

In this module we'll learn about normalisation, which is a way of ensuring that all tables of a relational database are structured according to the relational principles.

$\pi$

## Functional Dependencies

Employee

EmployeeID	BirthDate	LastName	FirstName	Street	Postcode	Suburb
127	'2012-10-01'	Wong	Andy	11 Sackville St	3101	Kew
146	'2012-10-03'	Collins	John	3 Camberwell Rd	3124	Camberwell
164	'2012-10-05'	Smith	Peter	4 High St	3022	Ardeer
188	'2012-10-10'	Nguyen	Lan	4 High St	3181	Prahran

Duplication

Now we are  
in a pickle!

Relation schema = Definition of the structure of a table (relation)

The goal of a good relational design is to create meaningful entities and relationships between them with the ultimate aim of minimising duplication. Duplication is not so much a waste of storage space as it is a threat to consistency – when we repeat values, we may update one copy but not the other, and we may have different values where we should have the same. Looking at the example, we have twice the postcode 3022 and the Melbourne suburb of Prahran. Actually, 3022 is not the postcode of Prahran at all.

Suppose someone realises this while working on one employee file, the file of Peter Smith. They correct the suburb to Ardeer, which is the location that actually has the postcode of 3022.

Someone else works on the file of Lan Nguyen and observes the same problem; but in this case, the person decides the suburb must be correct but the postcode is not. So she corrects the postcode to 3181.

Assuming that Peter and Lan actually do live in the same suburb, we have now introduced a mistake.

The actual problem here is flawed functional dependency. We can assume that the primary key in this relation is employeeID. IDs are tags that uniquely identify an entity. If you know the staff id of a person, you know exactly which person we mean. There is no doubt as to what that person's birthday is, and there is no doubt about what the person's name might be. The employeeID uniquely defines the person, so that we can derive all this data. We say that the employeeID functionally determines the birthdate, firstname, lastname and street, even postcode. But then it gets tricky: Actually the postcode alone is enough to

functionally determine the suburb. So in a way we are doubling up by having both the postcode and the suburb in the table.

$\pi$  Fixing Functional Dependencies

Employee

EmployeeID	BirthDate	LastName	FirstName	Street	Postcode
127	'2012-10-01'	Wong	Andy	11 Sackville St	3101
146	'2012-10-03'	Collins	John	3 Camberwell Rd	3124
164	'2012-10-05'	Smith	Peter	4 High St	3022
188	'2012-10-10'	Nguyen	Lan	4 High St	3022

foreign key relationship

Suburb

Postcode	Suburb
3101	Kew
3124	Camberwell
3022	Ardeer

Mistakes are a fact of life – inconsistency isn't!

Yes, this could be wrong, but it would be wrong for everyone!

SWINBURNE  
UNIVERSITY  
TECHNOLOGY

3

Most of the time, to fix flawed functional dependencies we just split up the table. If an attribute is not genuinely dependent on the key we have chosen as a primary key in the relation, we figure out what key it is really dependent on, and make this key a primary key in a new relation. This new table is now a tidy one – the suburb is mentioned only once and clearly defined by the postcode.

The postcode also has to appear in the original table – as a foreign key. Note that we now have a one-to-many relationship. Many employees may live in the same suburb, but only in one at any given time.

$\pi$

## Transitive Dependencies

Employee

EmployeeID	BirthDate	LastName	FirstName	Street	Postcode	Suburb
127	'2012-10-01'	Wong	Andy	11 Sackville St	3101	Kew
146	'2012-10-03'	Collins	John	3 Camberwell Rd	3124	Camberwell
164	'2012-10-05'	Smith	Peter	4 High St	3181	Prahran
188	'2012-10-10'	Nguyen	Lan	4 High St	3181	Prahran

```

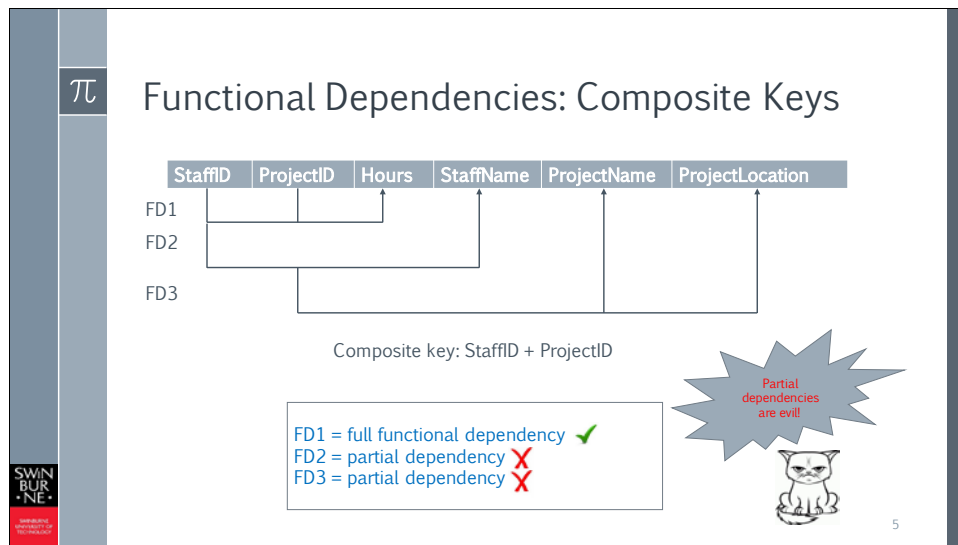
graph LR
    A[EmployeeID] --> B[BirthDate]
    A --> C[LastName]
    A --> D[FirstName]
    A --> E[Street]
    A --> F[Postcode]
    F --> G[Suburb]
  
```

SWINBURNE

4

A transitive dependency is a functional dependency that passes via some other possible key. In the case of the suburb, we can't actually say that the suburb is not at all dependent on the employeeID, because when we know the employeeID, we can tell for sure which suburb the person lives in. Just as we can tell what the person's name is. But the dependency of the suburb on the employeeID is transitive – it is actually sufficient to know the postcode to know the suburb. So the dependency of the suburb on the employeeID 'passes through' the postcode – and therefore we say it is transitive.

In the language of the relational world, we say that a transitive dependencies exists when one or more attributes depend on non-key attributes – but all attributes are still uniquely defined by the primary key.



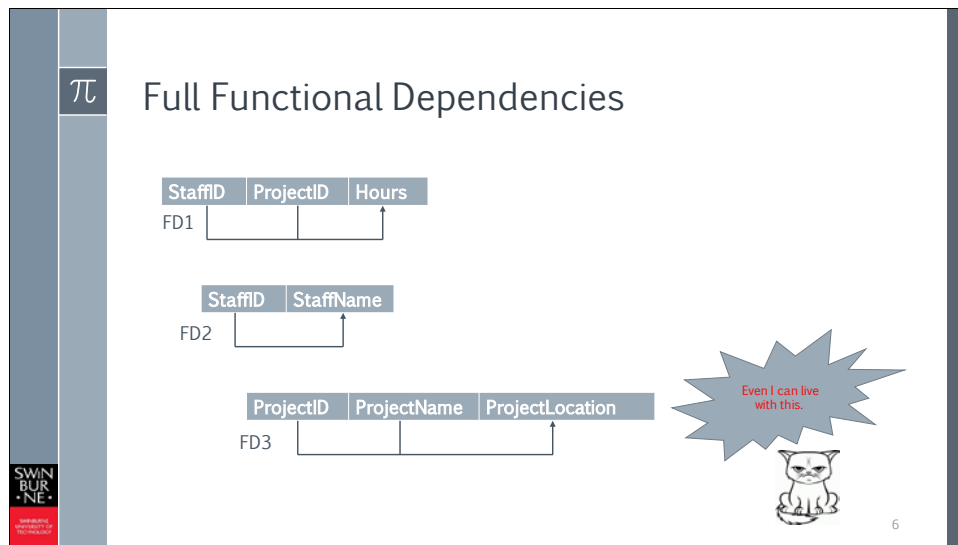
If we need more than one attribute to form a key that uniquely defines all other attributes, we call this a composite key. In the table, you can see that the Hours attribute depends on both the StaffID and the ProjectID, because we assume that the hours mean the hours a staff member has spent working on a project.

But the StaffName clearly only depends on the StaffID, and the ProjectName and the ProjectLocation clearly only depend on the ProjectID. We have to assume that the primary key is the key that defines all attributes in a relation. When we have such diverse attributes, we cannot have a single key, because every single attribute in the table only defines a part of the non-key attributes. For our functional dependency analysis we have to use the superkey – the candidate key that covers all attributes. Therefore we choose a composite of StaffID and ProjectID. Now that we are certain that all attributes are dependent on the primary key, we can check whether any attributes are only partially dependent on the superkey.

Having chosen this superkey, we realise that StaffName only needs half the primary key to be fully defined – we call this a partial dependency. Partial dependencies are bad – they are a so-called design-anomaly and indicate potential duplication! Because when a staff member starts working on a different project, we record the hours with the same StaffID and StaffName, but a different ProjectID and ProjectName (as well as location). This means StaffName is duplicated – and we run into potential consistency problems again.

Similarly, if several people work on the same project, we have duplication of ProjectName and location.

So what do we do? We do what we always do in relational modelling: make more tables!



As soon as we have identified the keys that uniquely describe some attributes, for each different key we create a new table that then holds these attributes with their keys.

This approach minimises duplication of data. You may think, how does this minimise duplication when all the key columns are repeated in every table? This is a good point, but if you think about the content of each table, the keys are unique and therefore, the attributes they define will also have unique values.

You'll find that in relational design we worry a lot about 'horizontal' redundancy – having the same data in different tuples. We don't worry about vertical redundancy – when columns repeat in different tables.

Dividing the data up into tables according to functional dependencies gives us a database that is sound according to the relational principles. Whether the design is also a good one is another story. This depends on your domain – on the actual information you are trying to capture.



$\pi$

## Normal Forms

desirable

→

First Normal Form	1NF
Second Normal Form	2NF
Third Normal Form	3NF
Boyce-Codd Normal Form	BCNF
Fourth Normal Form	4NF
Fifth Normal Form	5NF

7

Normalisation of the relation schemas of a database is the process that ensures the relations are well designed according to the relational model. It is an analysis that takes the relations through a series of tests that make sure the relations don't contain any design anomalies. The tests certify that a relation is in a certain Normal Form. When a relation is in a particular Normal Form, we know that it does not suffer from a particular anomaly. For example, second normal form eliminates partial dependencies.

The Normal Forms subsume each other, meaning that you cannot achieve second normal form without having achieved first normal form. It is a condition of second normal form that the table is in first normal form AND devoid of any partial dependencies. It is also a condition of the test for third normal form that your table is already in second normal form.

Although we have six normal forms, practitioners in industry never bother with anything beyond third normal form. In fact, practitioners often deliberately denormalise for speed. But this is for another week.

What we should not have is a database that is not in third normal form just because the designer doesn't know how to implement third normal form.

$\pi$  First Normal Form

› Repeating groups:

not in 1NF (has repeating groups)

propNo	propAddress	inspDate	inspTime	comments	staffNo	sName
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	Need to replace crockery	SG37	Ann Beech
		22/04/16	09:00	In good order	SG14	David Ford
		1/10/16	12:30	Damp rot in bathroom	SG14	David Ford
PG16	5 Novar Dr, Glasgow	22/04/16	13:00	Replace living room carpet	SG14	David Ford
		24/10/15	14:00	Good condition	SG37	Ann Beech

Plain awful.

Example from Connolly & Begg text

8

A relation is in first normal form if it has no repeating groups. So to ensure our tables are in first normal form, we have to test for repeating groups.

This relation describes viewings of rental properties: The responsible staff member at the real estate agent's visits a property and notes down comments about things that the renter thinks need fixing. We can see that the properties are being inspected about every 6 months. The staff member then takes down possible flaws that need addressing. The same property can be inspected by different staff members.

The longer a property is managed by this company, the more inspection entries it has for each property. You can see the predicament with the table structure: Many of the attributes are multivalued. The entries in these multivalued attributes are all in the same order – so we need to find the second entry in inspDate to match the second comment if we want to know when a particular flaw was observed. This is everything but practical – and therefore a design anomaly.

When we remove this anomaly, we have 1NF. How do we do this?

$\pi$

# First Normal Form

▷ Repeating groups removed:

in 1NF (has no repeating groups)

propNo	propAddress	inspDate	inspTime	comments	staffNo	sName
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	Need to replace crockery	SG37	Ann Beech
PG4	6 Lawrence St, Glasgow	22/06/16	09:00	In good order	SG14	David Ford
PG4	6 Lawrence St, Glasgow	1/10/16	12:30	Damp rot in bathroom	SG14	David Ford
PG16	5 Novar Dr, Glasgow	22/04/16	13:00	Replace living room carpet	SG14	David Ford
PG16	5 Novar Dr, Glasgow	24/10/15	14:00	Good condition	SG37	Ann Beech

Example from Connolly & Begg text

We simply give each of the inspections a row by itself. Each inspection now has a tuple to itself. One drawback we observe is that now the property number and address attributes have repeated values. We know that this is not good, and we have to fix it, but this is nothing to do with first normal form.

$\pi$  First Normal Form

› Repeating groups, variation

not in 1NF (has repeating groups)

propNo	propAddress	inspDate1	inspTime1	comments1	staffNo1	sName1
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	Need to replace crockery	SG37	Ann Beech

inspDate2	inspTime2	comments2	staffNo2	sName2
22/04/16	09:00	In good order	SG14	David Ford

inspDate3	inspTime3	comments3	staffNo3	sName3
1/10/16	12:30	Damp rot in bathroom	SG14	David Ford

I give up.

SWINBURNE

10

Repeating groups come in two different variations; the first we have already seen. When there are repeating groups, some database designers try to solve the problem by repeating column groups. Most of the time, this is absolutely terrible design. Why?

- Well, most of the time we can't be sure exactly how many groups we'll have. In the case of property inspections, we can be pretty sure the number will vary. This means we'll have a large number of columns but many with null values.
- The other problem is finding a particular visit: If we want to find a particular inspection by date, we have to look at three attributes to find it: inspDate1, inspDate2 and inspDate3. This is terribly impractical.

It would take a very inexperienced designer to structure a table like this one. But..

$\pi$

## First Normal Form

› Repeating groups, simpler:

not in 1NF (has repeating groups)

propNo	propAddress	inspDate	inspTime	comment_1	comment_2	comment_3
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	Need to replace crockery	Loose tiles above bath	Replace oven
PG16	5 Novar Dr, Glasgow	22/06/16	09:00	In good order	null	null
PG23	12 Glen St, Glasgow	1/10/16	12:30	Damp rot in bathroom	Living room carpet needs clean	null

11

People are quite likely to do something like this. When we have an inspection, it is likely a staff member will find several flaws. It is good practice not to put them all into the same field. But repeating the column isn't a good option either, for the same reasons as discussed with the last example: We don't want a restriction on the number of comments we can make, and we don't want lots of empty fields in the database.

How do we solve this?

Again, to achieve first normal form, we have to give each entry its own tuple. Can you guess what the result looks like?

$\pi$

## First Normal Form

› Repeating groups removed:

in 1NF (has no repeating groups)

propNo	propAddress	inspDate	inspTime	commentNo	comment
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	1	Replace oven
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	2	Loose tiles above bath
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	3	Damp rot in bathroom
PG16	5 Novar Dr, Glasgow	22/06/16	09:00	1	In good order
PG23	12 Glen St, Glasgow	1/10/16	12:30	1	Damp rot in bathroom
PG23	12 Glen St, Glasgow	1/10/16	12:30	2	Living room carpet needs clean

12

In this table, we represent each comment in its own row. If we think the numbering of the comment is important, we can add another attribute commentNo to store these.

Naturally, many of the other columns now have repeated values. But we don't worry about this while we are investigating first normal form. We leave this for second normal form.

$\pi$

## Second Normal Form

› Partial dependencies

not in 2NF (has partial dependencies)

in 1NF (has no repeating groups)

propNo	propAddress	inspDate	inspTime	comments	staffNo	sName
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	Need to replace crockery	SG37	Ann Beech
PG4	6 Lawrence St, Glasgow	22/06/16	09:00	In good order	SG14	David Ford
PG4	6 Lawrence St, Glasgow	1/10/16	12:30	Damp rot in bathroom	SG14	David Ford

FD1 ↑

FD2 | | ↑ ↑ ↑ ↑

SWINBURNE

13

Second normal form eliminates partial dependencies. First we have to check whether the relation is in first normal form. In this case it is, because it is the outcome of our previous normalisation step.

Now that we are convinced that the relation is in first normal form, we can begin to analyse candidate keys and functional dependencies.

To make things easier, we have left out some of the rows for space. Examining the table for candidate keys, we observe that propNo might be a good one – propAddress is determined by it.

inspDate does not depend on anything, really, but it defines the inspection in combination with the propNo. inspTime is defined by propNo and inspDate, because we can assume that a property will not be inspected twice on the same day, so we don't make it a key. If there could be several inspections a day, we would have to use the time as part of a key that defines the inspection of a property.

comments is inspection-specific, and we have already observed that propNo and inspDate can uniquely define an inspection. So like the inspection time, comments is defined by propNo and inspDate.

The staff member who conducted the inspection is also inspection-specific – if we know which inspection we are talking about, we can also tell what staff member conducted it.

So our superkey in this table is propNo and inspDate.

But as we have already established, propAddress is only dependent on the propNo, so we have a partial dependency we need to resolve before this table is in second normal form.



$\pi$  Second Normal Form

> Full functional dependencies

in 2NF (has no partial dependencies)

in 2NF (has no partial dependencies)

propNo	propAddress
PG4	6 Lawrence St, Glasgow
PG16	5 Novar Dr, Glasgow

FD1  $\xrightarrow{\text{propNo}}$   $\text{propAddress}$

propNo	inspDate	inspTime	comments	staffNo	sName
PG4	18/10/15	10:00	Need to replace crockery	SG37	Ann Beech
PG4	22/06/16	09:00	In good order	SG14	David Ford
PG4	1/10/16	12:30	Damp rot in bathroom	SG14	David Ford
PG16	22/04/16	13:00	Replace living room carpet	SG14	David Ford
PG16	24/10/15	14:00	Good condition	SG37	Ann Beech

FD2  $\xrightarrow{\text{propNo, inspDate}}$   $\text{inspTime, comments, staffNo, sName}$

14

If we remove the propAddress from the table, all other attributes are uniquely defined by the composite key of propNo and inspDate. This is our inspections table. Putting propNo and address into a separate table gives us a property table. The propNo in the inspection table can now serve as a foreign key to the properties table. You can see that the advantage is that the property address has no repeats any more. These tables are in second normal form.

$\pi$

## Third Normal Form

› Transitive dependencies

propNo	inspDate	inspTime	comments	staffNo	sName
PG4	18/10/15	10:00	Need to replace crockery	SG37	Ann Beech
PG4	22/06/16	09:00	In good order	SG14	David Ford
PG4	1/10/16	12:30	Damp rot in bathroom	SG14	David Ford
PG16	22/04/16	13:00	Replace living room carpet	SG14	David Ford
PG16	24/10/15	14:00	Good condition	SG37	Ann Beech

in 2NF (has no partial dependencies)

not in 3NF (has transitive dependencies)

15

To achieve third normal form, we first have to check whether the table is in second normal form. The inspections table is the outcome of our investigation of second normal form, so we already know it is in second normal form. (The properties table is automatically in third normal form, so it is not shown here.)

To bring a table into third normal form, all we have to do is check for transitive dependencies and eliminate them if found. As we remember, transitive dependencies are situations where an attribute is dependent on the primary key, but it does not really need the primary key to be uniquely defined. We can see that staffNo depends on the primary key – if we know which inspection we mean, we can also tell the staff member who conducted the inspection. But the staff name is already defined by the staff no we don't need the propNo and inspDate to tell the staff name. So although the staff number and name are uniquely defined by the primary key, the staff name can be determined by the staff number alone. So we have a transitive dependency.

$\pi$  Third Normal Form

› Transitive dependencies resolved

in 3NF (has no transitive dependencies)

propNo	inspDate	inspTime	comments	staffNo
PG4	18/10/15	10:00	Need to replace crockery	SG37
PG4	22/06/16	09:00	In good order	SG14
PG4	1/10/16	12:30	Damp rot in bathroom	SG14
PG16	22/04/16	13:00	Replace living room carpet	SG14
PG16	24/10/15	14:00	Good condition	SG37

in 3NF (has no transitive dependencies)

staffNo	sName
SG37	Ann Beech
SG14	David Ford

SWINBURNE  
UNIVERSITY  
TECHNOLOGY

16

So if we remove the staff name and put it into a separate table, we have eliminated the transitive dependency. You can see that there is less redundancy – only the staffNo has duplicate values. This cannot be avoided if we want to preserve the link to the staff table, so that we can tell who conducted an inspection. The staffNo serves as a foreign key to the staff table, so that we can make the connection between inspection and staff name.

Both tables are now in third normal form – we have established previously that they are in second normal form, and now we have removed the transitive dependency by making a second table for staff, so we have third normal form.

$\pi$

## The other example...

› Can we achieve third normal form?

in 1NF (has no repeating groups)

propNo	propAddress	inspDate	inspTime	commentNo	comment
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	1	Replace oven
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	2	Loose tiles above bath
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	3	Damp rot in bathroom
PG16	5 Novar Dr, Glasgow	22/06/16	09:00	1	In good order
PG23	12 Glen St, Glasgow	1/10/16	12:30	1	Damp rot in bathroom
PG23	12 Glen St, Glasgow	1/10/16	12:30	2	Living room carpet needs clean

SWINBURNE  
UNIVERSITY OF  
TECHNOLOGY

17

When we looked at repeating groups with 1NF, we had this example where multiple comments were allowed with each inspection. So we know from memory that this table is in 1NF.

But is it in 2NF?

Pause the recording and think about the solution before you continue.

$\pi$

## Solution: Second Normal Form

› Analysing for partial dependencies

in 1NF (has no repeating groups)

propNo	propAddress	inspDate	inspTime	commentNo	comment
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	1	Replace oven
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	2	Loose tiles above bath
PG4	6 Lawrence St, Glasgow	18/10/15	10:00	3	Damp rot in bathroom
PG16	5 Novar Dr, Glasgow	22/06/16	09:00	1	In good order

FD1 
↑

FD2 
↑

FD3 
↑

18

As you are becoming familiar with the rules, and the fact that they always aim to remove redundancy, you probably already suspected from the repetition in some columns that this table isn't normalised to our satisfaction. But how do you check for 2NF?

Right, you have to analyse which attribute depends on which key. This is probably easy by now – you can tell which combination of keys lets you derive any other attribute without any doubt.

The superkey is the key that determines all other attributes. This is the case with FD3 – propNo, inspDate and commentNo are the only composite key that describes all other attributes.

But we can see that not all attributes are dependent on the whole superkey. propAddress is determined by propNo alone, and inspection time only needs propNo and inspDate as a key. How do we eliminate these partial dependencies? Think about it before you continue.

$\pi$


## Solution: Second (and Third) Normal Form

› Creating separate tables

propNo	propAddress
PG4	6 Lawrence St, Glasgow
PG16	5 Novar Dr, Glasgow

propNo	inspDate	inspTime
PG4	18/10/15	10:00
PG16	22/06/16	09:00

propNo	inspDate	commentNo	comment
PG4	18/10/15	1	Replace oven
PG4	18/10/15	2	Loose tiles above bath
PG4	18/10/15	3	Damp rot in bathroom
PG16	22/06/16	1	In good order



The solution is to subdivide the data into three tables so that all attributes are only dependent on the one primary key of the table. You may not be too happy with this result; we were meant to reduce redundancy, and now we have lots of same columns in different tables! But you'll remember – in the relational model we don't mind additional columns, it's duplicate values in rows that we don't like. These tables are all in second normal form. Since we do not have any transitive dependencies, they are also automatically in third normal form.

Many practitioners will prefer the combined table to these three individual tables. This is a valid decision – it's called denormalisation – and it is often found in real applications. As long as you understand the implications – having to take care of duplicate values and their consistency – you can denormalise.

$\pi$

## Surrogate Keys

› Replacing Composite Keys

insplD	propNo	inspDate	inspTime
1	PG4	18/10/15	10:00
2	PG16	22/06/16	09:00

insplD	propNo	inspDate	commentNo	comment
1	PG4	18/10/15	1	Replace oven
1	PG4	18/10/15	2	Loose tiles above bath
1	PG4	18/10/15	3	Damp rot in bathroom
2	PG16	22/06/16	1	In good order

20

A related topic is the notion of surrogate keys. insplD can be an additional column with a unique number. The advantage is that rather than having several columns in the child table as foreign keys, we only have one foreign key column. Note that in the parent table, where the surrogate key is the primary key, we cannot remove the attributes that form the composite key, because of their information value.

In many relations, there is an obvious unique id field that can be used as the primary key. Staff id, property number, order number, invoice number are all examples of id fields that are well suited as unique identifiers. When we have such fields, it is recommended that we use them as primary keys, because they have a real meaning – the PG coding for the property number here no doubt follows some naming convention that actually tells the real estate people something.




But if there is no obvious id field – let's make one. Any decent relational database product has a tool called an autonumber or autoincrement or similar. If you define a field as such a number, the DBMS will take care it assigns a unique number to this field automatically – you don't have to do anything when you add a new tuple to a table, the key gets generated for you. In practical situations, people use these surrogate keys extensively.

In particular, people like to replace composite keys with surrogate keys. Composite keys are trouble – when you need to find matching data from two tables, the DBMS has to match the tuples on several keys rather than just one. That's lots of extra processing work.

Having said that, you still need to know about composite keys. Composite keys help you with data modelling, as you have seen during normalisation – functional dependencies are crucial to ER modelling. Surrogate keys cannot tell you anything about the dependencies, because they do not mean anything by themselves. They are surrogates – convenient replacements for complex combinations of keys.



## Summary



- › Normalisation is a way of ensuring that each relation schema within the database schema is functional and redundancy-free.
- › If a relation is in 1NF, it has no repeating groups.
- › If a relation is in 2NF, it is in 1NF and has no partial dependencies.
- › If a relation is in 3NF, it is in 2NF and has no transitive dependencies.
- › The normalising process works by separating entities into tables. Sometimes this is undesirable and people decide to denormalise.
- › Surrogate keys are widely used when no obvious attributes lend themselves as keys
  - especially with composite keys.

21

Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.