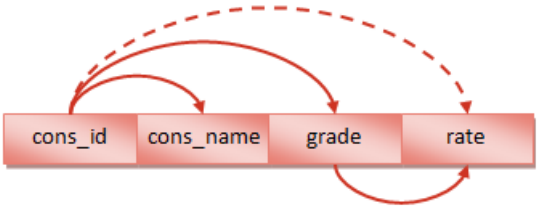


We now have three relations which satisfy the requirements of 1NF, and in addition have no partial dependencies. This structure, which could be written as shown below, is in 2NF.

consultant(cons_id, cons_name, grade, daily_rate)
project(proj_code, proj_name, start_date, end_date)
cons_proj(*proj_code*, *cons_id*, days)

Third normal form

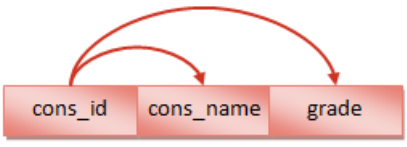
Whereas in second normal form we are eliminating partial dependencies, in third normal form (3NF) we are eliminating *transitive dependencies*. A transitive dependency exists if one non-key attribute is functionally dependent on another non-key attribute. In the consultant example, you will notice that there are two non-key attributes called *grade* and *rate* in the CONSULTANT table. If we assume that the consultant's grade determines their daily rate, then we have an instance of a transitive dependency:

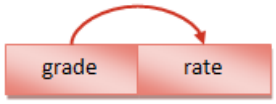


The term *transitive* indicates a mathematical concept in which if entity A is related to entity B and entity B is related in the same way to entity C, then by implication entity A is related to entity C. A real-world example is the relationship *is the sibling of*: if John is the sibling of Jack, and Jack is the sibling of Jenny, then John is the sibling of Jenny.

In the consultants example, *cons_id* determines *grade*, and *grade* determines *rate*. Therefore, *cons_id* also determines *rate* and this is the transitive dependency shown as a dashed arrow in the diagram. Because the other two dependencies exist, however, we do not need to represent the transitive dependency explicitly.

To remove the transitive dependency, extract the non-key determinant (*grade*) and its dependents (*rate*) into a new relation, and make the determinant the primary key. A 'copy' of the primary key remains in the original relation where it plays the role of foreign key:





The final set of relations for the consultants database in 3NF can be written:

consultant(cons_id, cons_name, *grade*)
grade(grade, daily_rate)
project(proj_code, proj_name, start_date, end_date)
cons_proj(*proj_code*, *cons_id*, days)