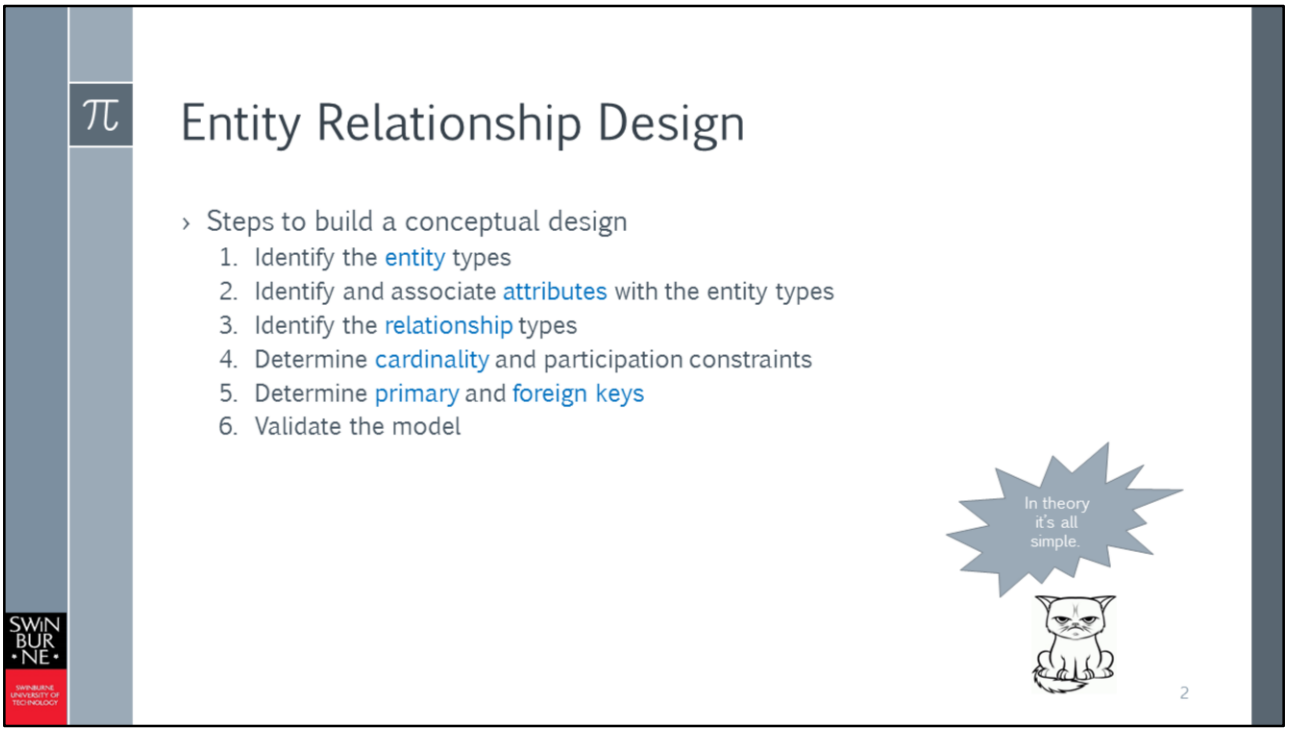
Relational Databases

Entity-Relationship Modelling

SWIN BUR NE
SWINBURNE UNIVERSITY OF TECHNOLOGY

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



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.

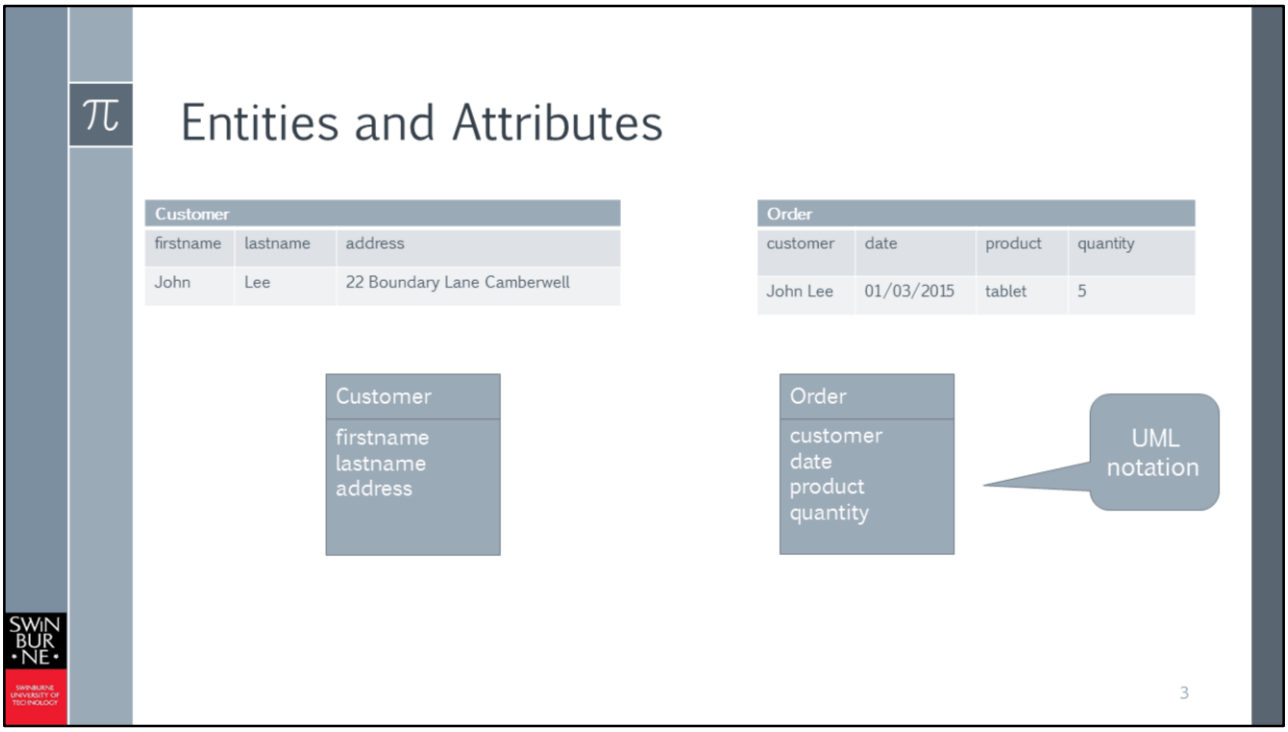
So when we start with the first step, the majority of our entities are easy to detect – customer, product, order, invoice and such will easily spring to mind.
When we get to the attributes, all we have to do is imagine what pieces of information our application needs to keep track of. Most of the time, 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.

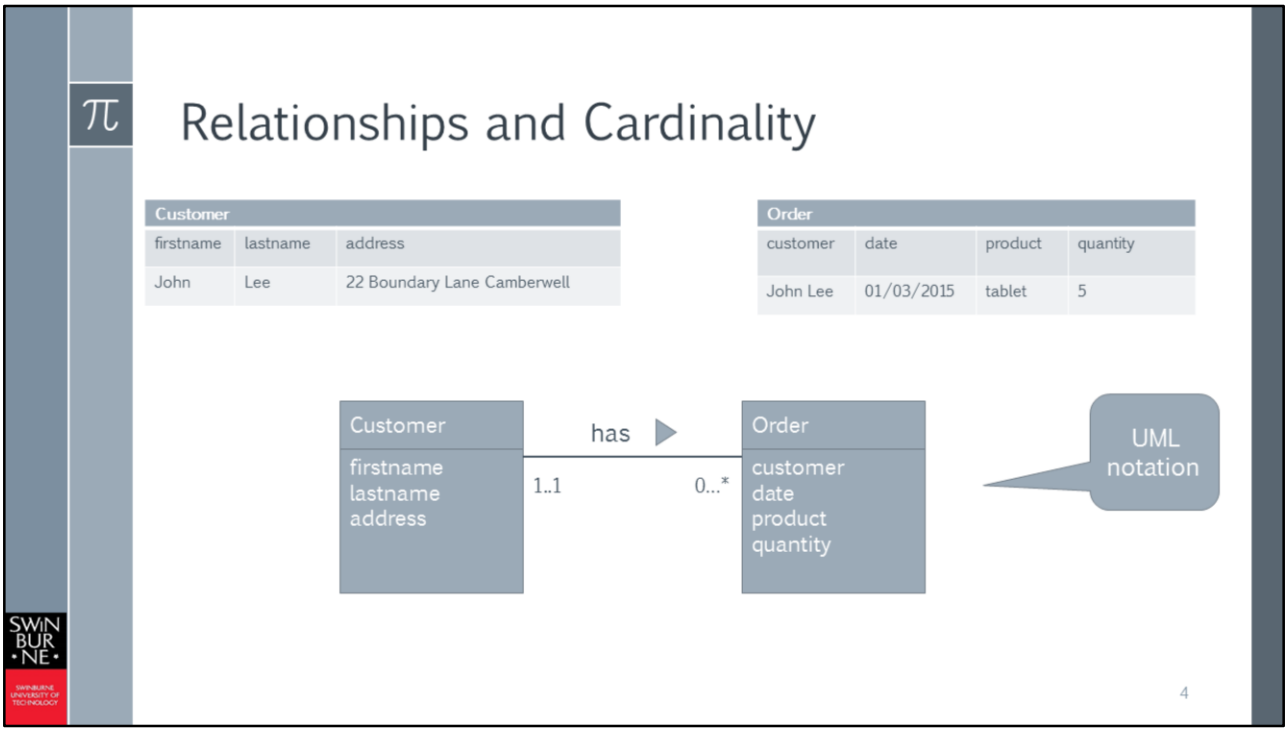
Foreign keys are links that implement the relationships between entities. If we want to link to entities of different tables, we have to include that table's primary key in our table as a foreign key.

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.



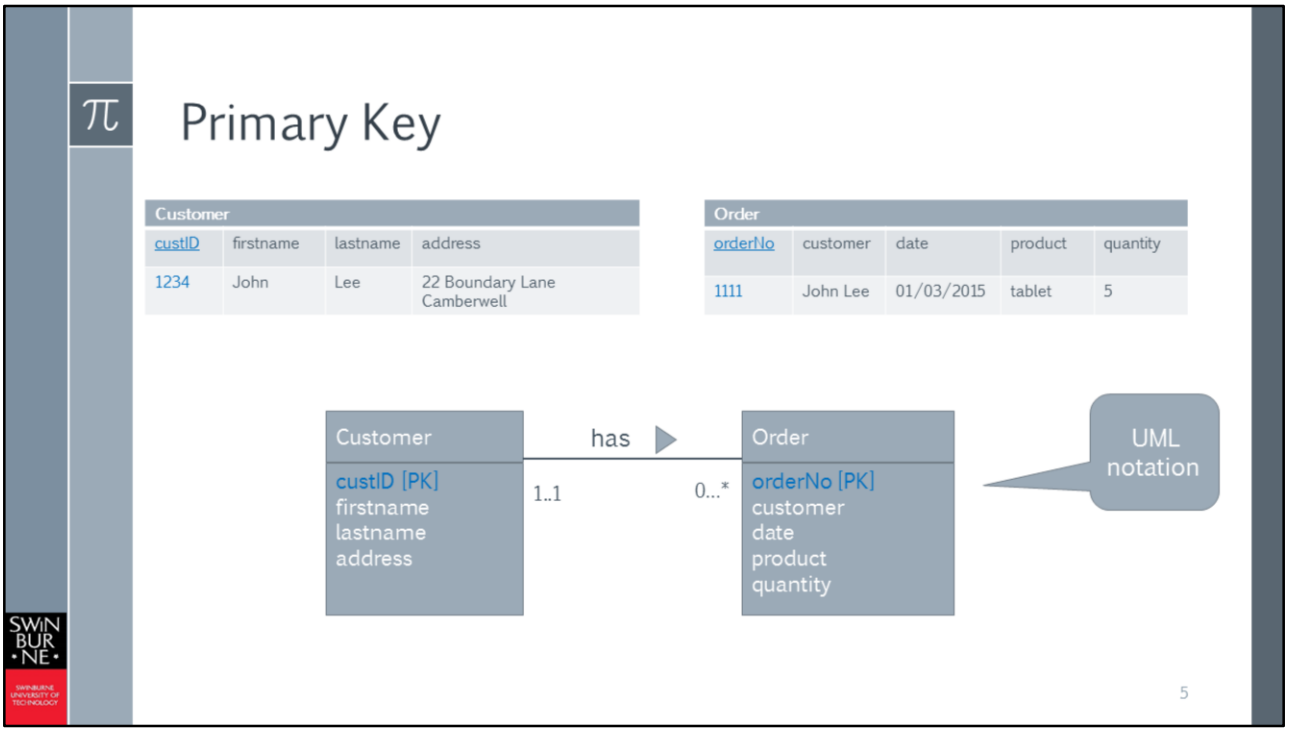
Most of the time, you'll find the steps of identifying entities and their attributes relatively easy. Entities are stored in their own tables – so each different entity you find will lead to creating a new table. The attributes of the entities make columns in a table. Therefore, all entities of the same 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.



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 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.

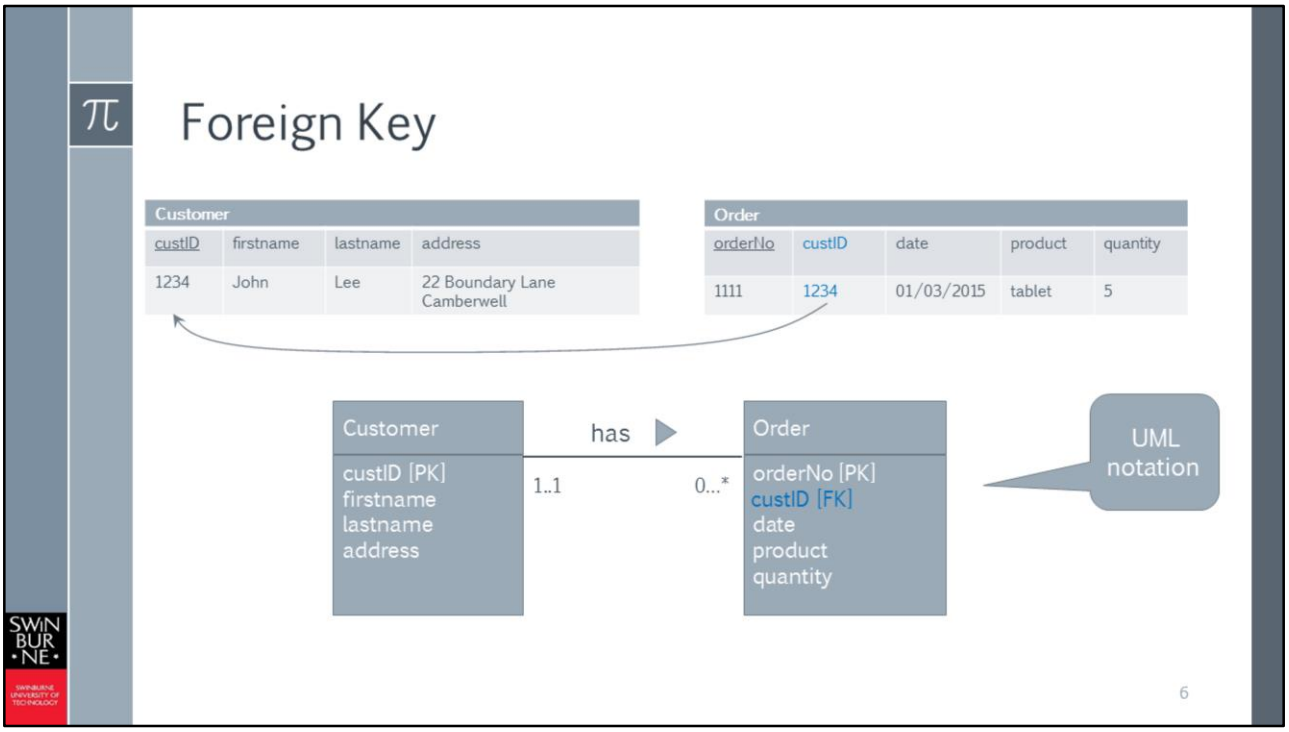
Therefore most companies use ids for many entities. Order and invoice numbers help keep track of the documents, and customer ids make sure one John Smith doesn't get the invoice belonging to another John Smith. In databases, such ids are very helpful – they solve the problem of the primary key.

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.

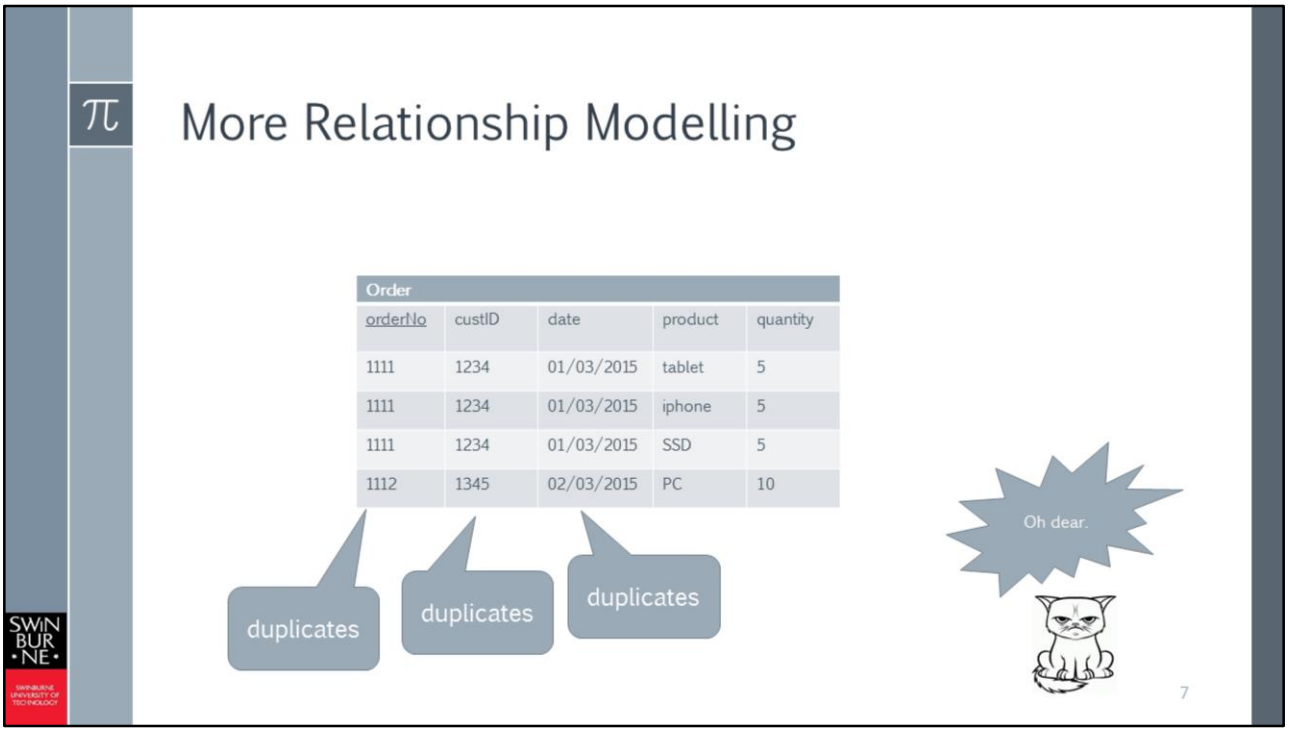


Foreign keys are attributes in a table that link the table's entities to the entities of another table.

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.

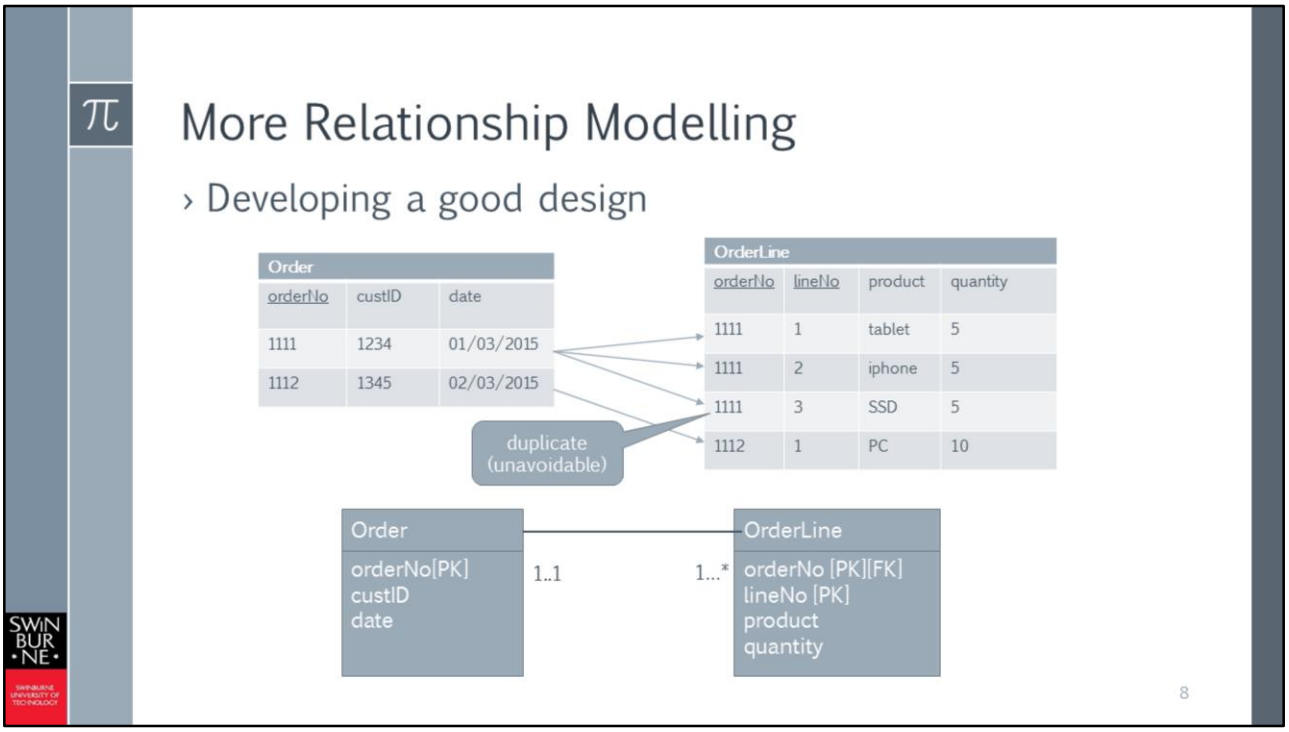
Instead of mentioning the customer's name, we add the primary key attribute of the customer to the order table. Because the customer ID uniquely identifies the customer, we can now make a definitive link between two entities.

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.



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.

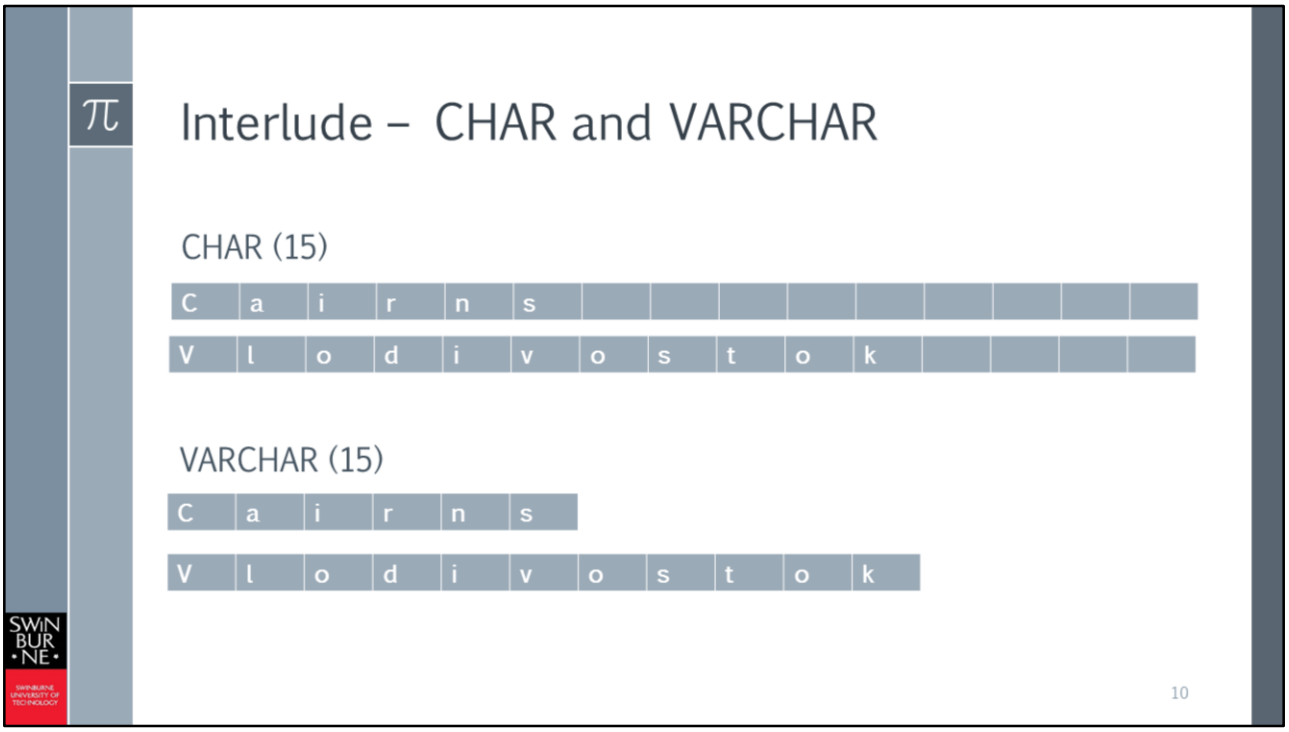
Candidate keys are valid options for primary keys. In the OrderLine table, we can see we have two candidate keys – orderNo and lineNo or orderNo and product.
The case against product is that it is a text field – searching by such a field is time-consuming. So the best candidate here is orderNo and lineNo.

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

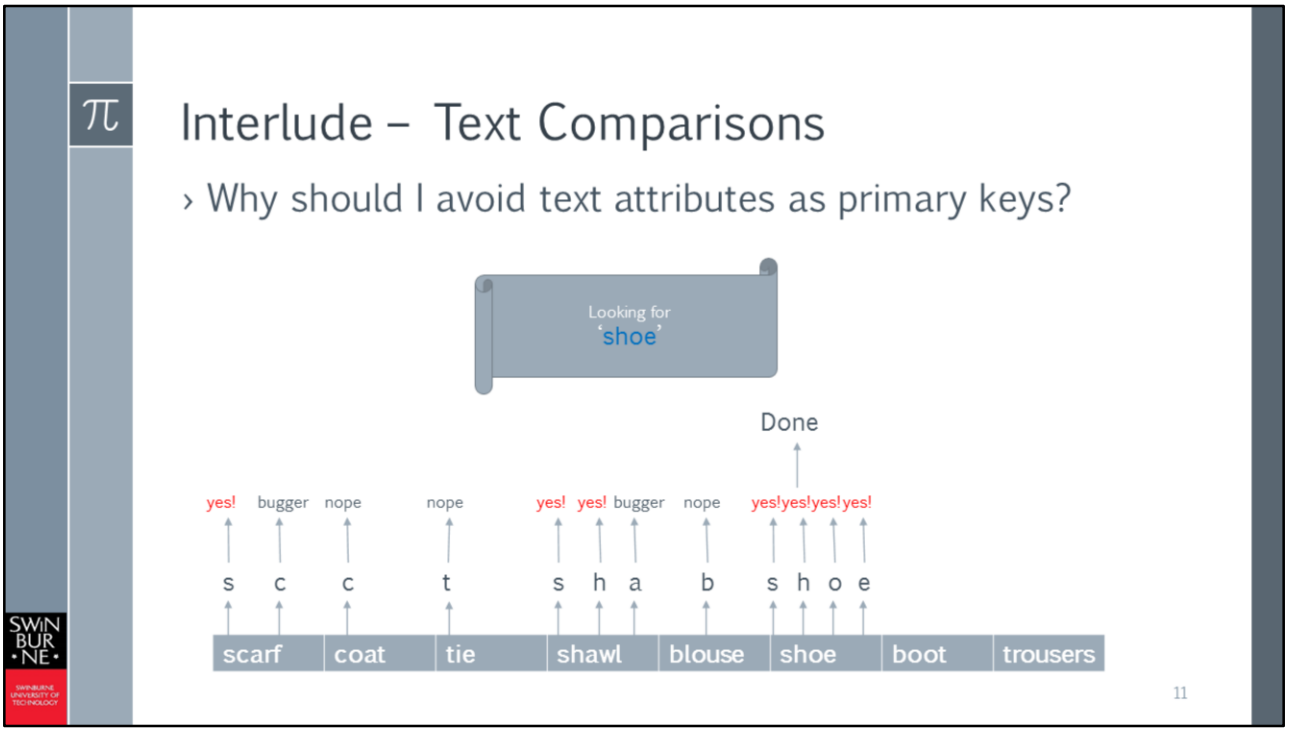
9

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.



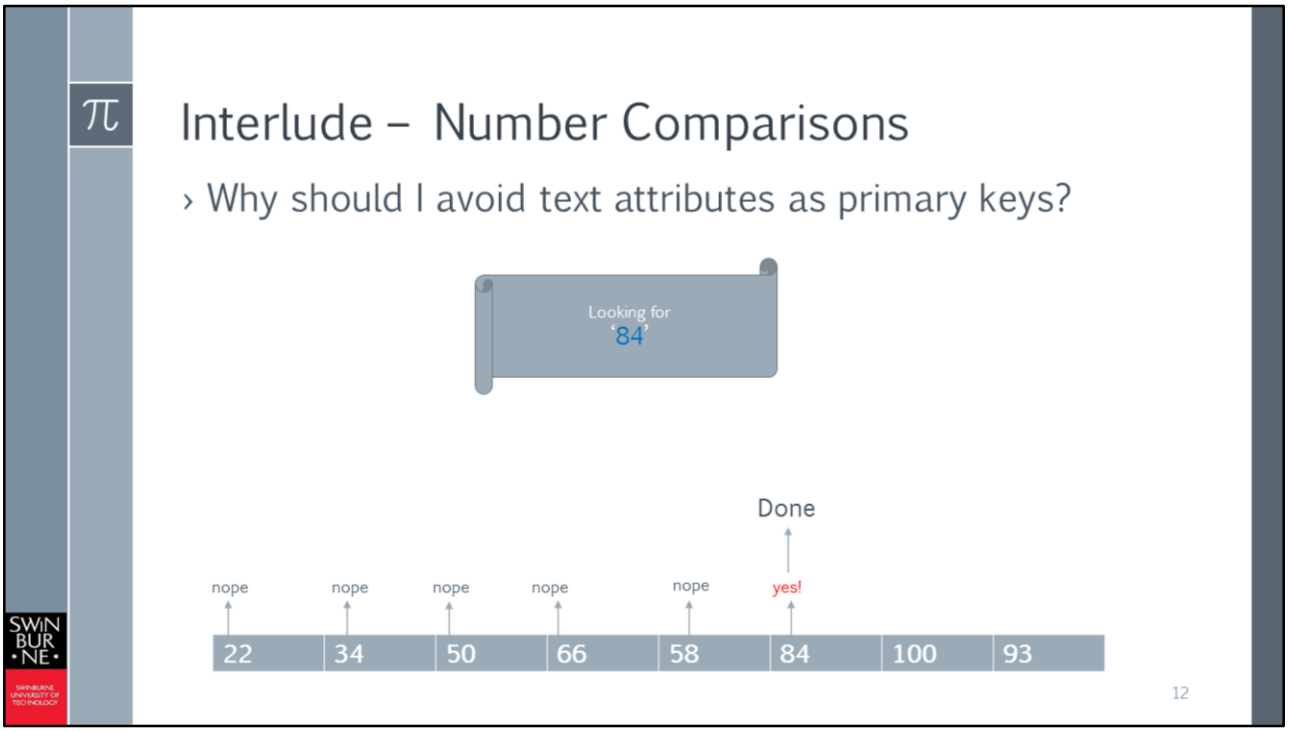
The most important character string formats in relational databases are char and varchar. They when you declare an attribute as char or varchar, you have to tell the DBMS how much space you want for your string field. This number goes in the brackets.

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.



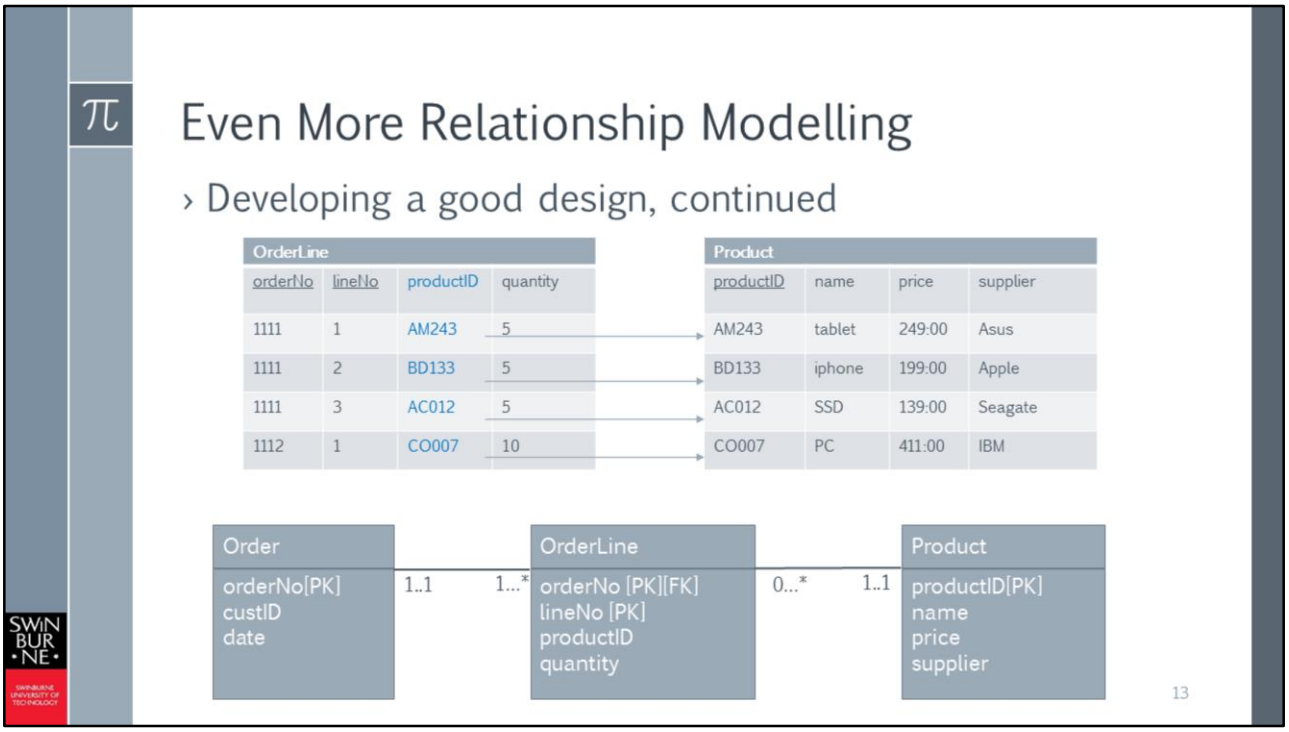
When you search on text attributes, this means string comparisons. Assuming you are looking for the entire content of the text field (and not just a substring inside the field), you have to compare each character to see if there is a match. Comparing a single character means transforming the character 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.



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.



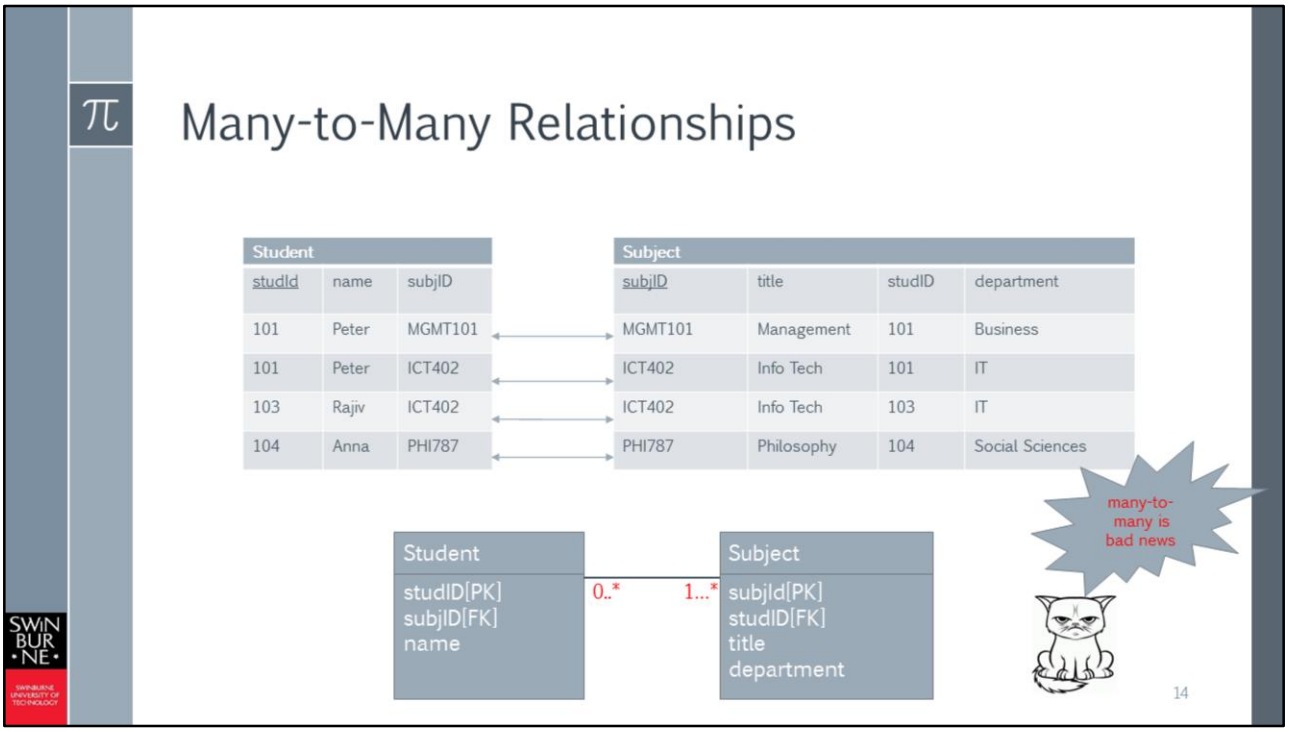
Continuing with the modelling of the order line table, the product attribute that is part of a candidate key is actually not an ideal attribute for an order line table. The product name alone is not going to tell people very much. Don't we need to know the unit price of the product? Availability, supplier? 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.

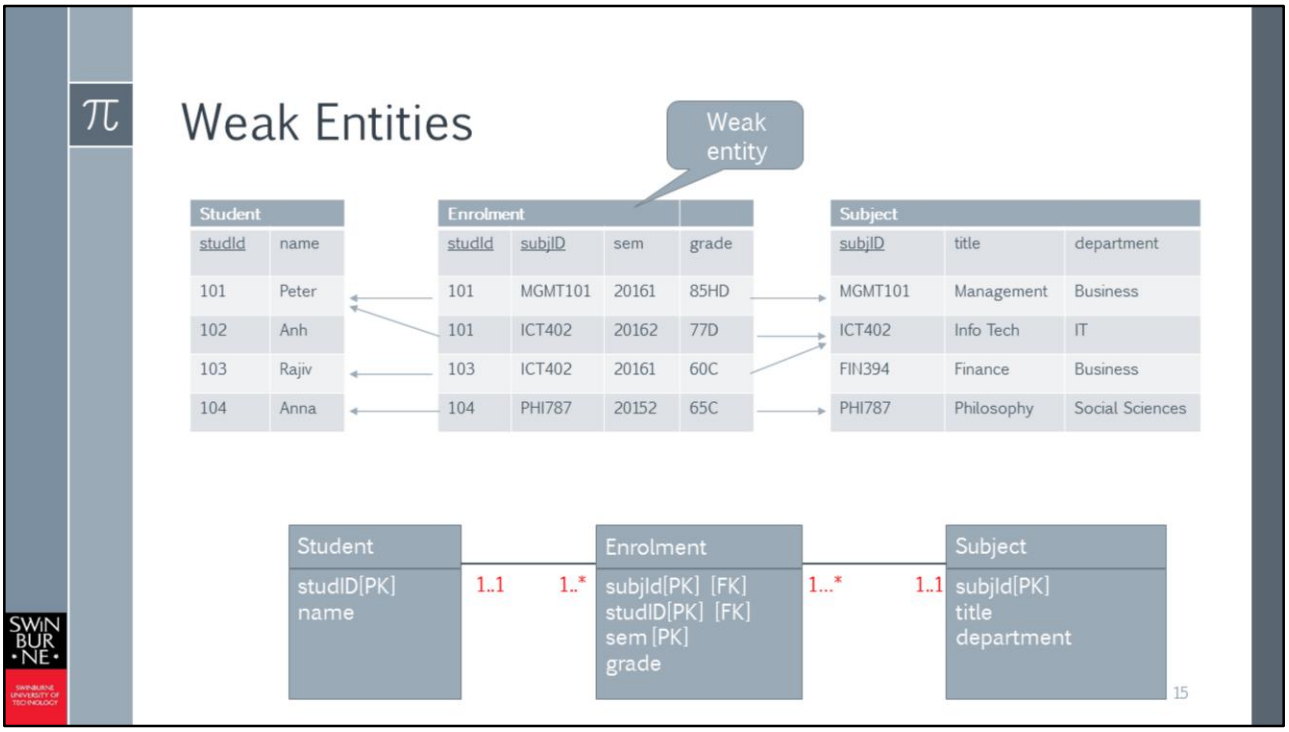
To create a primary key for orderLine, we could also add an id field for OrderLine. Creating a surrogate key is sometimes practical, but in this case we still need orderNo and productID as foreign keys, so using a surrogate key wouldn't help reduce the columns. Surrogate keys are discussed later in the course.



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.

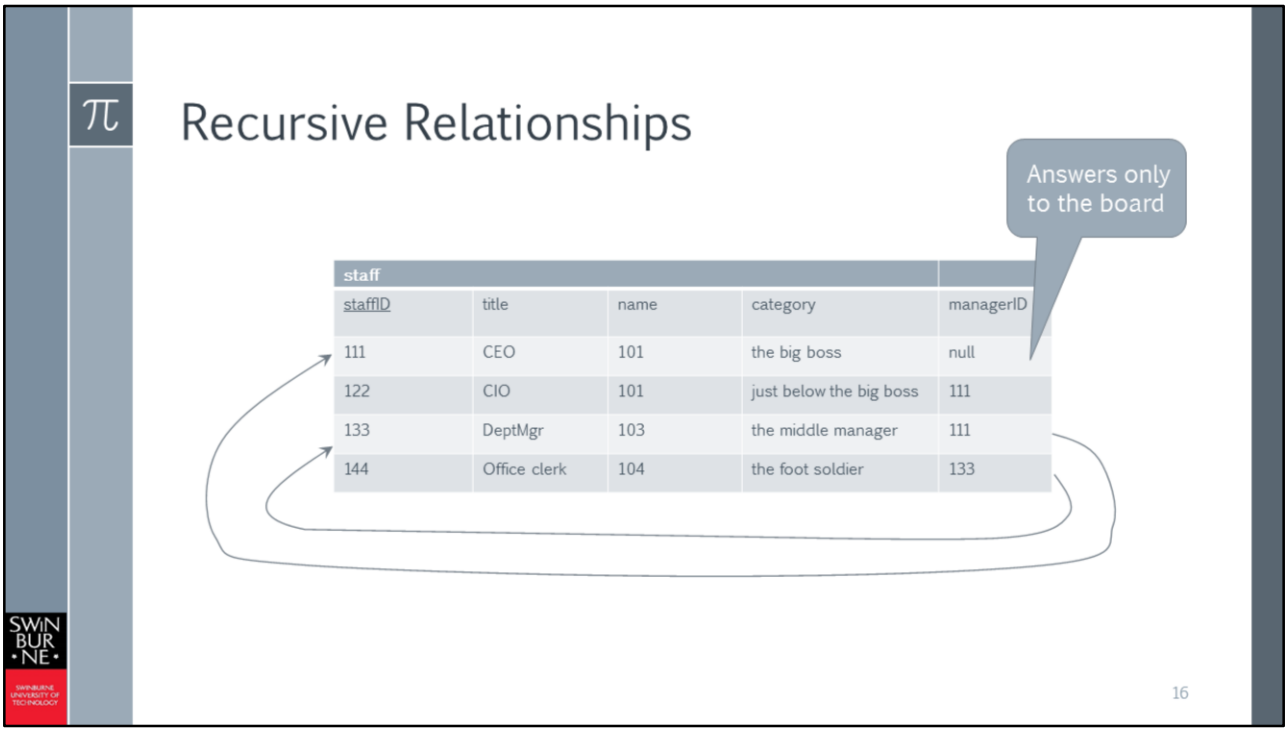
Coping with so many combinations is not practical. Actually, it is not even logical. We have subjects that have titles and convenors, but the subject entity shouldn't have grades for each student. So the simple story is, many-to-many relationships just don't work in the relational world. What do we do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



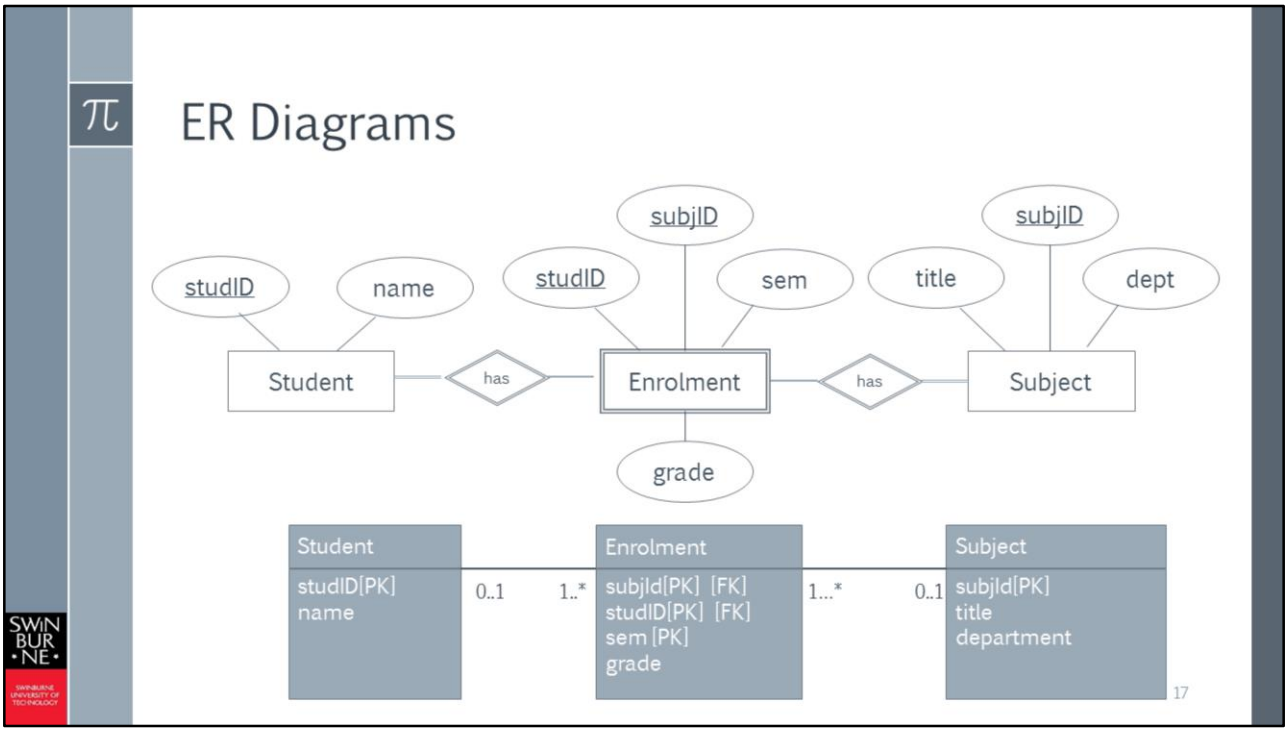
Creating a table that connects the tables student and subject is called expanding the relationship between student and subject. You can see immediately that this makes sense: now we can have columns for the semester a student has taken a subject and the grade the student has achieved.

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.



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

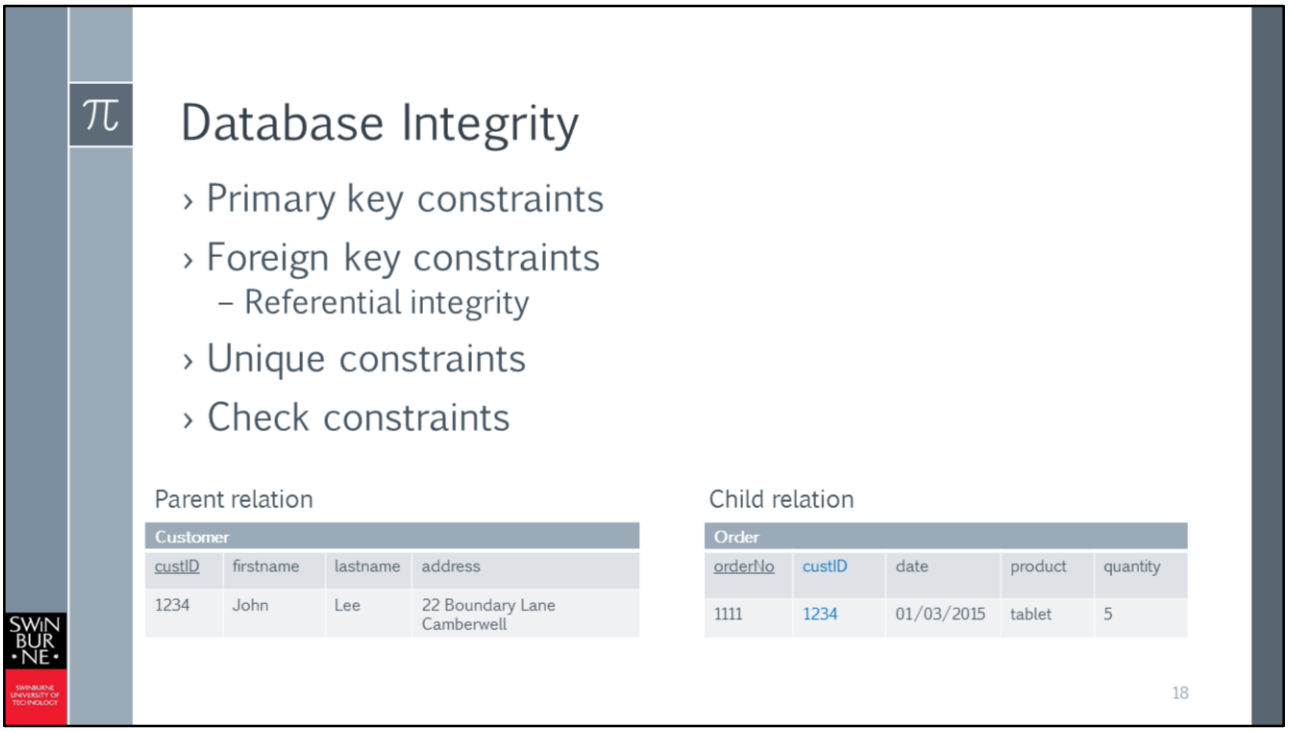


This is an ER diagram for the design shown as UML below. ER diagrams were devised at the early stages of the relational database development. They come in two flavours, this is Chen notation. There is also Crow's feet notation.

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.

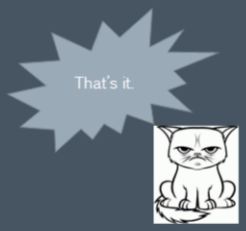
Database products such as Oracle, MySQL and DB2 offer mechanisms that enforce constraints. Once you have decided which of your candidate keys is the most appropriate primary key, you need to implement the table with a primary key constraint. This will enforce uniqueness – the DBMS will not allow you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