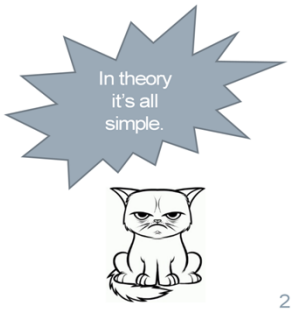
Relational Databases

Entity-Relationship Modelling

Entity-Relational modelling is the process that lets you create a relational database as your data storage.

Entity Relationship Design

- Steps to build a conceptual design
  1. Identify the entity types
  2. Identify and associate attributes with the entity types
  3. Identify the relationship types
  4. Determine cardinality and participation constraints
  5. Determine primary and foreign keys
  6. Validate the model



2

In the relational model, every concept and every process is well defined. This is the benefit of a technology having been around for a long time.
When you are in need of a data store, you are often in the process of creating an application of some kind, and the data store has to accommodate the data of your application domain.
Generally, you have a fair idea what kind of data you need to store. In the typical case of an application selling products of some kind, we usually have product data, customer data, order data and so on.

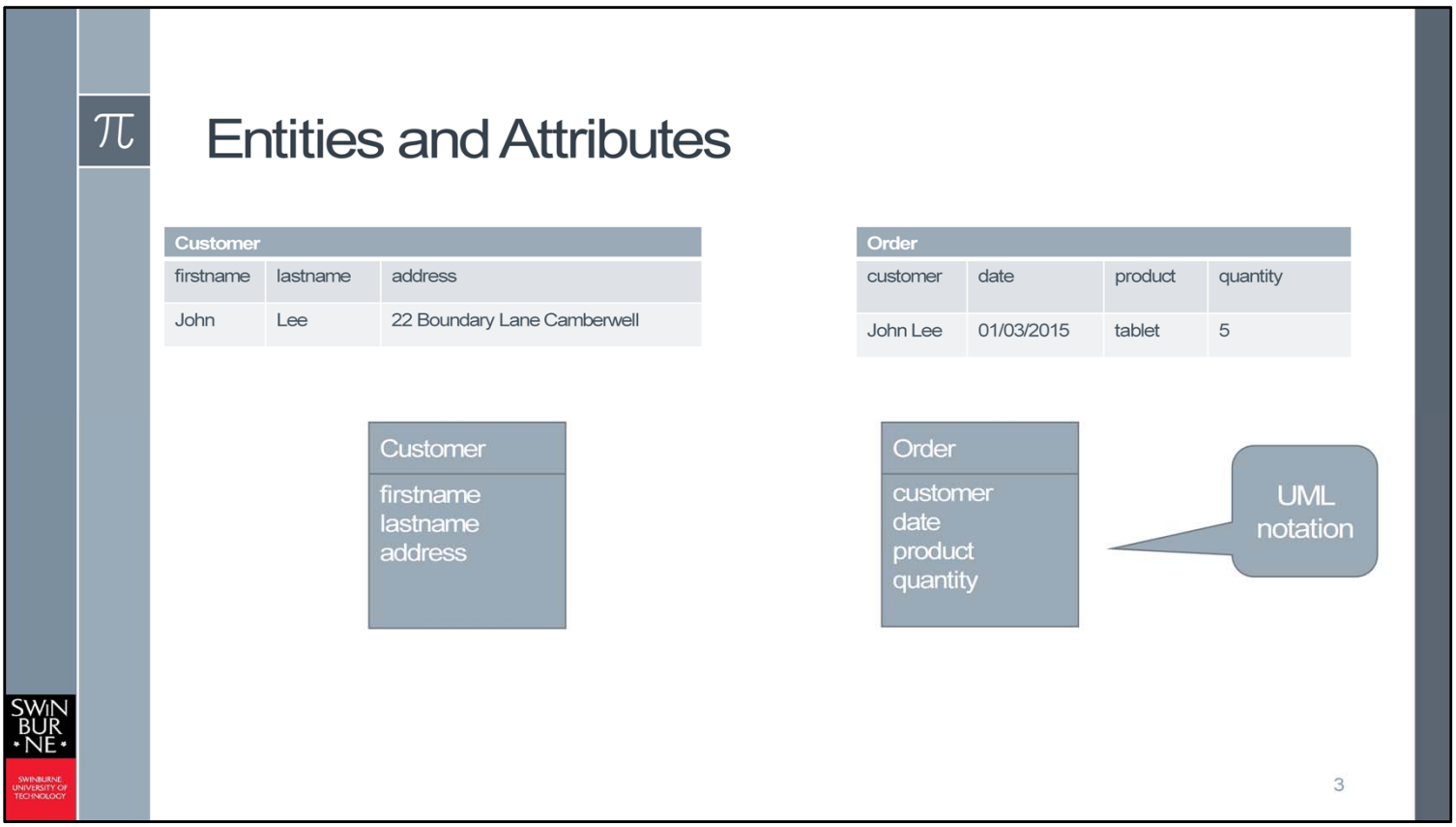
So when we start with the first step, the majority of our entities are easy to detect – customer, product, order, invoice and such will easily spring to mind.
When we get to the attributes, all we have to do is imagine what pieces of information our application needs to keep track of. Most of the time, we can even tell what entities these pieces of information are connected with – orders have order dates, discounts and totals. If we remember pieces of information that are not related to any existing entity, we need to think about making new entities.
We can also define the some relationships easily. We can tell that there must be a relationship between the customer and the order, because the customer places the order, and there must be a relationship between orders and products, because people usually order products from us.
The cardinality and participation constraints relate to the relationships – we can have one-to-one or one-to-many relationships. Thinking about customer and order, one customer can have many orders, but an order can only have one customer. So we have a one-to-many relationship.

The participation constraint is about whether an entity can live without a link to another entity. Can we have customers without orders? The answer will be yes for most applications, because customers can register without making their first orders. But can we have orders without customers? In most cases, the answer will be no.
Primary keys are identifiers of entities. Once we are happy with the entities we have created, we will look for an attribute that uniquely identifies the entity. Fields like order number will be likely candidates.

Foreign keys are links that implement the relationships between entities. If we want to link to entities of different tables, we have to include that table's primary key in our table as a foreign key.

The step of validating the model is best achieved by working through the normal forms of relational design. These are explained in the Normalisation module.

You have to be aware, though, that this is not the end of it – data stores are used by software applications. Software applications have a life cycle, and they evolve over time. Your data model is likely to evolve along with it. The better your design in the first place, the easier it will be to extend. This is why the validation step is very important – and we dedicate an entire week to it.



Most of the time, you'll find the steps of identifying entities and their attributes relatively easy. Entities are stored in their own tables – so each different entity you find will lead to creating a new table. The attributes of the entities make columns in a table. Therefore, all entities of the same type have the same attributes. When there are differences in the attributes of the same type of entities, we have a design problem.
For designing relational schemas, UML diagrams have become popular lately. They show the entities in boxes with the entity label in the header section of the box and the attributes (possibly with types) in the lower box.

Relationships and Cardinality

Customer		
firstname	lastname	address
John	Lee	22 Boundary Lane Camberwell

Order			
customer	date	product	quantity
John Lee	01/03/2015	tablet	5

4

Since each order needs a customer, the UML diagram shows a link between the symbols for the relations. Sometimes, the nature of the relationship is marked with a verb. Here, the relationship is simply ‘has’. In practice, you’ll find many diagrams that don’t include this.

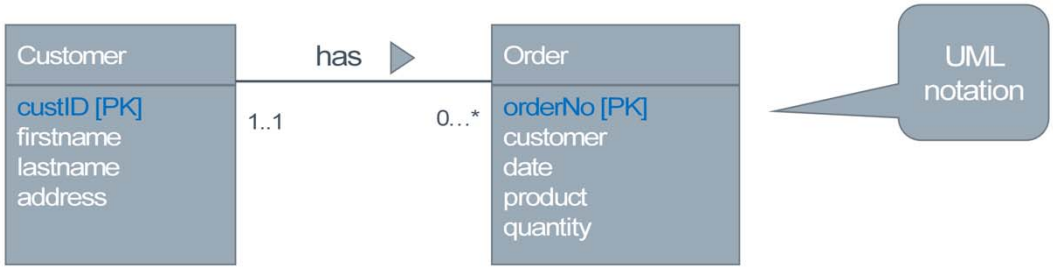
The next step is determining the cardinality or multiplicity of the relationship. The most common relationship is one to many. In this case, one customer will have zero or many orders, but an order will have precisely one customer. The UML notation shows the cardinality as 1 … 1 on the side of the customer, and as zero … * on the side of the order. The star or asterisk means an unlimited number. There are also one-to-one relationships. For example, if our staff are issued with company cars, each car would have one user.
Many-to-many relationships do not work in the relational model – if we have them, we have to solve them by using another table in between the tables that have the many-to-many relationship.

The participation constraint checks whether an entity must have a link to an existing entity. In the UML notation, this information is included in the cardinality – if there is a zero, participation is not mandatory, otherwise it is.

Primary Key

Customer			
custID	firstname	lastname	address
1234	John	Lee	22 Boundary Lane Camberwell

Order				
orderNo	customer	date	product	quantity
1111	John Lee	01/03/2015	tablet	5



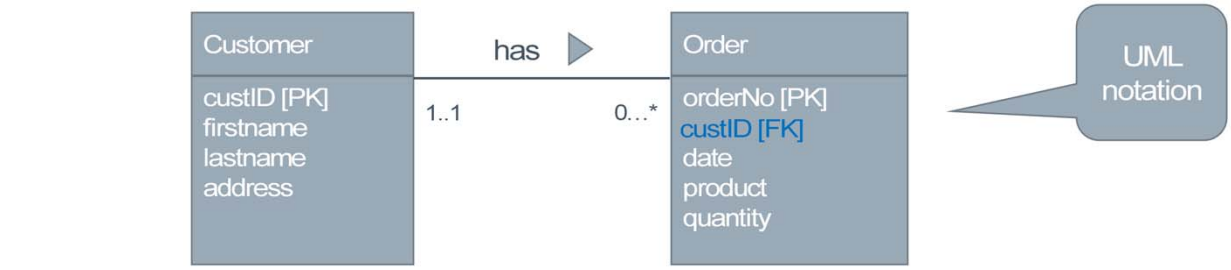
5

Primary keys are attributes that uniquely define the entity. Among all the attributes, we have to find one that will be different in each entity. Clearly, names are not very good at distinguishing between customers, many people have the same name.
Primary keys can be composite – this means that you can combine a number of attributes and make the combination of them the primary key.
If we combined the name, last name and address, we would be pretty safe from duplicate customers. But having composite primary keys is not the best option, especially when we have fields with lots of text such as address. Looking up a customer by primary key would be very time-consuming for the DBMS; it has to compare three fields to the search criterion.
Therefore most companies use ids for many entities. Order and invoice numbers help keep track of the documents, and customer ids make sure one John Smith doesn't get the invoice belonging to another John Smith. In databases, such ids are very helpful – they solve the problem of the primary key.
When we know the customer id, we can be sure what the customer's name and address is; the identity defines the values of all attributes.
OrderNo does the same for the order entities.
In the UML notation, the primary key field is identified with the PK abbreviation. Many practitioners underline the primary key fields to identify them. So if you see underlined fields in a table, these fields are part of the primary key.

Foreign Key

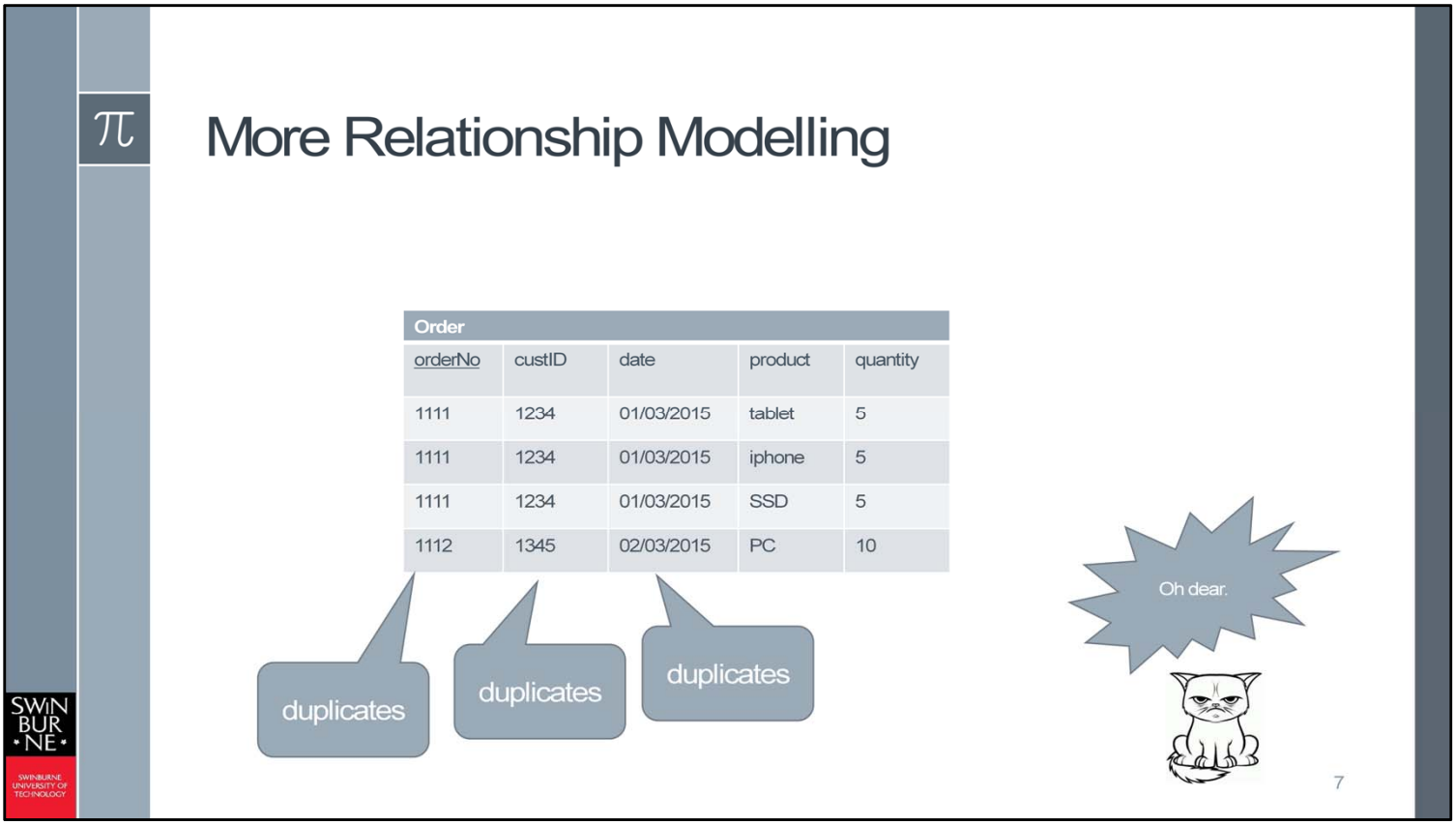
Customer			
custID	firstname	lastname	address
1234	John	Lee	22 Boundary Lane Camberwell

Order				
orderNo	custID	date	product	quantity
1111	1234	01/03/2015	tablet	5



6

Foreign keys are attributes in a table that link the table's entities to the entities of another table.
At the moment, the order table only has an attribute that tells us the customer's name. Names can be ambiguous, and the attribute would make it hard to infer the address of the customer when we want to decide where to send the delivery. We are in a pickle because all we can do is try to find the customer in the customer table based on a name. This is not good; a relational database doesn't work like google.
Instead of mentioning the customer's name, we add the primary key attribute of the customer to the order table. Because the customer ID uniquely identifies the customer, we can now make a definitive link between two entities.
Given that customers can place several orders, would the custID foreign key be unique in the order table? Naturally not. Foreign keys are only unique in one-to-one relationships. Primary keys are always unique.



In most enterprises, one order comprises a number of line items. If the customer is a corporate, one order is likely to be for many products, and the order quantity for each product will be more than one. So we have to store the product and the quantity for each line item. The line item in the order is product-specific; within the same order, we don't mention the same product twice. But across all orders, the same product will be mentioned many times.

When we look at the table, the obvious flaw is that orderNo, custID and date repeat as many times as we have line items for a particular order. In relational design, we really dislike repeat entries in rows. That's why we usually make two tables for orders and order line items.

More Relationship Modelling

› Developing a good design

Order

orderNo	custID	date
1111	1234	01/03/2015
1112	1345	02/03/2015

OrderLine

orderNo	lineNo	product	quantity
1111	1	tablet	5
1111	2	iphone	5
1111	3	SSD	5
1112	1	PC	10

duplicate (unavoidable)

8

Now that we have separated the entities into different tables, the custID and date fields no longer repeat. But the orderNo still does – is this ok?
OrderNo is a foreign key to the Order table, and because there is a one-to-many relationship between Order and OrderLine, repeating foreign keys are unavoidable. But we have avoided repeating custID and date, which are dependent on the primary key orderNo.

There is an additional column lineNo in the order line table. This is necessary if we care about the sequence in which the order lines appear on the order, for example when we print the order or show it on the screen. DBMSs give no guarantee in which order tuples are sent to the user. The sequence may be different every time.
But more importantly, we need a primary key for the OrderLine table. OrderNo won't work by itself, because we know that there will mostly be several order items in each order. What options do we have? We are looking for a field that is unique in combination with the orderNo. Could it be product? Not a bad idea, because in most companies, each product will only appear in one line on an order. So product and orderNo in combination might make a good composite primary key.
Would quantity qualify as the second part of a primary key? Clearly not, because we can already see in the table that order 1111 has a quantity of five for all its products, so this is not going to lead to a unique identifier.
Candidate keys are valid options for primary keys. In the OrderLine table, we can see we have two candidate keys – orderNo and lineNo or orderNo and product.

The case against product is that it is a text field – searching by such a field is time-consuming. So the best candidate here is orderNo and lineNo.

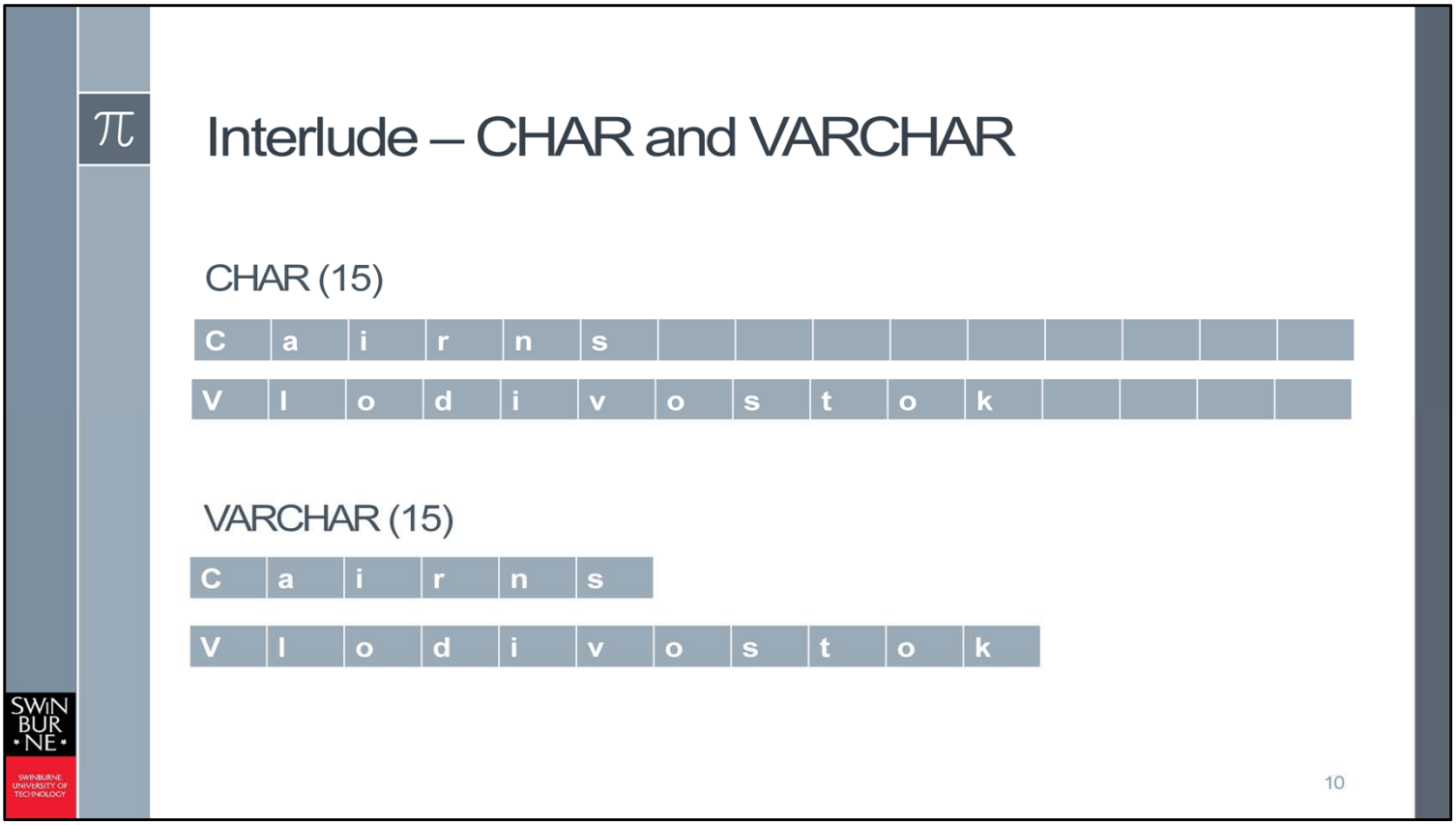
Interlude – Data Types

Type	Variations	Meaning
Integer	int, tinyint, smallint, bigint	A number without any decimals
Decimal	number, decimal, numeric, float	A number with decimals
Date	date, time, datetime, timestamp	A calendar date. Depends on locale
String	char(n), varchar(n), text, memo	Character strings
LOBs	CLOB, BLOB, image, text, memo	For large objects – character (CLOB) or bits (BLOB)
Binary	boolean, binary, varbinary	Binary fields and strings of true/false

9

These are the main groups of fields in the most common database products. Some of them have data types with the same name; int and date are available in almost all of them, but remember they all work differently depending on the database product – you have to study the data types of the DBMS you are using – no way around reading the manual.

One point worth making is that inexperienced database designers tend to think that large objects such as pictures should stay on the file system, and the database should only store the paths to such objects. This thinking is flawed – what if you change server and the directory structure is different? For example, you might migrate to a unix system. All your links to pictures would have to be updated. If you think this is ok, there is another problem. Databases need to be backed up regularly. If you put your large objects outside the database, you'll have a hard time automating their backup.

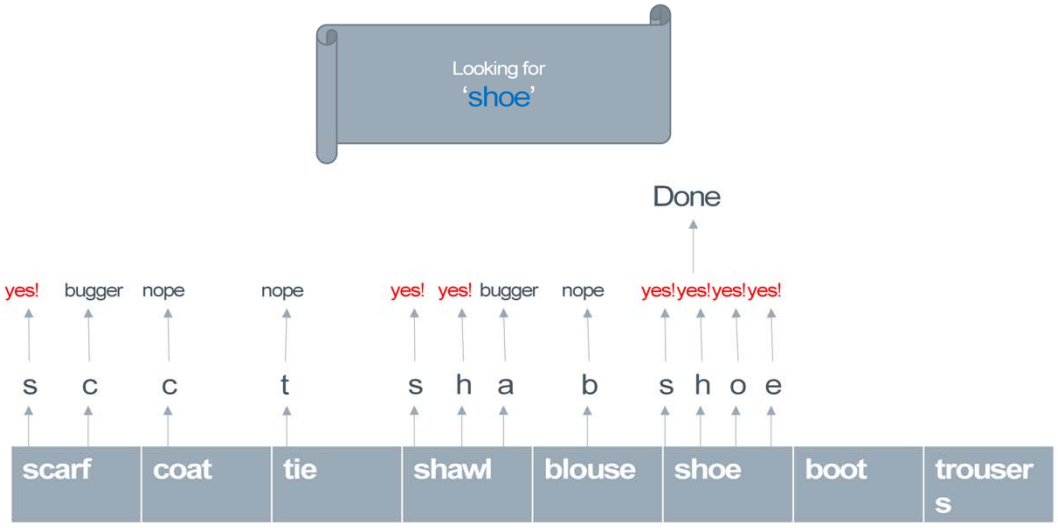


The most important character string formats in relational databases are char and varchar. They when you declare an attribute as char or varchar, you have to tell the DBMS how much space you want for your string field. This number goes in the brackets.

The important difference is that for char, this is an exact length – if you try to put a longer string into the field, the operation will fail. If you put a shorter string into the char field, it will still occupy the same space – although there is nothing in the fields after the word.
Varchar is more flexible – for varchar the number means the maximum we can have. If the string is shorter, we save the remaining space.
This is mostly a good thing, although it can be a drawback if we update a field later. But more about this in a later module.

Interlude – Text Comparisons

› Why should I avoid text attributes as primary keys?



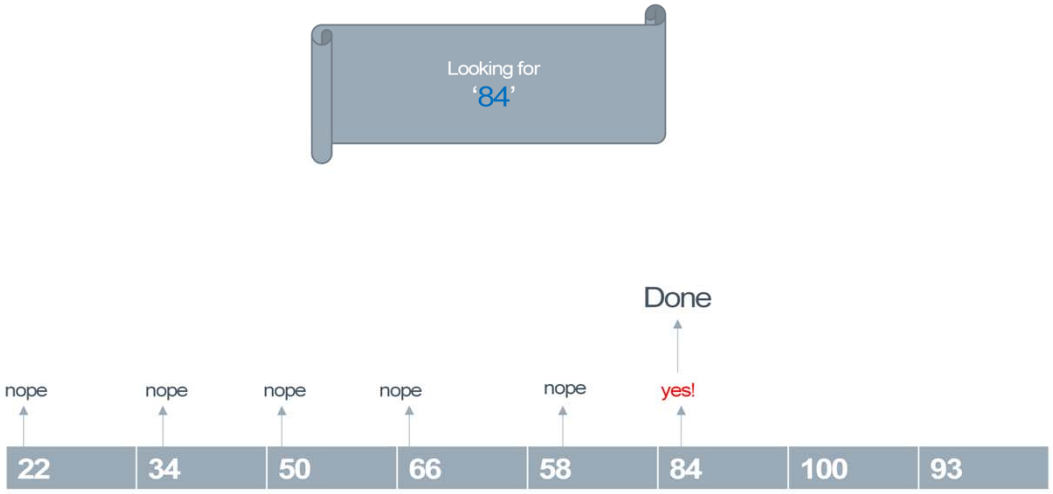
11

When you search on text attributes, this means string comparisons. Assuming you are looking for the entire content of the text field (and not just a substring inside the field), you have to compare each character to see if there is a match. Comparing a single character means transforming the character to its binary number value and subtracting it from the other character's binary value. If the result is zero, we have a match.

In strings, sometimes there is a match on the initial characters, so you keep comparing, just to find that the later characters don't match. Depending on the length of the string, this can take quite some time. Primary keys often have to be matched, because often we look for related rows based on a foreign key.

Interlude – Number Comparisons

› Why should I avoid text attributes as primary keys?



12

When you compare numbers, the entire number is compared at the same time. Numbers are encoded in a single binary value. To compare them, the processor subtracts one number from the other. If the result is zero, we have a match. This is why numbers should be preferred to text fields when looking for a primary key. Having said that, we often find string coded ID fields. This is because people like to categorise – so you can have a few letters for the category and then a number. These fields are not ideal for primary keys, but ok. As long as you don't use very long text fields, say, over 10 characters long.

Even More Relationship Modelling

› Developing a good design, continued

OrderLine

orderNo	lineNo	productID	quantity
1111	1	AM243	5
1111	2	BD133	5
1111	3	AC012	5
1112	1	CO007	10

Product

productID	name	price	supplier
AM243	tablet	249:00	Asus
BD133	iphone	199:00	Apple
AC012	SSD	139:00	Seagate
CO007	PC	411:00	IBM

13

Continuing with the modelling of the order line table, the product attribute that is part of a candidate key is actually not an ideal attribute for an order line table. The product name alone is not going to tell people very much. Don't we need to know the unit price of the product? Availability, supplier? There are many more properties products have that customers want to know about. We cannot show them to the customer if we don't have them in the database.
So naturally, our products would be in their own table, or there might be several product-related tables. The order line items then reference the product table.

Each order line item has one product it points to, but each line in the product table can have zero related order lines – if the product has never been ordered before – or it can have many related rows, if the product has been ordered many times.

Having a proper productID in the order line table means that we now have a candidate key of orderNo and productID. This makes more sense than a composite of product name and orderNo.
If we wanted to remove the lineNo attribute from the order line table (unlikely, but possible), we could use orderNo and productID as a composite key. productID still has to be a string, but it is a short one an borderline acceptable as a key column.

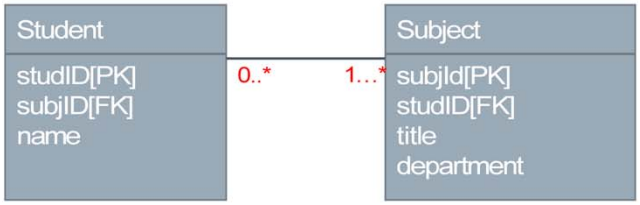
To create a primary key for orderLine, we could also add an id field for OrderLine. Creating a surrogate key is sometimes practical, but in this case we still need

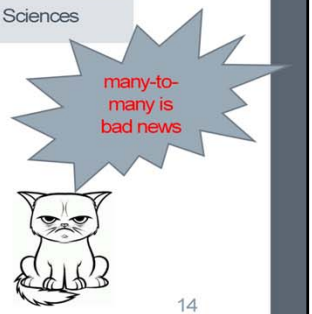
orderNo and productID as foreign keys, so using a surrogate key wouldn’t help reduce the columns. Surrogate keys are discussed later in the course.

Many-to-Many Relationships

Student		
studId	name	subjID
101	Peter	MGMT101
101	Peter	ICT402
103	Rajiv	ICT402
104	Anna	PHI787

Subject			
subjID	title	studID	department
MGMT101	Management	101	Business
ICT402	Info Tech	101	IT
ICT402	Info Tech	103	IT
PHI787	Philosophy	104	Social Sciences



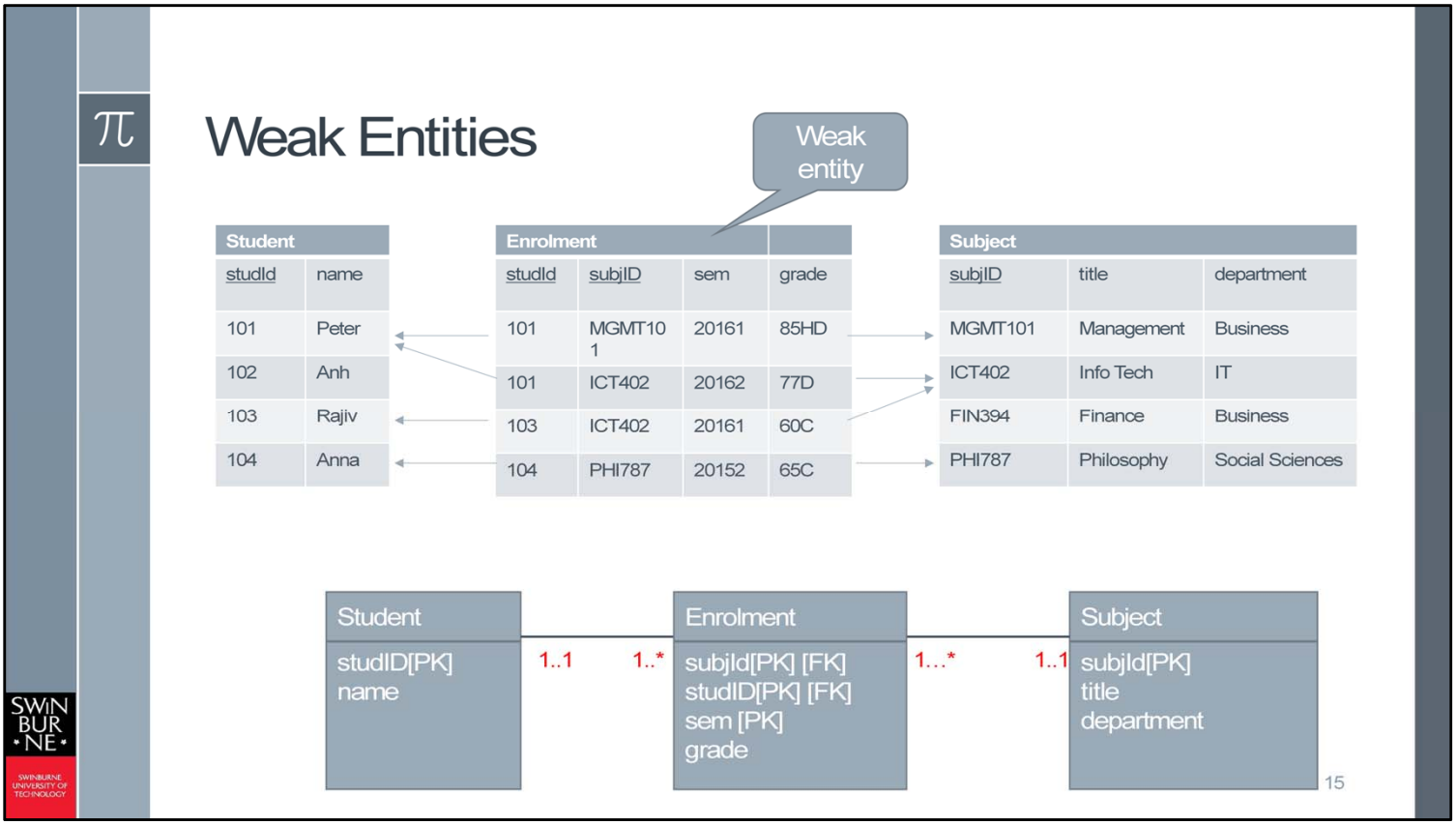


14

Thinking about the scenario of education, students may take a subject several times (because they may fail) and subjects have many student enrolments at any given time. This leaves us with a many-to-many scenario.

Many-to-many relationships just don't work in relational modelling. In effect, many-to-many means a circular relationship – each table has a foreign key that links to the other table. Because for each row, there are links to many rows in the other table, the row has to repeat to accommodate all the links. The same happens in the other table – we are duplicating the same rows just to accommodate different foreign keys.

Coping with so many combinations is not practical. Actually, it is not even logical. We have subjects that have titles and convenors, but the subject entity shouldn't have grades for each student. So the simple story is, many-to-many relationships just don't work in the relational world. What do we do about it? We create another table in between.

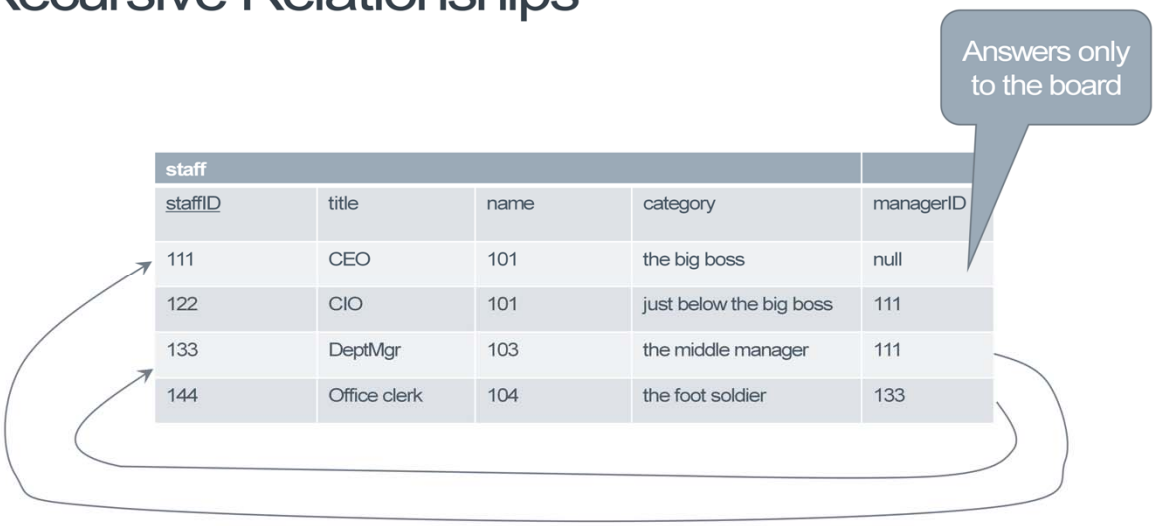


Creating a table that connects the tables student and subject is called expanding the relationship between student and subject. You can see immediately that this makes sense: now we can have columns for the semester a student has taken a subject and the grade the student has achieved.

All the entities we have looked at so far were strong entities; strong entities have their own id-type fields as primary keys and they usually describe something very tangible in the real world. The way you recognise weak entities is that their primary keys are all made up of foreign keys – in fact the entire table often contains more foreign key than other columns.

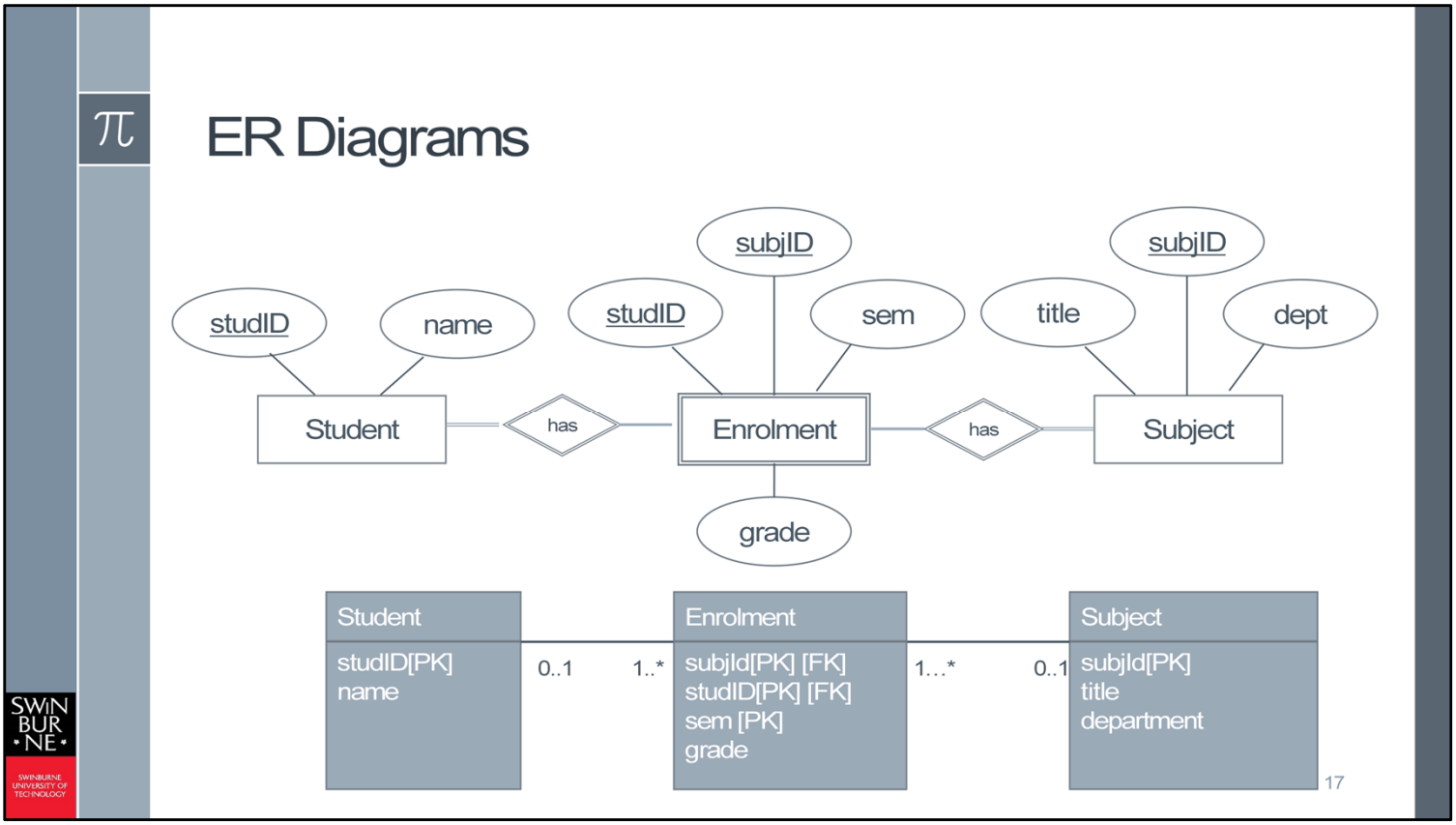
This is an important concept to remember, because IT professionals often refer to weak entities when they talk about database design.

Recursive Relationships

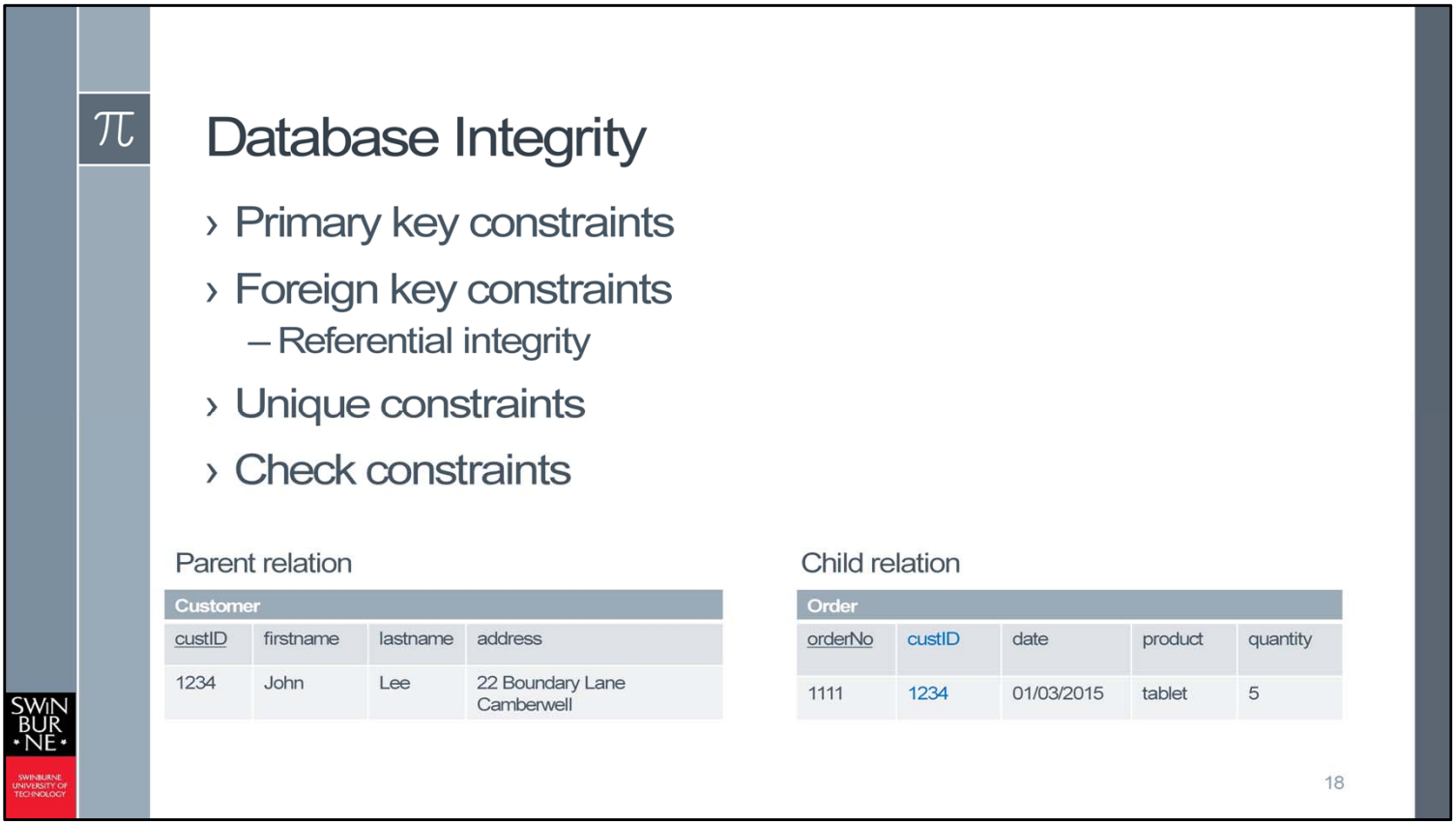


16

We can have foreign key relationships within the same table. This sounds weird, but this example demonstrates that the concept isn't far fetched at all.
When we have employees, some employees are managers who manage other employees. So we can add an attribute called managerID which references the staffID.
FIX THE NAME column



This is an ER diagram for the design shown as UML below. ER diagrams were devised at the early stages of the relational database development. They come in two flavours, this is Chen notation. There is also Crow's feet notation.
The attributes are depicted in oval shapes and connected to the entity, which is a square shape. The double line around enrolment means that it is a weak entity. The relationship between enrolment and student, the diamond shape, also has a double line because it is a relationship with a weak entity.
But the double line that connects the diamond to the student entity shows that participation is mandatory, meaning that there cannot be enrolment tuples without a link to a student. The solid line between the relationship and the enrolment means that there can be students without enrolments. The participation of enrolment is optional. The same applies between enrolment and subject – the subject's participation in the enrolment is mandatory, whereas the participation of enrolment in subject is not – we can have subjects that have never been offered, so there are no enrolments yet.
These days people are more likely to use UML notation for relational design, but if someone presents you with an ER diagram, you should look as if you knew what she was talking about..



Database integrity means a database devoid of inconsistencies. Having a good database schema – meaning a good structure with properly designed relations and relationships between them – reduces redundancy and therefore decreases chances of inconsistencies. Primary keys and foreign keys are tools that help enforce good table and relationship structures.

Database products such as Oracle, MySQL and DB2 offer mechanisms that enforce constraints. Once you have decided which of your candidate keys is the most appropriate primary key, you need to implement the table with a primary key constraint. This will enforce uniqueness – the DBMS will not allow you to enter a duplicate primary key. This stops users and client applications from breaking a good design.

Similarly, when you have identified the foreign keys that define the relationships between tables, you add a foreign key constraint in the implementation. This ensures that you cannot enter a foreign key value that doesn't exist in the parent table. The parent table is the table whose primary key is referenced by the foreign key in the child relation. So the child relation is the table that has the foreign key.
Referential integrity constraints are rules that define what should happen when a parent row is deleted, or when a parent row's primary key is updated. Both scenarios are opportunities for the database to become inconsistent – when the parent row disappears, the child row becomes orphaned. In the example, if the customer John Lee disappears from the customer table, there is an order without

a customer. This is considered a bad thing in most applications. If you implement the custID in the Order table as a foreign key, the default referential integrity rule that most databases use when you don’t specify anything else will stop the delete from going ahead.

A unique constraint can be imposed on a column or a combination of columns. This makes sure the values in the attributes are unique, even though they are not the primary key.
Check constraints are constraints that ensure the values of an attribute have a certain value. For example, you can specify that a birthdate has to be in a certain range, so we don’t enter nonsensical dates for our employees.

Summary

- Relational databases are designed using ER Modelling.
- Careful ER modelling ensures that your database is safe from unnecessary threats to its integrity.
- Careful ER modelling also ensures that your model is extensible when your application demands it.
- A good relation should have no duplication of data in the tuples (except for foreign keys).
- A good relationship is either one-to-one or one-to-many.
- Use weak entities to expand many-to-many relationships.
- Implement the constraints in the database.

19

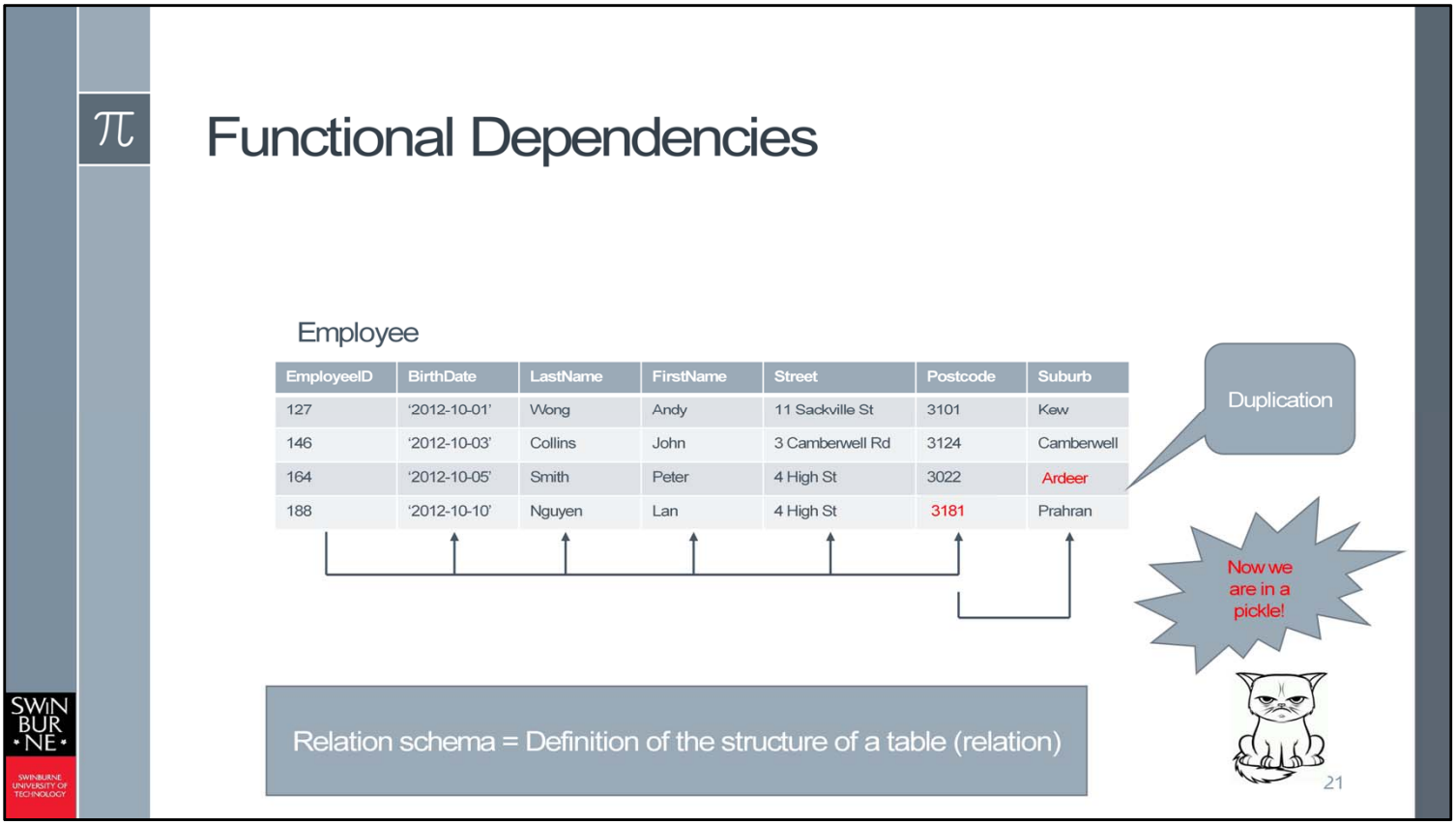
Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.

Normalisation

Relational Design Principles

SWIN BUR NE
SWINBURNE UNIVERSITY OF TECHNOLOGY

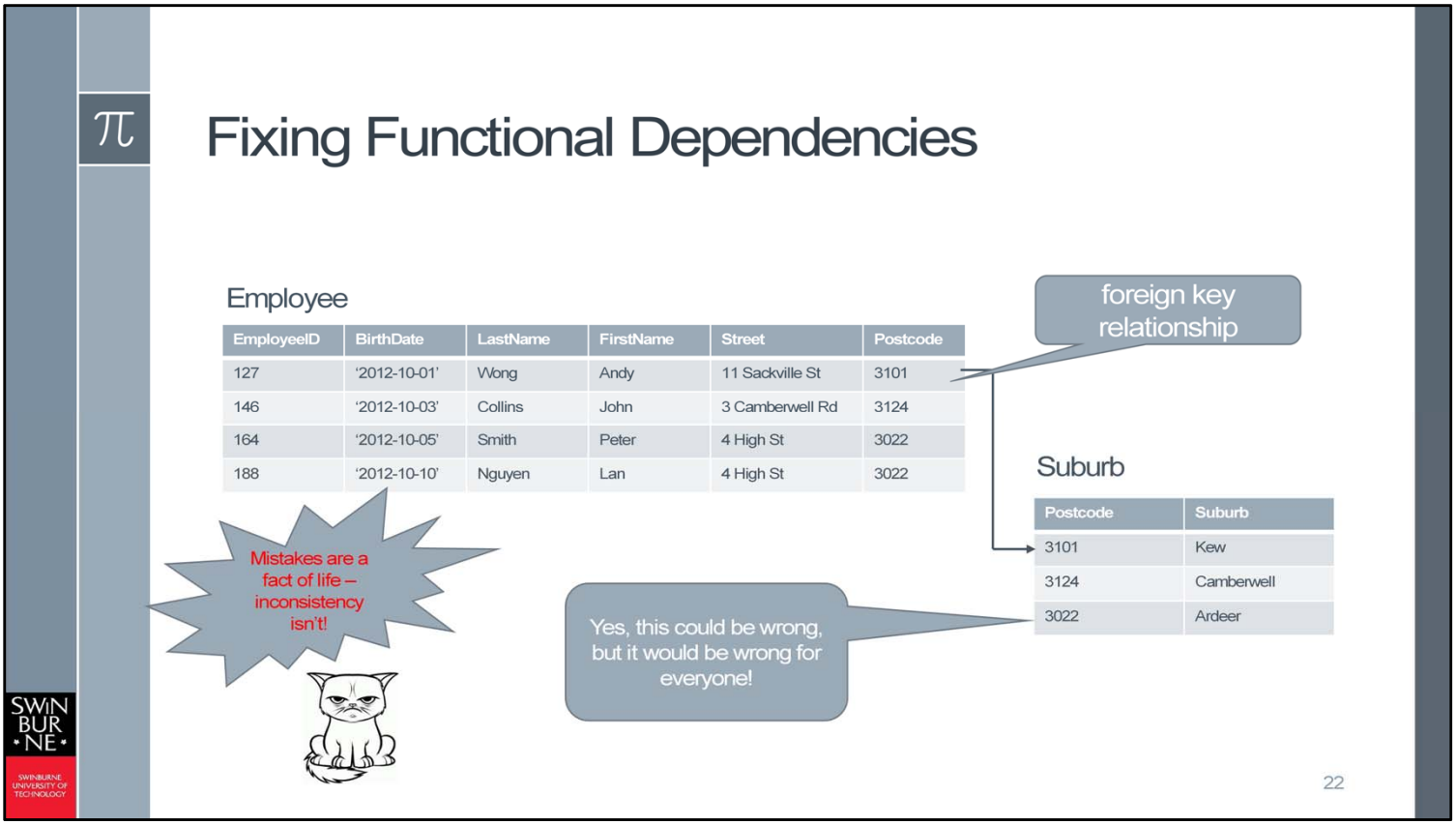
In this module we'll learn about normalisation, which is a way of ensuring that all tables of a relational database are structured according to the relational principles.



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.

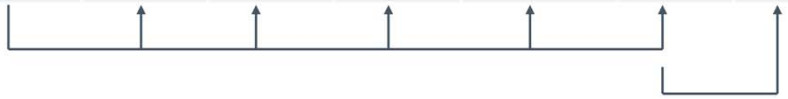


Most of the time, to fix flawed functional dependencies we just split up the table. If an attribute is not genuinely dependent on the key we have chosen as a primary key in the relation, we figure out what key it is really dependent on, and make this key a primary key in a new relation. This new table 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.

Transitive Dependencies

Employee

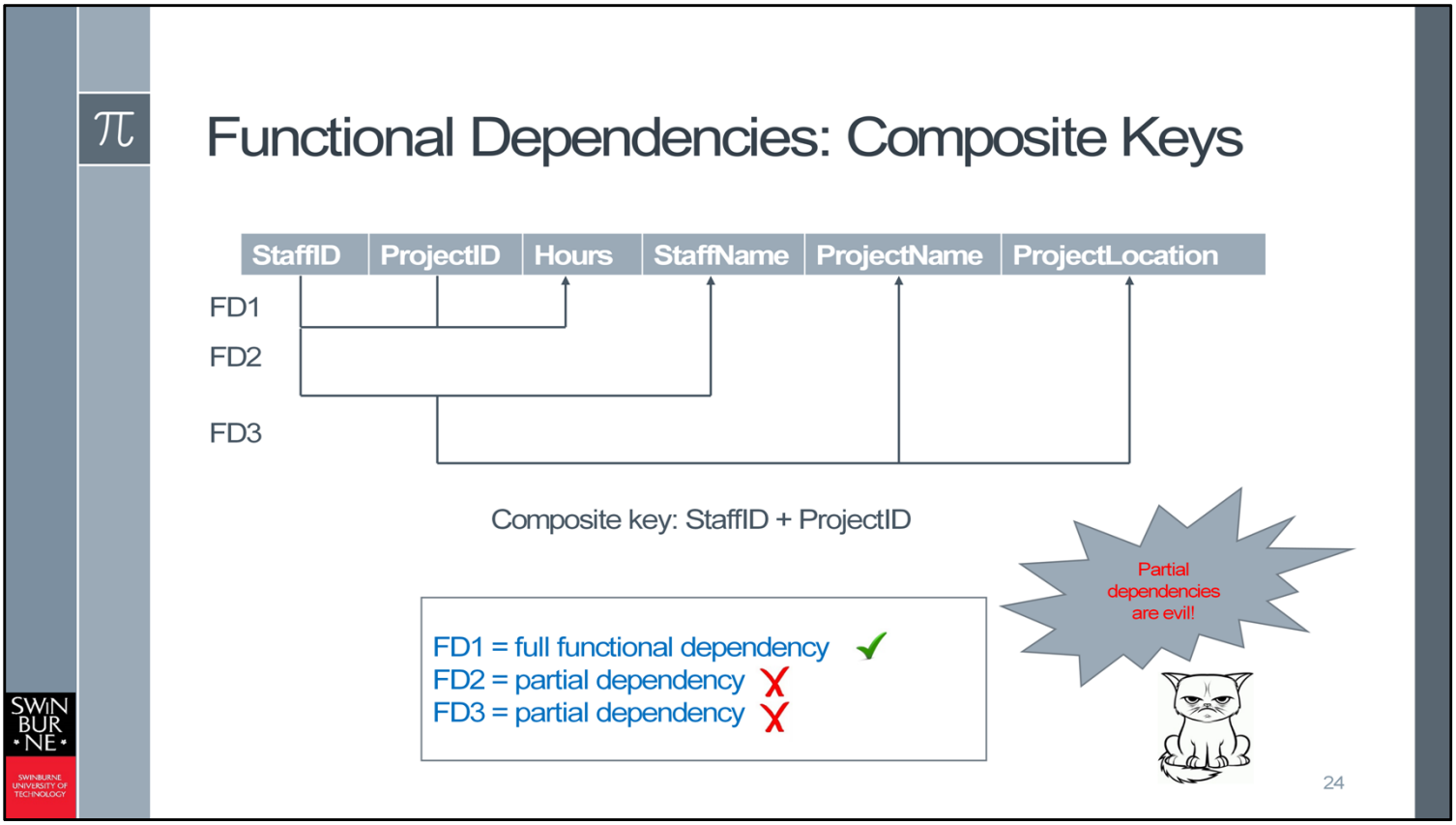
EmployeeID	BirthDate	LastName	FirstName	Street	Postcode	Suburb
127	'2012-10-01'	Wong	Andy	11 Sackville St	3101	Kew
146	'2012-10-03'	Collins	John	3 Camberwell Rd	3124	Camberwell
164	'2012-10-05'	Smith	Peter	4 High St	3181	Prahran
188	'2012-10-10'	Nguyen	Lan	4 High St	3181	Prahran



SWINBURNE UNIVERSITY OF TECHNOLOGY

23

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.



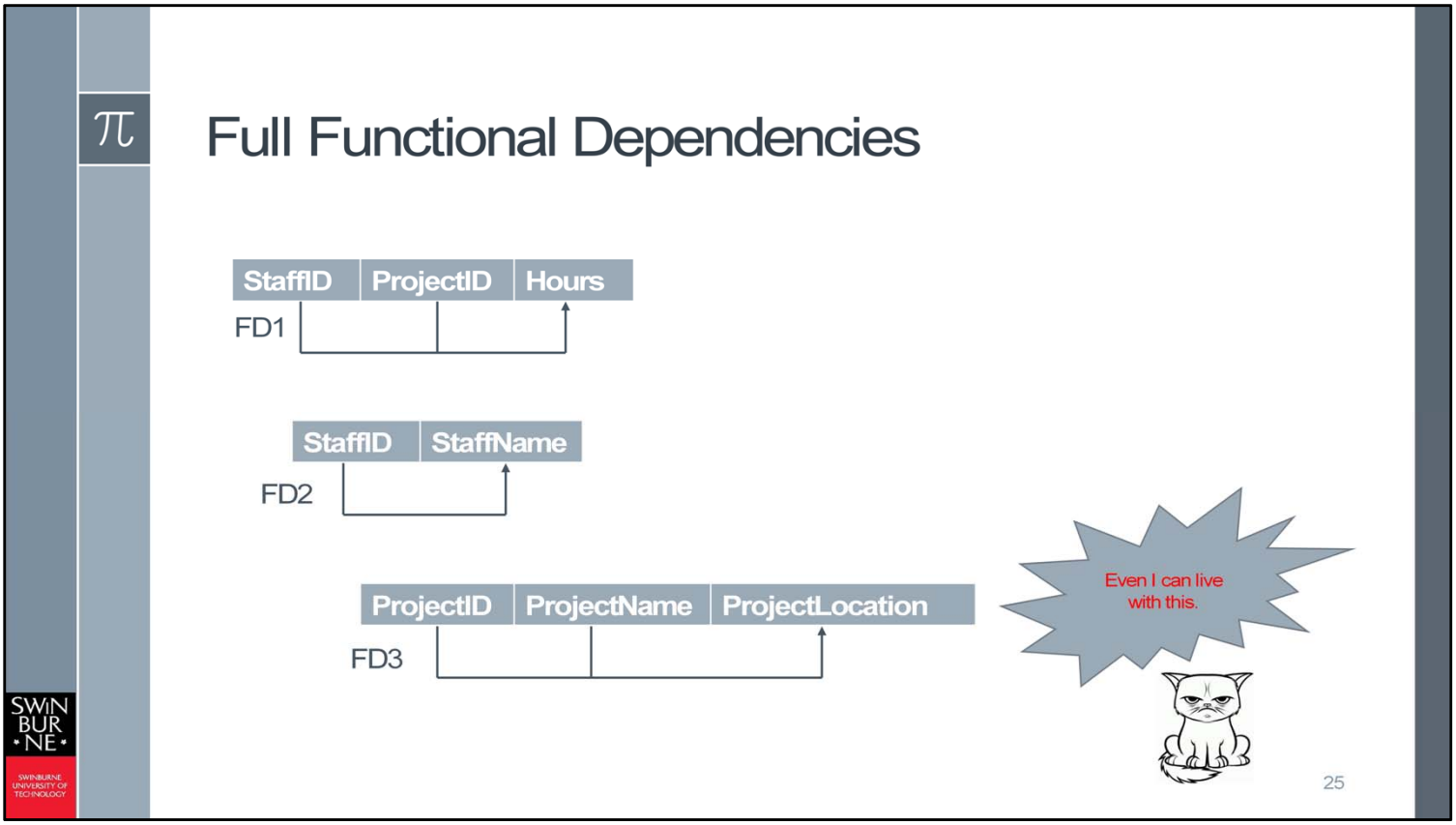
If we need more than one attribute to form a key that uniquely defines all other attributes, we call this a composite key. In the table, you can see that the Hours attribute depends on both the StaffID and the ProjectID, because we assume that the hours mean the hours a staff member has spent working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

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

26

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.

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 22/04/16 1/10/16	10:00 09:00 12:30	Need to replace crockery In good order Damp rot in bathroom	SG37 SG14 SG14	Ann Beech David Ford David Ford
PG16	5 Novar Dr, Glasgow	22/04/16 24/10/15	13:00 14:00	Replace living room carpet Good condition	SG14 SG37	David Ford Ann Beech

Plain awful.

Example from Connolly & Begg text

27

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?

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

28

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.

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.

29

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..

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

SWINBURNE UNIVERSITY OF TECHNOLOGY

30

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?

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

31

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.

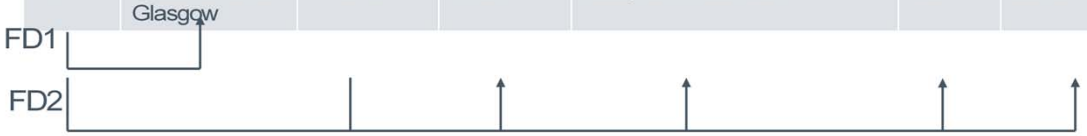
Second Normal Form

› 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



32

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.

Second Normal Form

› Full functional dependencies

in 2NF (has no partial dependencies)

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

FD1

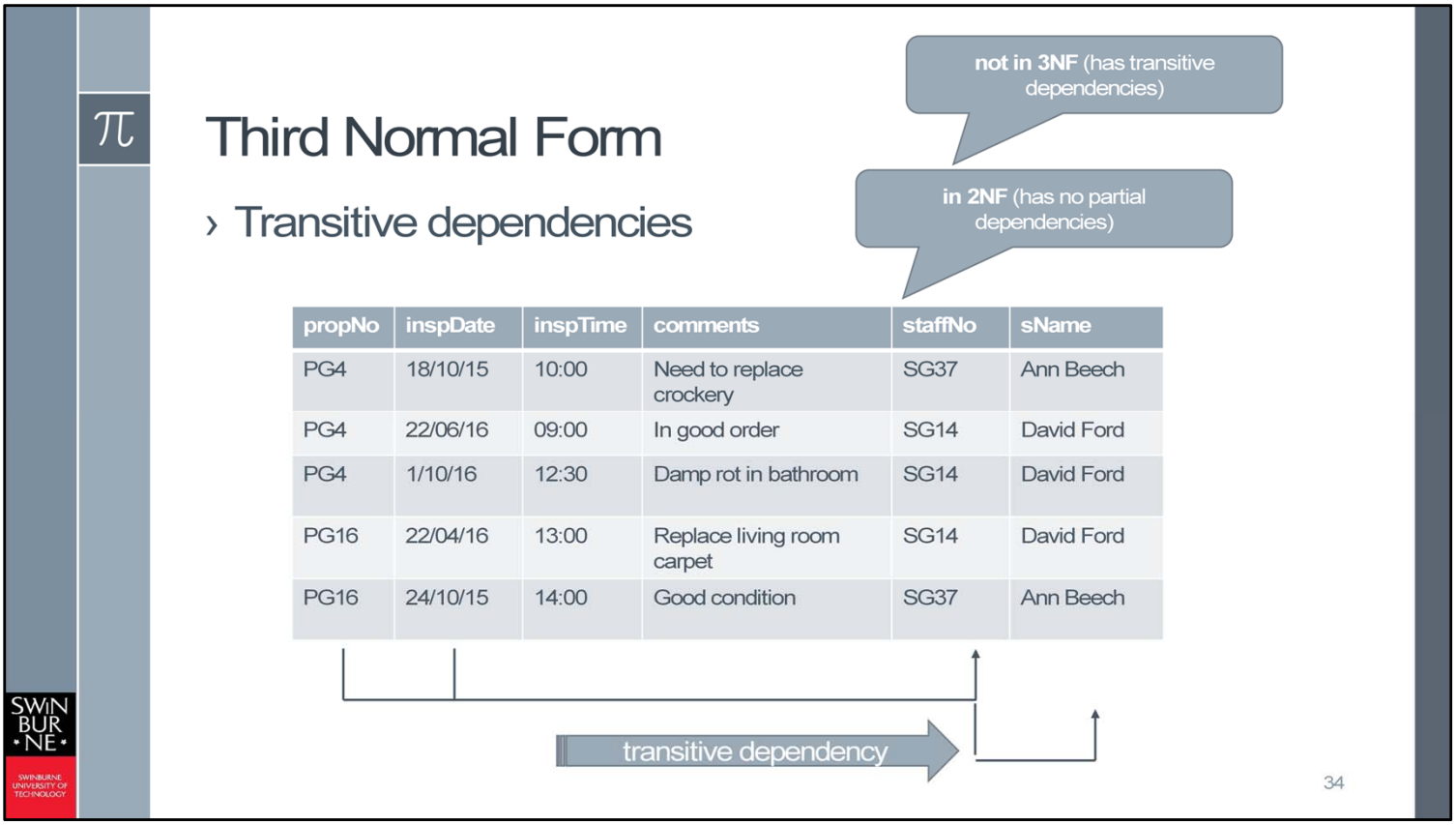
in 2NF (has no partial dependencies)

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

FD2

33

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.



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.

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

35

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.

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

36

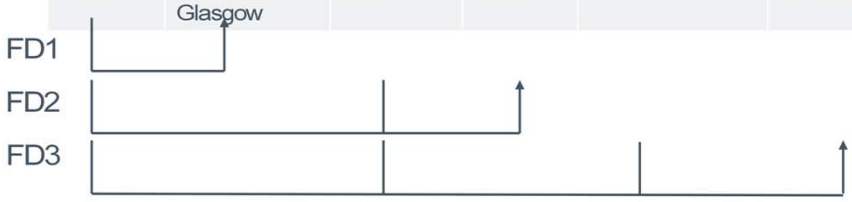
When we looked at repeating groups with 1NF, we had this example where multiple comments were allowed with each inspection. So we know from memory that this table is in 1NF.
But is it in 2NF?
Pause the recording and think about the solution before you continue.

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



37

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.

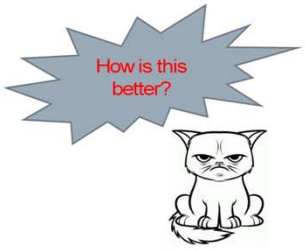
Solution: Second (and Third) Normal Form

› 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

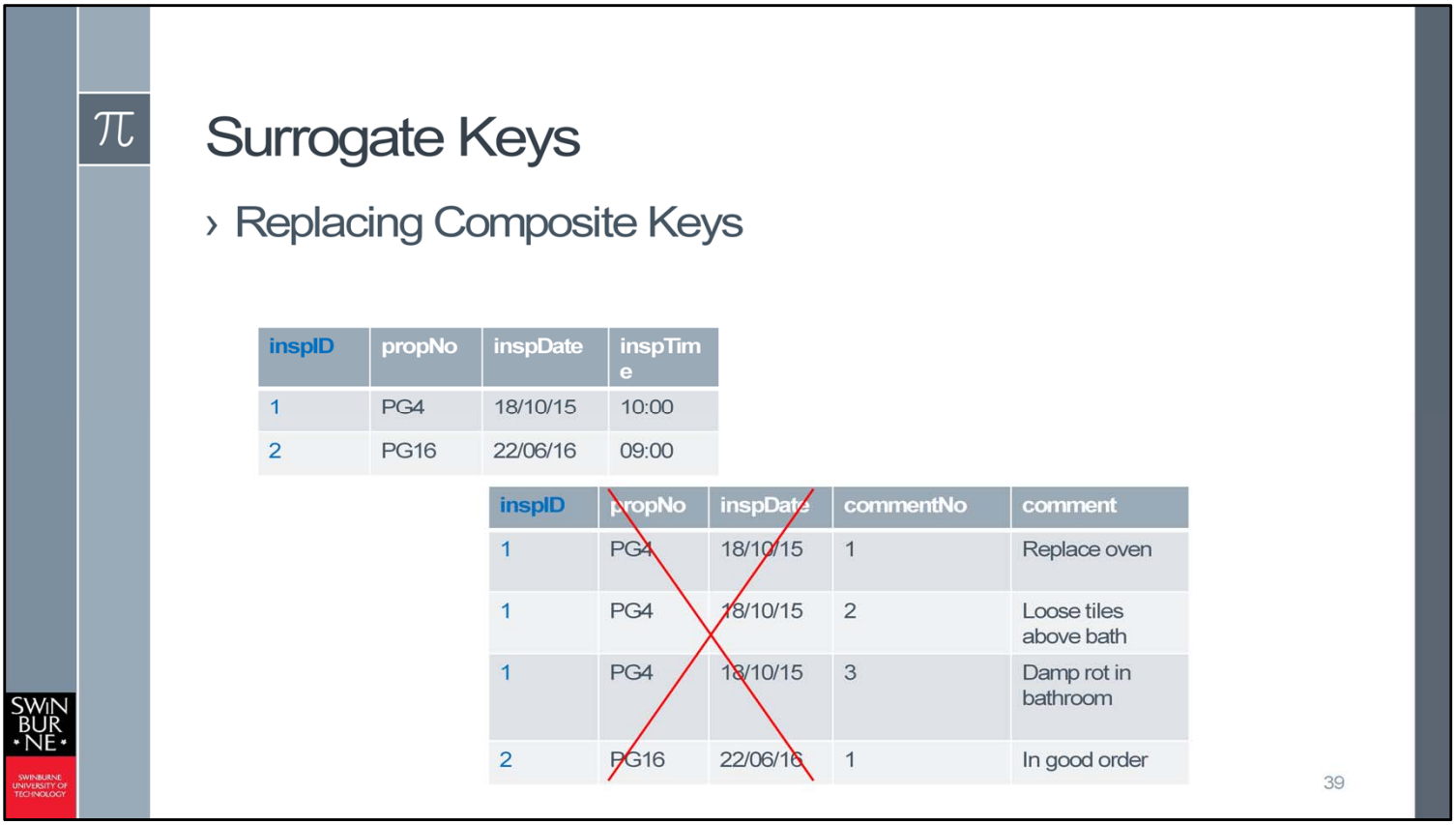
propNo	inspDate	commentNo	comment
PG4	18/10/15	1	Replace oven
PG4	18/10/15	2	Loose tiles above bath
PG4	18/10/15	3	Damp rot in bathroom
PG16	22/06/16	1	In good order



38

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.



A related topic is the notion of surrogate keys. inspID 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 automagically – 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.

40

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.