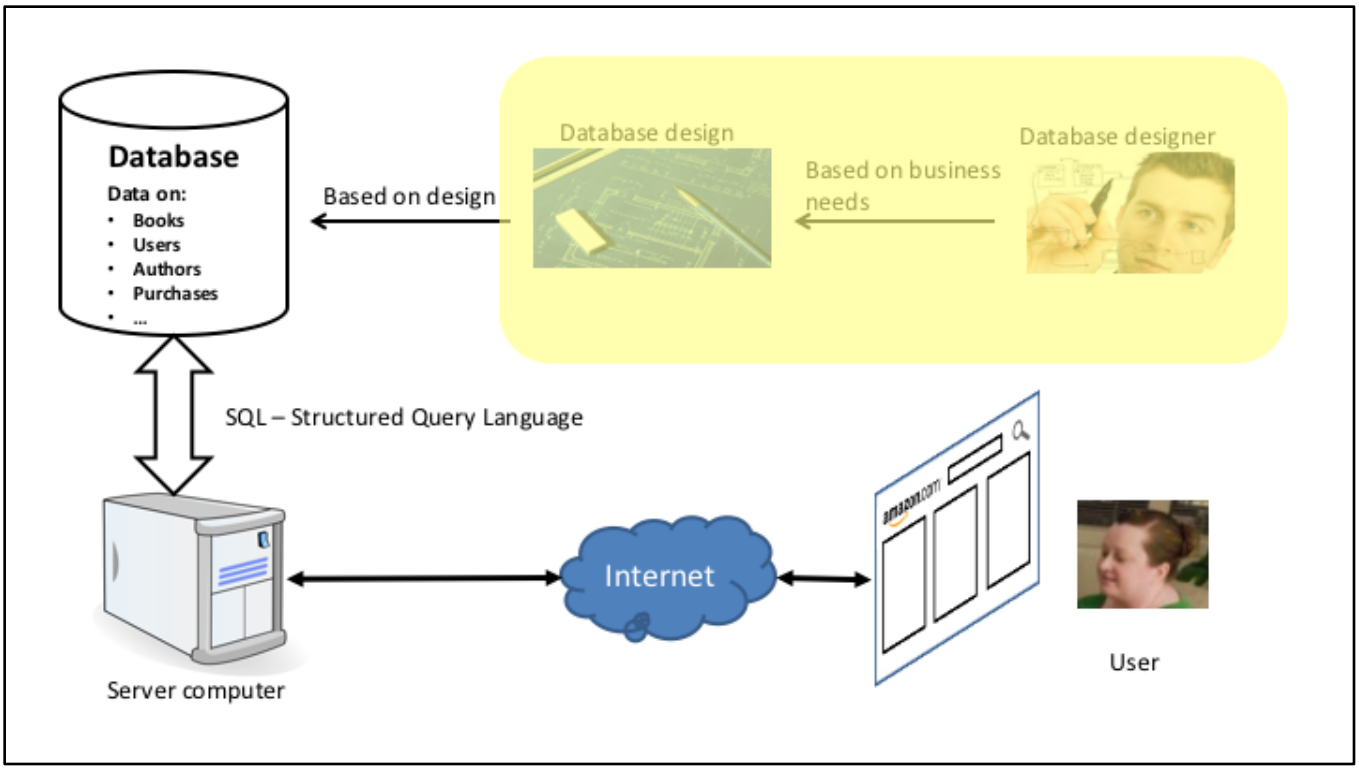
Database Design: Part 3



We have already looked at how SQL can help us to retrieve almost any information we need from a relational database.

We have gone into database design to some extent – at least to the extent of encountering the core concepts.

We will deepen our understanding of the ideas this week.

We will focus on deepening our understanding while also looking at how we might address bugger chunks of business knowledge.

You will also encounter a few new concepts that will round off your mastery of database design.

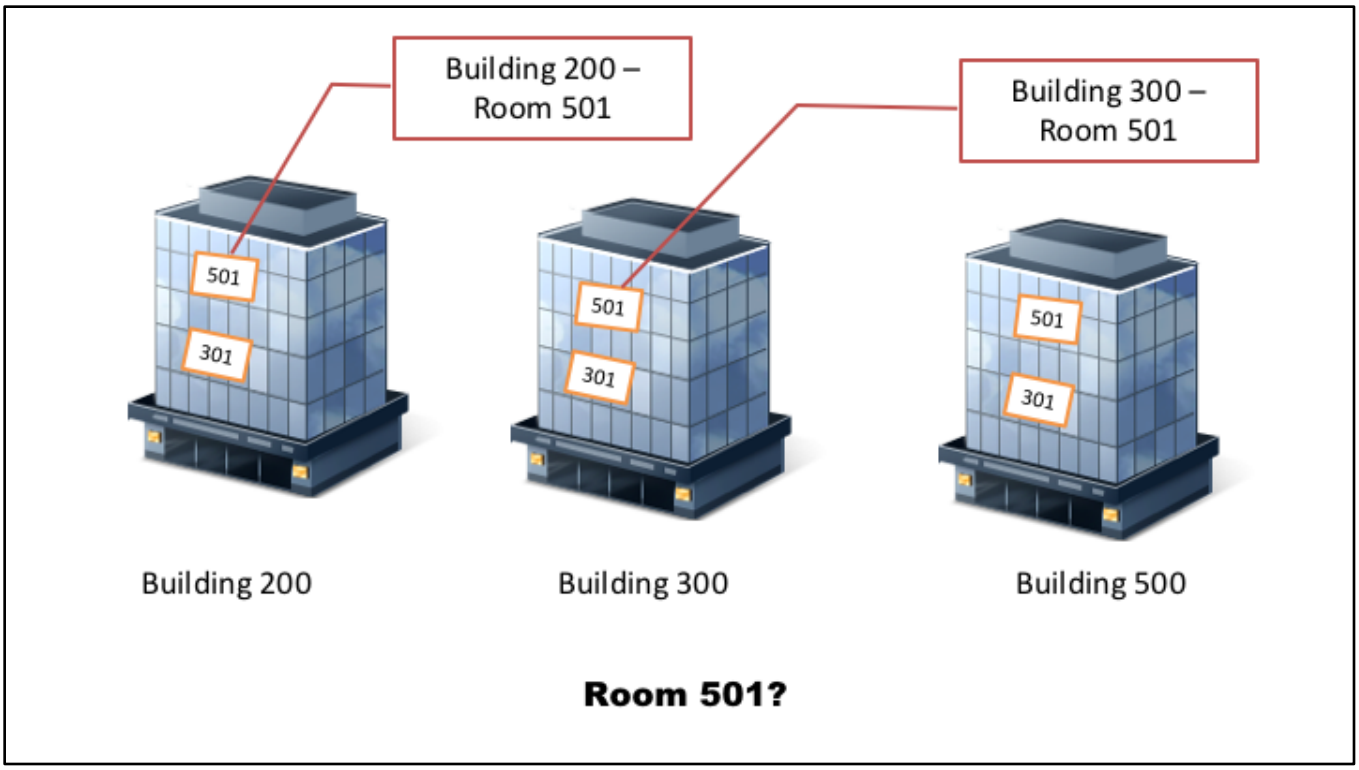
Key Migration Recap

When to use?

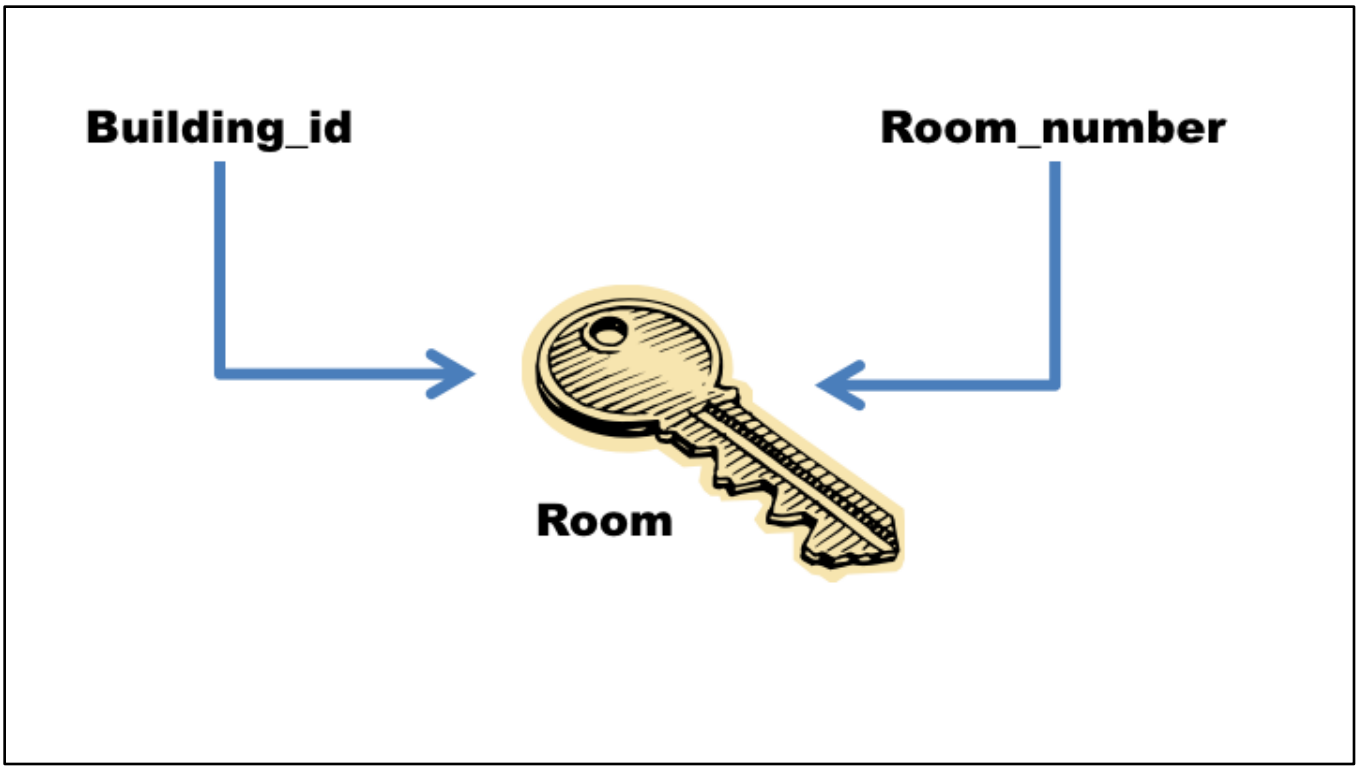
Let us first revisit key migration – a simple concept that might seem tricky at first.

Do try and understand the underlying concept rather than aiming to remember it by rote – the rote approach seldom works; even if it seems to sometimes come in handy in shallow examinations, it lets you down badly in crucial situations.

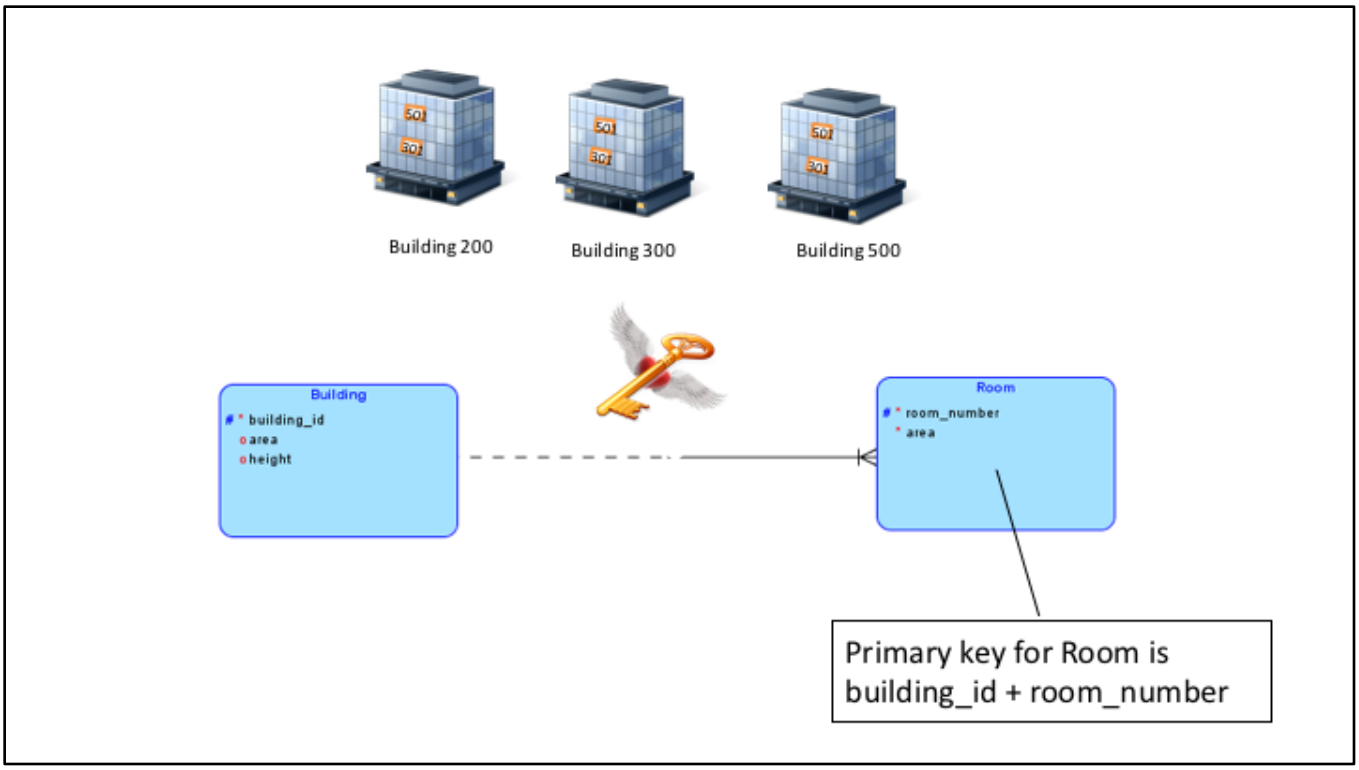
You are paying good money. What a waste to fritter it away on just a measly grade.

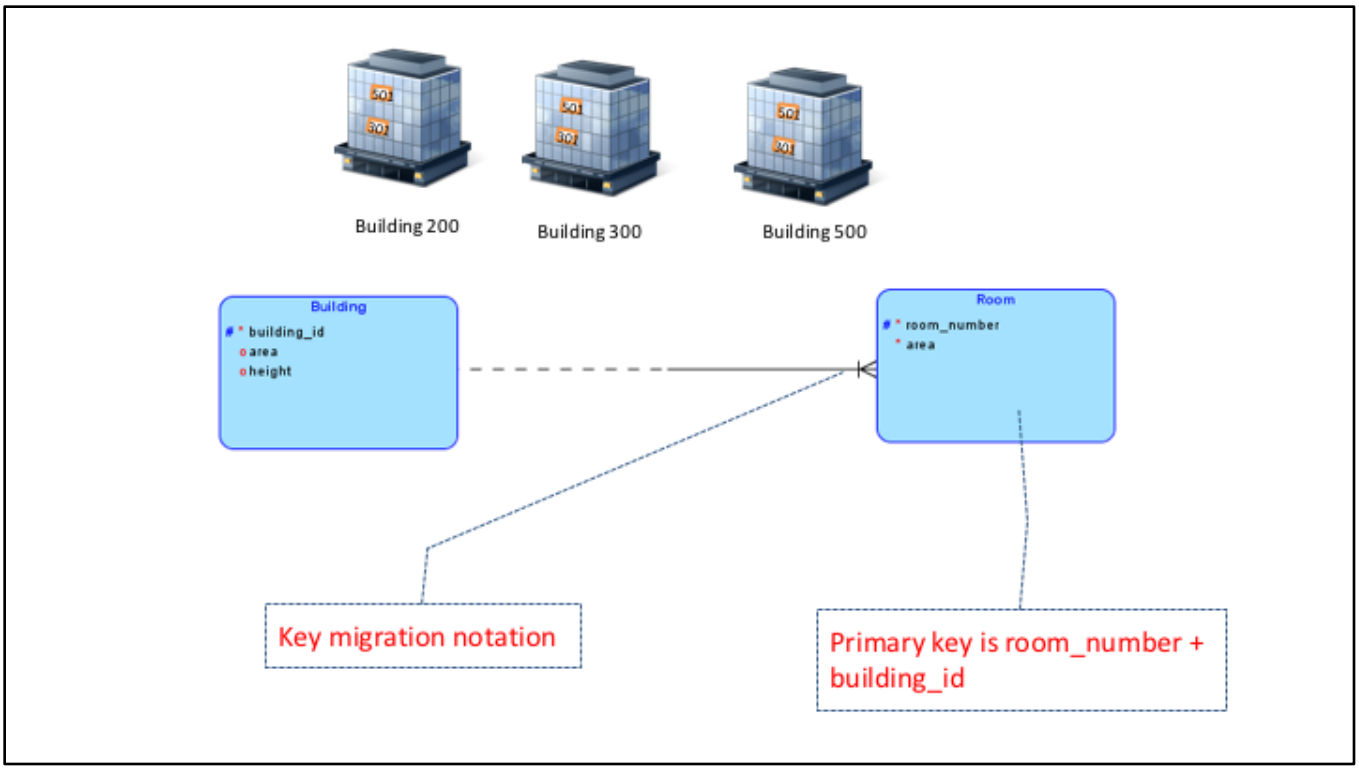


The same room number can occur in several buildings and hence the room number alone does not suffice to uniquely identify a room across all the buildings.



The combination of room number and building number provides a unique identifier for a room. The combination will be unique across all rooms of all buildings.


Building 200
Building 300
Building 500
Building
* building_id
o area
o height
Room
* room_number
* area
Primary key for Room is
building_id + room_number

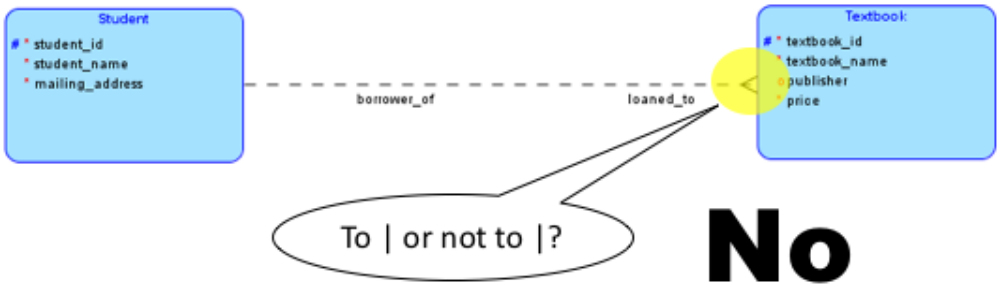

Building 200
Building 300
Building 500
Building
* building_id
o area
o height
Room
* room_number
* area
Key migration notation
Primary key is room_number + building_id

Your turn

To migrate or not to migrate?

A school keeps several textbooks and loans them out to students for the duration of a course.

- Each student could have zero or more books on loan; each book can be loaned to at most one student.
- Each student has a unique student_id; each text book has a unique textbook_id.



To | or not to |?

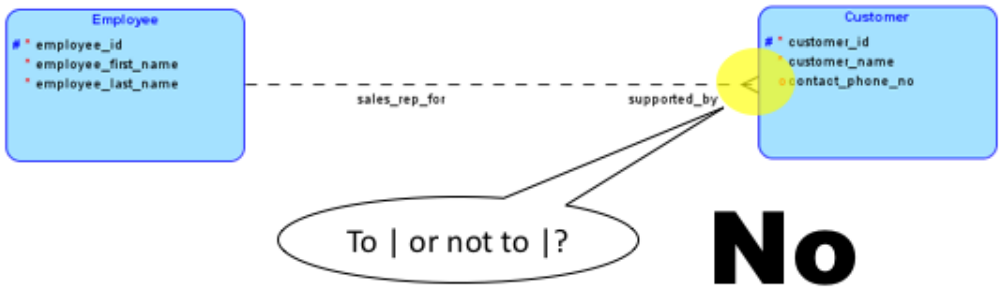
No

In this situation each entity type has its own unique identifier or primary key that, without help from any other field, can identify an instance.

So we do not need any key migration notation here.

A company has several employees and several customers.

- Each customer is assigned one employee to serve as the sales rep for that customer.
- Each employee might be the sales rep for zero or more customers.
- Each employee has a unique employee_id and each customer has a unique customer_id

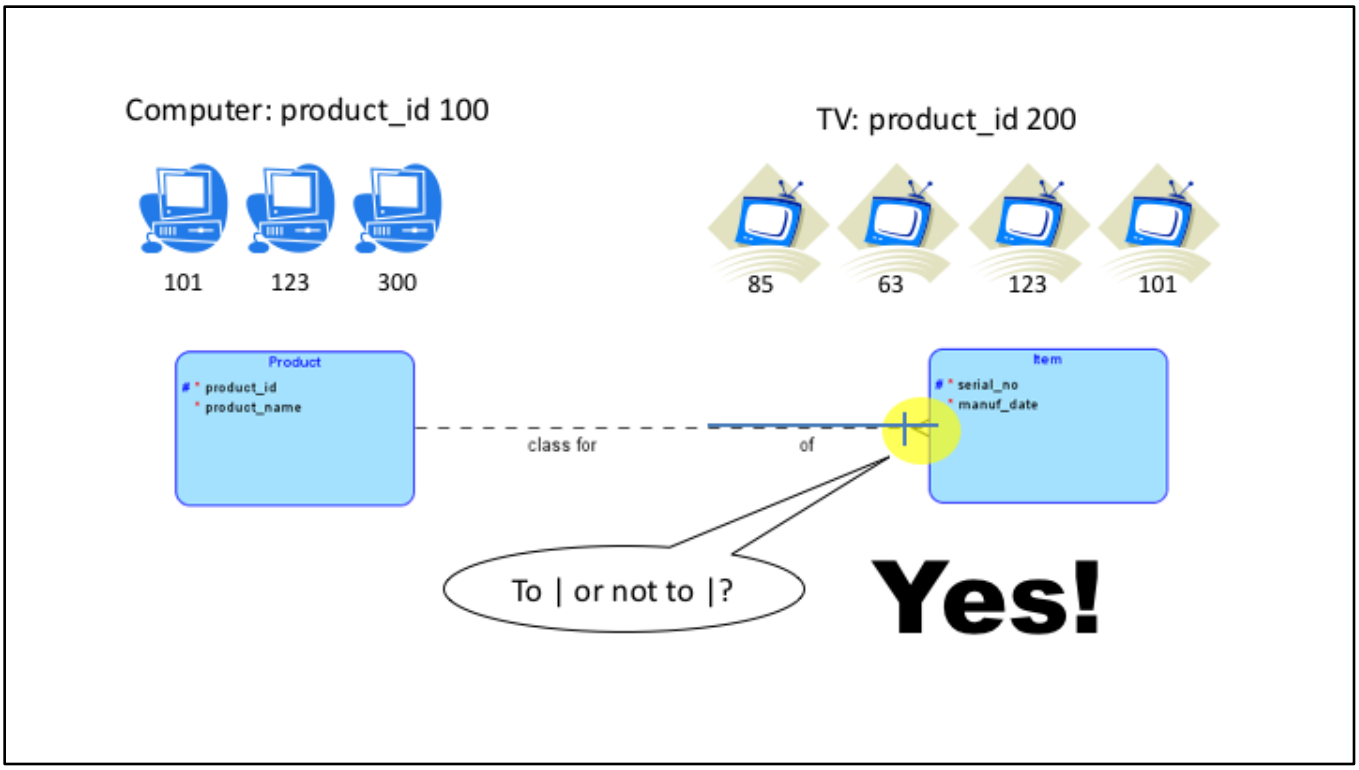


To | or not to |?

No

In this situation each entity type has its own unique identifier or primary key that, without help from any other field, can identify an instance.

So we do not need any key migration notation here.

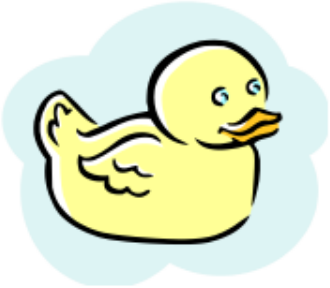


A company has several products and, for each product, has many units in stock. Within each product, each unit is assigned a unique serial_no, but the same serial_no could repeat across products.

- For example, we can have a "computer" (product_id 100) with serial_no 123 and also a "TV" (product_id 200) with serial_no 123.
- However, no two computers can have identical serial_nos, nor can two TVs can have identical serial_nos.
- To identify the above instance of TV, we need to specify the product_id 100 as well as its serial_no 123.

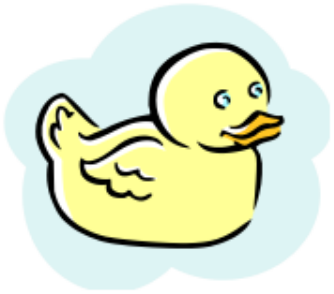
In this case, **we do need the key migration notation** because we need both the product_id and the item's serial_no to uniquely identify an instance of Item across items of all products.

Associative Entity Recap



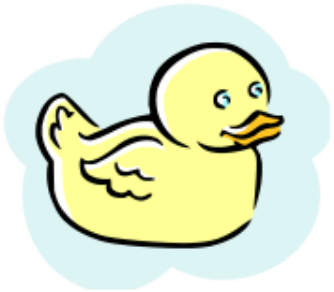
If it swims like a duck and quacks like a duck it must be a duck

When to use?



If a relationship seems to have attributes, then it is an entity type

m:n relationship?



Always create a new entity type – an associative entity

Whenever we encounter m:n relationships, we always create a new entity type, called an associative entity.

Why should we do this?

Understand this section. This represents the crux of database design. Without deeply understanding this section, you can NEVER design useful relational databases.

Tycoons

- Many people and many buildings.
- Each person has ownership stake in one or more buildings
- Each building has one or more owners who share the ownership.

The Gatsby

Jill David

Crystal Palace

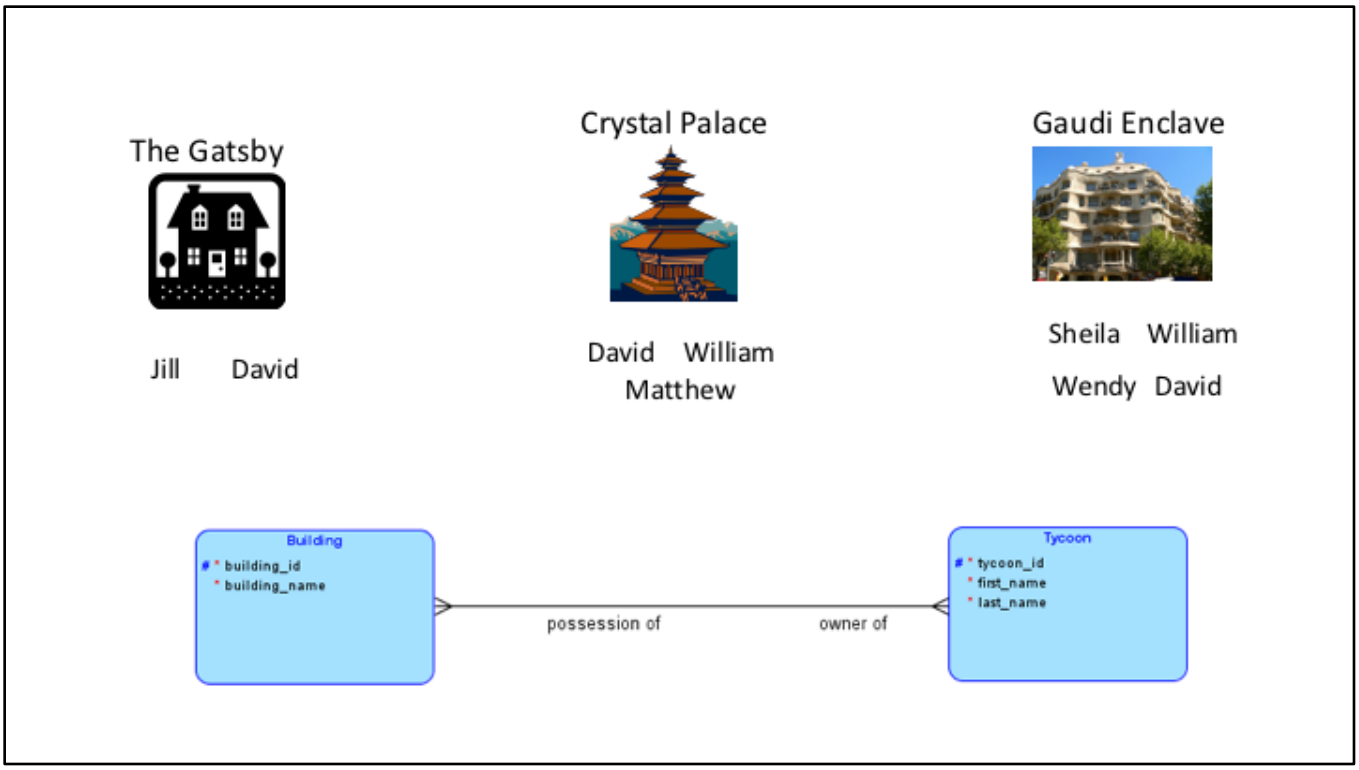
David William
Matthew

Gaudi Enclave

Sheila William
Wendy David

We have several people and several buildings. People jointly own buildings. Each person has an ownership stake in one or more buildings and each building has one or more owners who have a stake in it. This results in a m:n relationship.

For example, Jill and David jointly own The Gatsby and David, William and Matthew jointly own Crystal Palace and so on.



We first create the obvious m:n relationship that the description implies. We have also given suitable names to the relationships.

We have earlier indicated that whenever we see a m:n relationship we need to create a new entity type. Why?

You can either take this as Gospel Truth or try and understand why.

I think you will prefer the latter approach in this context.

Tycoons

tycoon_id	first_name	last_name
10	Jill	McIntyre
20	Sheila	Jones
30	Wendy	Hu
40	David	Scrooge
50	William	Owner
60	Matthew	Warner

Buildings

building_id	building_name
321	The Gatsby
500	Crystal Palace
343	Gaudi Enclave

tycoon_id	building_id
10	321
40	321
40	500
50	500
60	500
20	343
50	343
30	343
40	343

Let us suppose that we need to represent this situation in the form of relational tables.

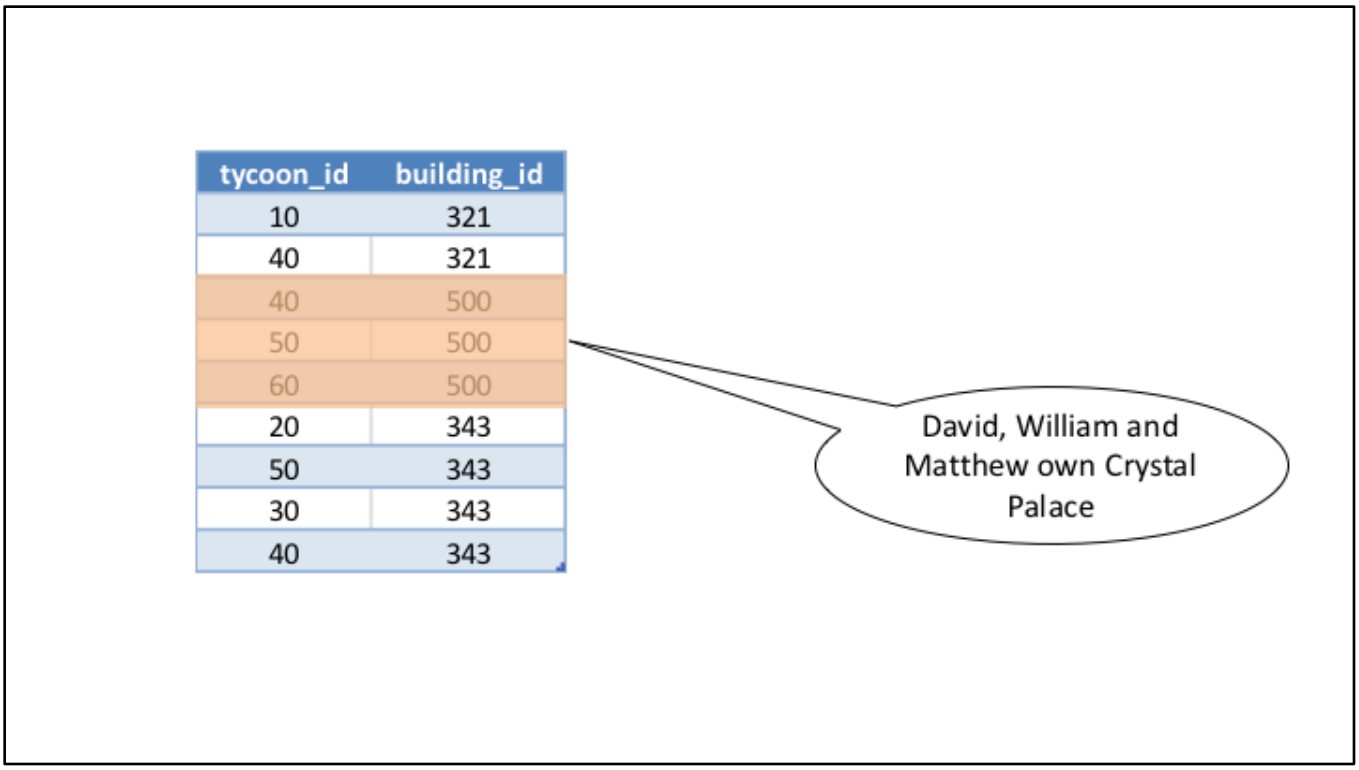
Given that each Tycoon has an ownership stake in many buildings and that each building could be jointly owned by many tycoons, we cannot represent this relationship as we have done in the past – adding a foreign key to one of the tales.

Think about this before moving on. If you do not understand this, then you do not understand ERD – no matter how much of the other things you understand or how cool your ERDs look. Without getting this, you cannot use ERDs in practice. As simple as that.

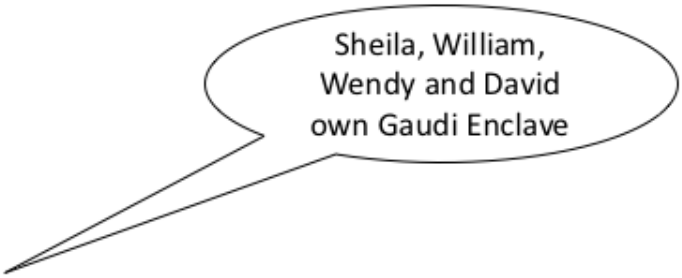
The third table on the right shows how we can use a totally new table to capture the m:n relationship. Each row of this table represents the ownership of one person in one building. Thus if a particular building has two joint owners (such as The Gatsby), then that building will have two rows in this new table. Conversely, if a person has ownership stakes in three buildings (as David does), then we will see three rows for that person.

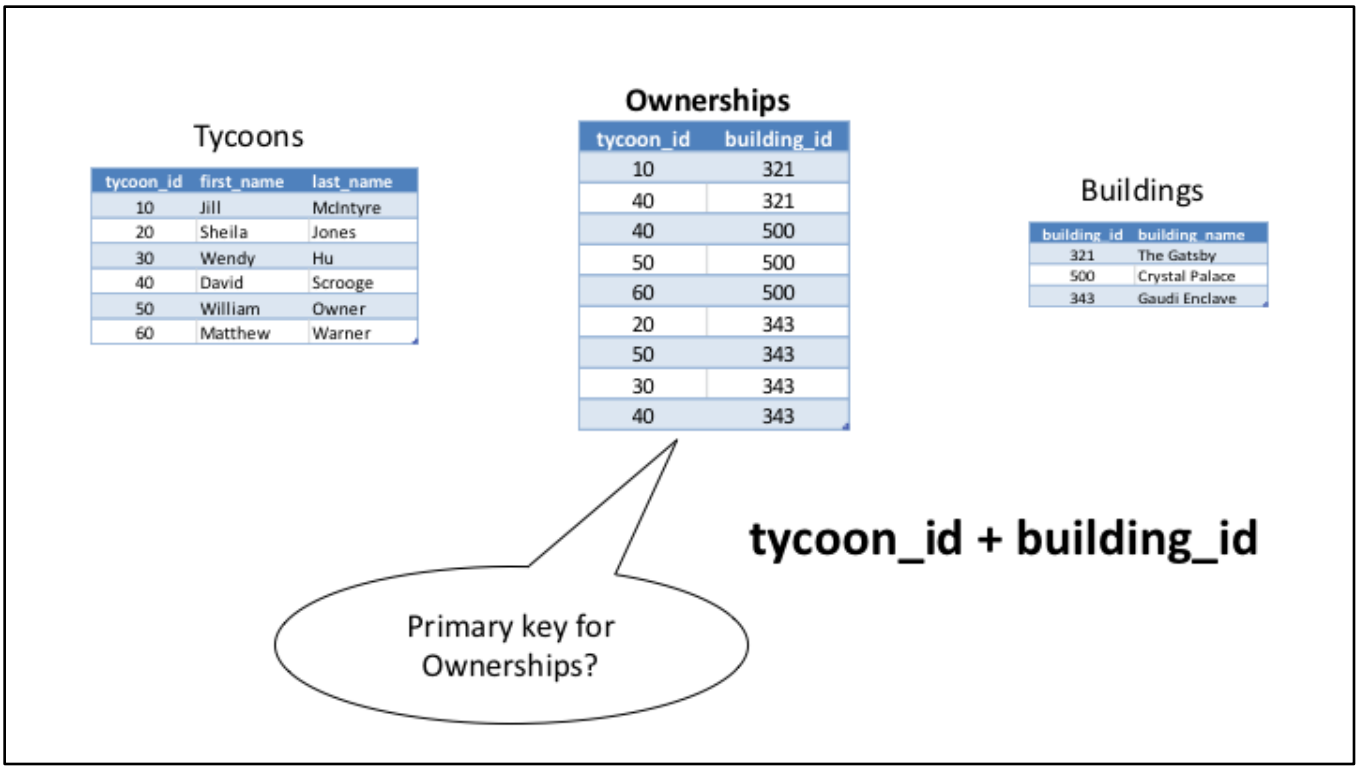
The following three slides illustrate this point diagrammatically.

tycoon_id	building_id
10	321
40	321
40	500
50	500
60	500
20	343
50	343
30	343
40	343


tycoon_id
building_id
10
321
40
321
40
500
50
500
60
500
20
343
50
343
30
343
40
343
David, William and Matthew own Crystal Palace

tycoon_id	building_id
10	321
40	321
40	500
50	500
60	500
20	343
50	343
30	343
40	343




Tycoons
tycoon_id first_name last_name
10 Jill McIntyre
20 Sheila Jones
30 Wendy Hu
40 David Scrooge
50 William Owner
60 Matthew Warner
Ownerships
tycoon_id building_id
10 321
40 321
40 500
50 500
60 500
20 343
50 343
30 343
40 343
Buildings
building_id building_name
321 The Gatsby
500 Crystal Palace
343 Gaudi Enclave
Primary key for Ownerships?
tycoon_id + building_id

Tycoons

tycoon_id	first_name	last_name
10	Jill	McIntyre
20	Sheila	Jones
30	Wendy	Hu
40	David	Scrooge
50	William	Owner
60	Matthew	Warner

Ownerships

tycoon_id	building_id
10	321
40	321
40	500
50	500
60	500
20	343
50	343
30	343
40	343

Buildings

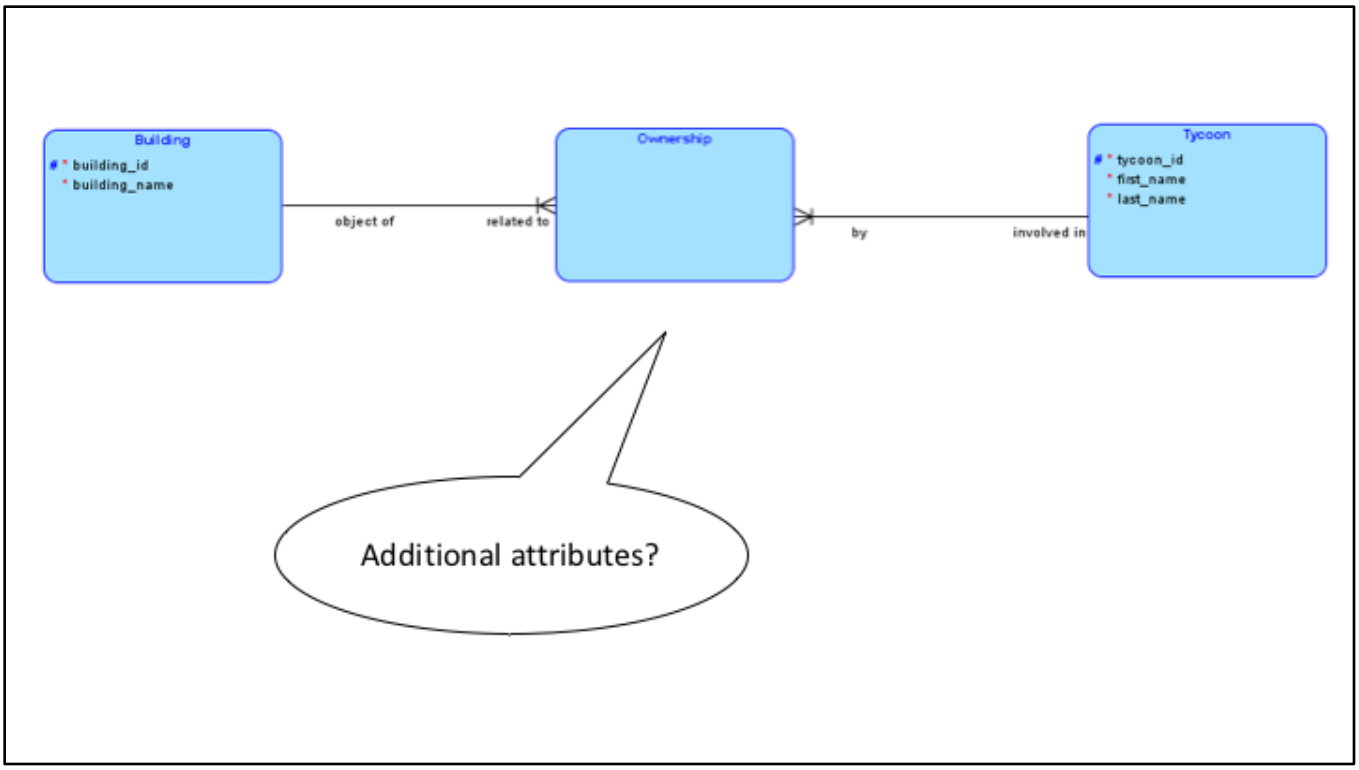
building_id	building_name
321	The Gatsby
500	Crystal Palace
343	Gaudi Enclave

This third table is what the associative entity represents on the ERD. Since each row of the table represents an ownership stake that a tycoon has in a building, I felt that Ownership might be a good name for the new entity type. We swill talk about its attributes shortly.

Going from our original description, every tycoon has an ownership stake in at least one building and this explains why the line near Tycoon is solid. Similarly, every building has at least one person with an ownership stale in it and this explains the solid line near Building.

Since each Tycoon can be connected to many buildings and each Building can have many Tycoons owning it, we see the two crow-feet on the associative entity type.

Clearly the primary key for Ownership can be the combination of tycoon_id and building_id and therefore I have used key migration to capture this.



Associative entity types usually have their own attributes. In rate cases they might not have any attributes of interest. However, I can bet that you will always be able to find meaningful attributes in every single case.

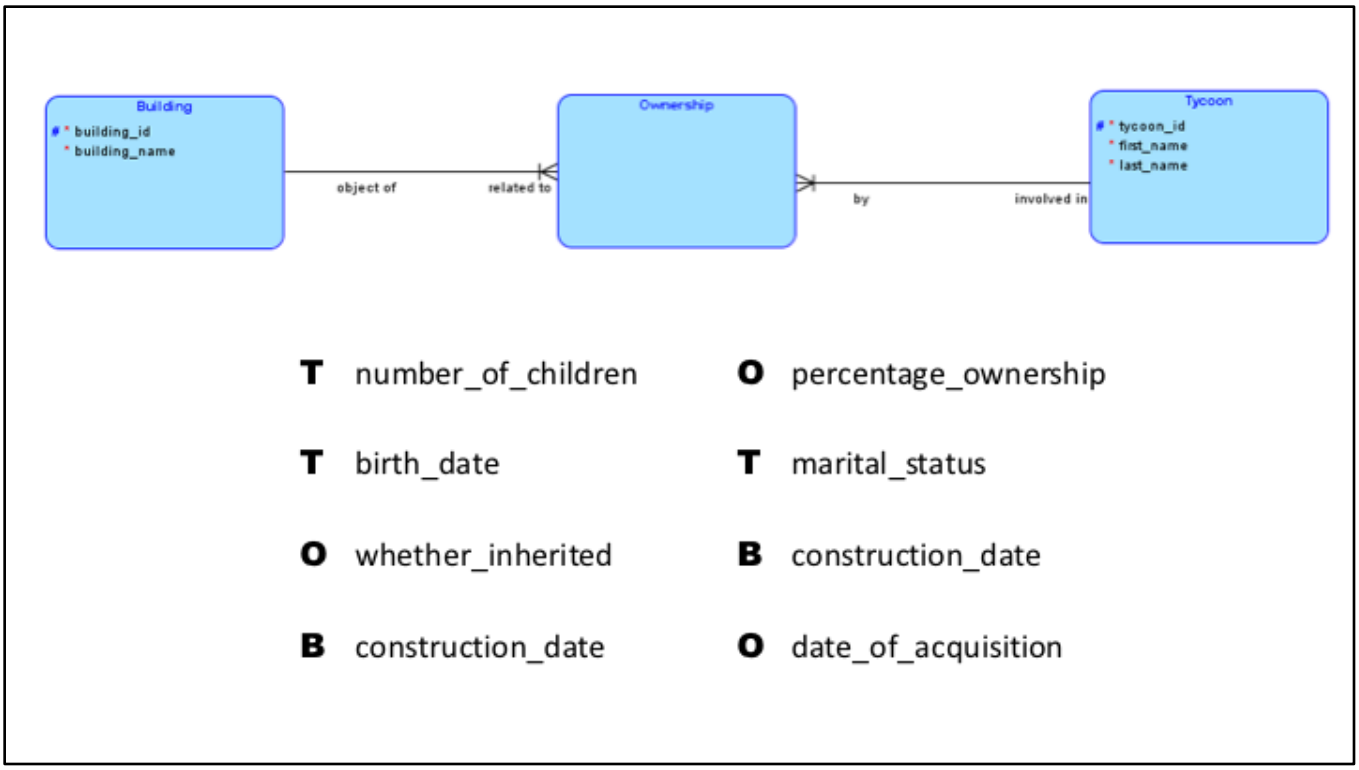
What could be some attributes for Ownership?

On the next slide we consider some additional attributes that play a role in this situation and see where we might assign them.

Before you move on to the next slide, consider the attributes mentioned below and name the entity type to which you will assign each. In other words, which entity type most naturally owns each of these attributes?

number_of_children
birth_date
whether_inherited
construction_date

percentage_ownership
marital_status
construction_date
date_of_acquisition



We have shown with T, B and O the entity type to which each of these attributes most logically belongs.

Note that those marked with O do not fit into either Tycoon or Building and are attributes of the relationship.

For example, percentage_ownership represnets what percentage of a building a tycoon owns. The very sentence describing it mentions both the entity types Tycoon and Building. We cannot talk about percentage_ownership while mentioning only a tycoon or only a building. It is a property of a tycoon-building combination and thus belongs suqarely on Ownership.

You can reason similarly for whether_inherited and date_of_acquisition.

Sales Orders

- Many products and many orders.
- Each order has some quantity of one or more products
- Each product might be on zero or many orders

In a company we have many products and receive many orders for various products. Each order can mention one or more products and each product can be a part of many orders.

Let us build an ERD for this scenario.

For an example, we go back in time to the Coliseum in Rome and consider an order that Emperor Nero might have placed for wild animals for his gladiatorial entertainment.

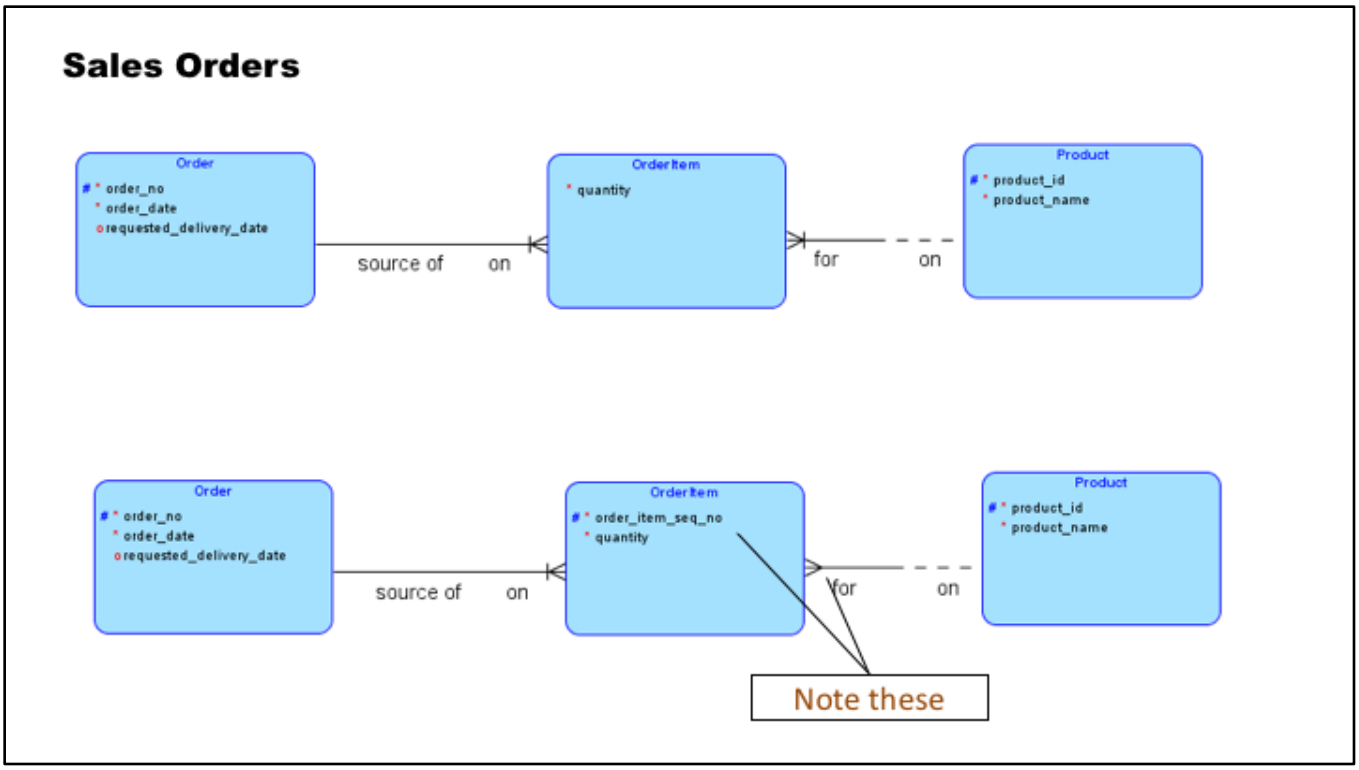
Sales Orders

- Many products and many orders.
- Each order has some quantity of one or more products
- Each product might be on zero or one order

As before we create the obvious m:n relationship between Order and Product. Since every order has to have at least one product, we have drawn a solid line near Product.

The description allows for some products to have not been every part of any order and hence we have shown the line near Product as dashed.

Of course we need an associative entity type to deal with the m:n relationship.



The first diagram shows the usual way to capture the relationship. We have called the new entity type as OrderItem as is traditionally the case. Each row represents an line on an order. Some times people call this entity type LineItem, but I chose OrderItem because line items appear in many other situations and the name seems to me a little too general. OrderItem seems much more specific. (In fact we might get even more specific by calling it SalesOrderItem because we might also have PurchaseOrderItem when we look at Purchase Orders.)

Quantity clearly belongs to OrderItem and the primary key for OrderItem can be the combination of order_id and product_id.

Sometimes however, the same product could multiple time in an order and the above combination might not be unique even within an order. In these cases, we introduce a new attribute order_item_seq_no and use the combination of order_id and order_item_seq_no as the key. Note that the second ERD shows key migration only from Order and not from Product.