3 Examples of Database Normalisation

Three different examples of the database normalisation process, from start to finish

Ben Brumm

www.databasestar.com

What Is This Guide?

Welcome to the 3 Examples of Database Normalisation guide!

So, you've read my articles on normalisation, and have seen the three normal forms.

But you wanted to know more.

There was one example in that article, which explained what the normal forms were and showed the example in each normal form.

But it's always good to see more examples, and seeing them more in-depth can help you understand how the process works, and how it can apply to your situation.

So, this guide contains three different examples of designing a database, across three different domains:

- Bank
- Shopping website
- Airline reservations

Each of these examples explains what the domain is, outlines the requirements in a few sentences, follows the process through each of the normal forms to third normal form, and finishes with some example queries on this database to show different information

So let's get into the examples!

Example 1 - Bank

This first example is a database design for a bank system to capture bank accounts and transactions.

Requirements

The requirements and scope for this database are:

> The database should store customers, the accounts they hold, and the transactions between these accounts.
>
> Accounts can be one of several different types, such as a credit card, transaction account, savings account, or loan. It can also be an active or inactive account. Accounts are identified by an account number, and should store information about when they are opened, and when they are closed. Transactions will have a from and to account, a date and time it occurred, and can store a description.
>
> Customer information such as first name, last name, date of birth, and the date the customer signed up to the bank should be captured. A customer can open many accounts, and an account can be related to one or many customers.

Entities

First, we need to work out what the entities are, which will form the tables in our database.

To do this, read through the requirements and list any nouns (things).

Reading the list above, we can find:

- Customer
- Account
- Transaction

What about all of the other nouns in the requirements, such as type, description, and dates? Well, these are all mentioned in the context of something else, such as "Accounts should store information about open dates". They are probably not entities on their own, but this process could mean that they end up being entities.

Now, with these three entities, or tables, we need to work out what needs to be stored for each of them.

Fields

To work out what fields need to exist for each table, read through the requirements again and look for anything that needs to be stored for something else.

We can put it next to the table name.

- customer: first name, last name, date of birth, signup date, accounts
- account: account type, status, account number, date opened, date closed, customers
- transaction: transaction date time, from account, to account, description

Now, we can see what information needs to be captured for each entity. I've left the information in a plain and readable form (as opposed to adding underscores to make it single words).

How can we improve this database? We start with first normal form.

First Normal Form

First normal form, as mentioned in my Step by Step Guide to Database Normalisation, means:

Each set of columns must uniquely identify a row.

Let's take a look at the columns in each of our tables to see if the columns are unique.

customer: first name, last name, date of birth, signup date, accounts

Does this combination mean that the row is unique? Could a customer with the same name, date of birth, and signup date, have the same accounts? Probably not.

Another question to ask is if having a unique ID column makes sense. In a lot of cases, adding an ID column isn't needed.

But in this case, it does. Should a customer have a customer ID? I think it should. It's reasonable to assume that customers are identified by a customer ID. This could be used in letters or emails to the customer, or perhaps when logging on to their online banking. All of this is out of scope, but it's a good assumption to make.

So, we can add an ID to this table.

This new ID is used to uniquely identify the row. So let's make it the primary key (which I've indicated by underlining the column).

customer: <u>customer_id</u>, first_name, last_name, date_of_birth, signup_date, accounts

I've also changed the names of the tables and columns to remove spaces and add underscores, so they can match any database tables we create.

Let's look at the next table.

account: account type, status, account number, date opened, date closed, customers

Does the combination of these rows mean it is unique?

Yes, I think it does. The requirements say that an account is identified by an account number, so two rows can't exist with the same account number.

To make this clearer, we can indicate the account number as the primary key, and move it to the first spot on the list.

account: <u>account_number</u>, account_type, status, date_opened, date_closed, customers

Let's look at the last table.

transaction: transaction date time, from account, to account, description

Does a combination of these columns ensure it is unique? Could a transaction occur on the same date and time, between the same two accounts, with the same description?

Yes, it could. It's not likely, but it's possible.

Should we create an ID column for this?

Yes, I think we should. There is no existing column or set of columns that can be used to uniquely identify the row. So let's add one.

transaction: <u>transaction_id</u>, transaction_datetime, from_account, to_account, description

Result

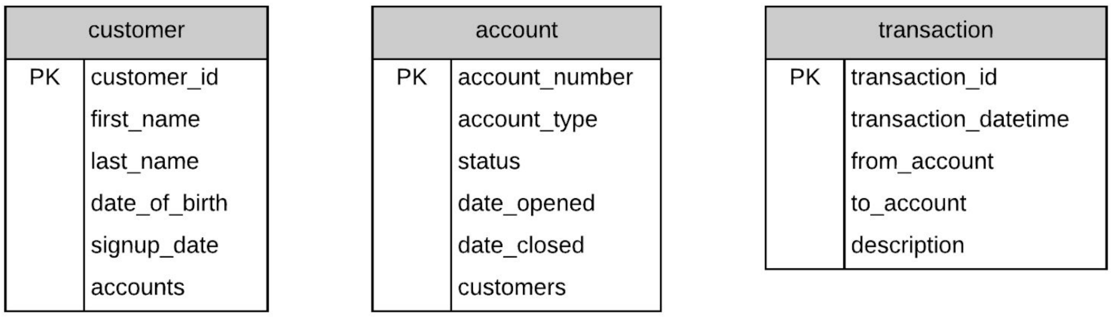
Here are our tables now, which are in first normal form:

customer: <u>customer_id</u>, first_name, last_name, date_of_birth, signup_date, accounts

account: <u>account_number</u>, account_type, status, date_opened, date_closed, customers

transaction: <u>transaction_id</u>, transaction_datetime, from_account, to_account, description

Our diagram looks like this:



Second Normal Form

We have our initial set of tables for the bank database, which is in first normal form.

How do we get it to second normal form?

Well, as mentioned in the article, second normal form is:

1. The tables are in first normal form.
2. All of the non-key attributes are dependent on the primary key.

“All of the non-key attributes are dependent on the primary key” means that all fields that aren’t the primary key are linked to or dependent on the value of the primary key.

Another way to phrase that is to say, “does this column describe what the record is?”

Let’s look at each of our tables.

customer: <u>customer_id</u>, first_name, last_name, date_of_birth, signup_date, accounts

Are each of these fields dependent on the customer ID value?

- first_name: Yes, the ID determines the first name. This field helps to describe the customer.
- last_name: Yes, the ID also determines the last name. This field helps to describe the customer.
- date_of_birth: Yes, this is determined by the ID and is also used to describe the customer.
- signup_date: This field is also used to describe the customer.
- accounts: The accounts that are held by the customer is necessary to describe the customer. It’s good to know, but it’s not an attribute of the customer.

So what do we do about the accounts field?

This needs to be split into a separate table.

Our requirements earlier stated this:

> A customer can open many accounts, and an account can be related to one or many customers.

This is a many-to-many relationship.

So, the best way to do this is to create a joining table. This table will hold the customer Id and the account number of all of the customer’s accounts.

customer: <u>customer_id</u>, first_name, last_name, date_of_birth, signup_date customer_account: customer_id, account_number

The customer ID and account number are both foreign keys (they refer to the primary keys of other tables), so I have changed them to italics here.

In this table we store each combination of customer ID and account number for the accounts that each customer has.

How do we prevent duplicates? What's to stop us putting the same data in two records without a primary key?

Nothing - so we need a primary key.

We can either create a new field, or make the existing fields a primary key.

We don't need to create a new field for a primary key in this case. So let's make the two existing fields the primary key.

customer_account: *customer_id*, *account_number*

account: account_number, account_type, status, date_opened, date_closed, customers

Let's look at each of the columns in the account table:

- account_type: Yes, this is used to describe an account.
- status: Yes, this is used to describe an account.
- date_opened: Yes, this is used to describe an account.
- date_closed: Yes, this is used to describe an account.
- customers: Like the accounts table in the customer table, this does not describe an account. It represents which customers are related to this account.

The customers field here is already represented in our new customer_account table, so we can remove it from here.

Our table now looks like this:

account: account_number, account_type, status, date_opened, date_closed

transaction: transaction_id, transaction_datetime, from_account, to_account, description

Let's look at each of these fields:

- transaction_datetime: Yes, this describes a transaction
- from_account: Yes, this describes a transaction
- to_account: Yes, this describes a transaction
- description: Yes, this describes a transaction

The from account and to account need to represent the accounts that the transaction is sent from, and to. So, let's mark them as foreign keys as they link to the account table.

transaction: <u>transaction_id</u>, transaction_datetime, *from_account*, *to_account*, description

Is something missing here?

What about the actual amount of money sent?

It wasn't mentioned in the requirements, but we can make an assumption here. Sometimes we need to make assumptions. If this is part of a school, college, or university assignment, you can check your assumptions with the teacher or lecturer. If it's part of a project at work, you can check them with the team or the business representative.

In any case, it's fairly safe to assume a transaction needs an amount. So let's add that in.

transaction: <u>transaction_id</u>, transaction_datetime, *from_account*, *to_account*, description, amount

Result

Our full database will look like this:

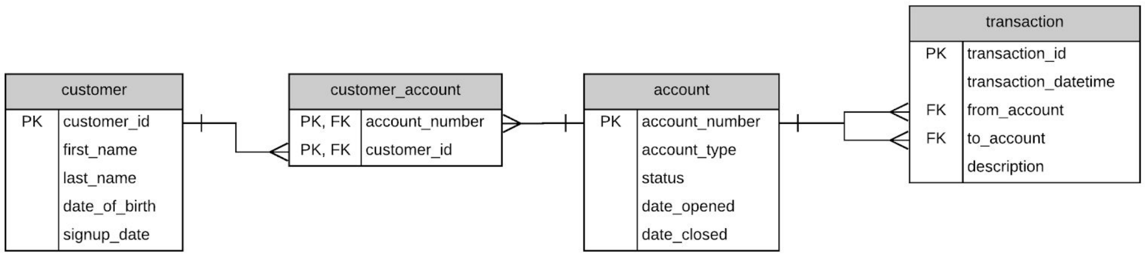
customer: <u>customer_id</u>, first_name, last_name, date_of_birth, signup_date

customer_account: *<u>customer_id</u>*, *<u>account_number</u>*

account: <u>account_number</u>, account_type, status, date_opened, date_closed

transaction: <u>transaction_id</u>, transaction_datetime, *from_account*, *to_account*, description

The diagram looks like this:



This design is now in second normal form! Let's look at third normal form.

Third Normal Form

For a database design to be in third normal form, it must meet this criteria:

1. The database meets the criteria of second normal form.

2. Every non-prime attribute of a table is not transitively dependent on the primary key of the table.

The second point means that **every attribute in your table that is not a key, is not dependent on other attributes in the table**.

Let's take a look at each of our tables.

customer: customer_id, first_name, last_name, date_of_birth, signup_date

- first_name and last_name: Yes, these are dependent on the customer ID, and nothing else.
- date_of_birth: This is dependent on the customer ID and nothing else
- signup_date: This is also dependent on the customer ID and nothing else.

So, the customer table is OK.

customer_account: *customer_id*, *account_number*

The customer_account table is also OK because it only has two key values.

account: account_number, account_type, status, date_opened, date_closed

Let's look at each of these fields.

- account_type: No, this is not dependent on the account number. A type (which could be credit card, transaction, savings) exists independently of an account.
- status: No, this is not dependent on the account number. A status (e.g. active, inactive, or something else) exists independently of an account.
- date_opened: This is also dependent on the account number and nothing else.
- date_closed: This is also dependent on the account number and nothing else.

So, how can we improve the account type?

The possible values of the account type are not dependent on the account number, but the type of that account is.

This means we should create a separate table for all of the different account type values, and link this table to the account table using the primary key.

Our new table would look like this:

account_type: account_type_id, account_type

We can relate it to the account table by using a foreign key to the account_type_id.

account: account_number, *account_type_ID*, status, date_opened, date_closed

We can do the same thing for status. This status value is used to capture if an account is active or inactive. But it could also be used to capture other statuses in the future, such as pending or suspended. So, let's move it to another table in the same way.

account_status: status_id, status_value

Our account table should now refer to the primary key of this table.

account: account_number, *account_type_ID*, *status_id*, date_opened, date_closed

transaction: transaction_id, transaction_datetime, *from_account*, *to_account*, description

Let's look at each of these fields and see if they are dependent on the transaction ID and nothing else:

- transaction_datetime: Yes, it's dependent on the ID and nothing else
- from_account: Yes, it's dependent on the ID and nothing else
- to_account: Yes, it's dependent on the ID and nothing else
- description: Yes, it's dependent on the ID and nothing else

So, the transaction table is OK the way it is.

But, how do we know what amount the transaction is for?

We should add a column for "amount" to the transaction table.

transaction: transaction_id, transaction_datetime, *from_account*, *to_account*, description, amount

Final Database Design

After going through the normal forms for our database design, our design now looks like this:

customer: customer_id, first_name, last_name, date_of_birth, signup_date

customer_account: *customer_id*, *account_number*

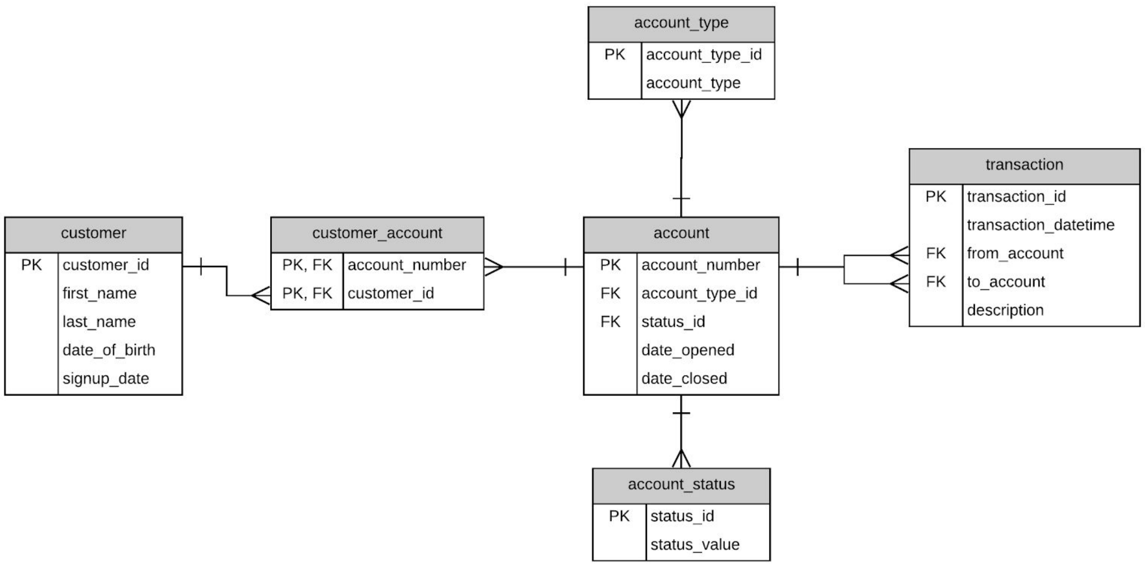
account_type: account_type_id, account_type

account_status: status_id, status_value

account: account_number, *account_type_ID*, *status_id*, date_opened, date_closed

transaction: transaction_id, transaction_datetime, *from_account*, *to_account*, description, amount

Our diagram looks like this.



This database is now in third normal form, and meets the requirements specified earlier.

Sample Queries

One way to validate that our database design is correct is to imagine some queries or questions that might be asked about this data, and explain how you would get this data.

This doesn't have to be written in SQL. I'll explain the answers in plain English or pseudocode.

Q: Find all transactions that came from a particular account

A: Select all of the transaction records where the from_account is the account number you're looking for.

Q: Find all accounts which are a type of credit card.

A: Select all of the accounts where the account_type_id relates to the account_type table, where the account_type is Credit Card.

Q: Find all transactions for a particular account, both from the account and to the account

A: Select all of the transactions where the from account is the account number you're looking for, and union those results with all transactions where the to account is the account number you're looking for.

Q: Find all customer and accounts information, including status and type, for a particular customer

A: Select all customer columns and all account records, matching on the joining table of customer_account, for a particular customer ID. Also select the status_value from the account_status table, and the account_type from the account_type table.

So, it looks like we can answer various queries that might be run against this database.

Example 2 - Shopping Website

This example will store information about products and sales on a shopping website.

Requirements

The requirements and scope for this database are:

> This database should store information about registered users of the shopping website, their addresses, and any orders they place. Users have an email address, a username which serves as their login, and a first and last name.
>
> The website has a range of products, which belong to one of many categories and can be from one of many different brands. These products can be similar and can differ by colour and size, each of which has separate prices.
>
> Users can place orders on the website, and an order can have many different products included in it with different quantities. Individual products in an order can be discounted, and the overall order could be discounted as well.
>
> An order must have a shipping address and a billing address, which can be entered for an order and are saved against the user for use with any future order.
>
> Products each have a supplier that the website gets the products from. These are ordered from the supplier, and like a customer order, these supplier orders can have many different products with different quantities.

Entities

The entities for this database, like with the earlier example, can be found by looking at the nouns in the requirements description. These would be:

- user_account
- customer_order
- product
- supplier_order

There are many other nouns mentioned in the requirements, but at the moment, they seem like attributes or fields of these entities (supplier, brand, address, discount).

We can always expand this list later, so let’s keep going with these entities.

Fields

Just like the earlier example, to work out what fields need to exist for each table, read through the requirements again and look for anything that needs to be stored for something else.

We can put it next to the table name.

- user_account: username, first_name, last_name, email_address, addresses
- customer_order: products, quantities, order_discount, product_discounts, shipping_address, billing_address
- product: product_category, brand, colour, size, supplier
- supplier_order: products, quantities, order_discount, product_discounts

At the moment, the supplier_order and customer_order tables look very similar. The only difference is that there was no mention of the addresses for a supplier order. We could combine them into the one table later in this process, but for now, we can keep them separate.

Let's look at first normal form.

First Normal Form

First normal form, as mentioned earlier, means:

Each set of columns must uniquely identify a row.

Let's take a look at the columns in each of our tables to see if the columns are unique.

user_account: username, first_name, last_name, email_address, addresses

Does a combination of these columns uniquely identify a row?

I think it does. A username is assumed to be unique as it is used for logging in to the system.

We can make that field the primary key.

user_account: <u>username</u>, first_name, last_name, email_address, addresses

customer_order: products, quantities, order_discount, product_discounts, shipping_address, billing_address

Does a combination of these columns uniquely identify a row?

Not necessarily. An order could have the same products, in the same quantities, with the same discounts and to the same address, and be a different order.

So, we should add an ID to this table to serve as the primary key and a way to identify the order.

customer_order: order_id, products, quantities, order discount, product discounts, shipping address, billing address

product: product_category, brand, colour, size, supplier

Does a combination of these fields make a product unique?

No - a product could have the same category, brand, colour, size, and supplier as another product and would not be a duplicate.

We should add an ID.

We should also add a name. A name of a product is not mentioned in the list of requirements, but we can make an assumption that one needs to be added. We should check this assumption with whoever gave us the requirements, but for now we can add it as a field.

So here's the table with both the ID and the name:

product: product_id, product_name, product_category, brand, colour, size, supplier

supplier_order: products, quantities, order_discount, product_discounts

Does a combination of these columns uniquely identify a row?

No - like the customer order table, there could be two rows with the same products, quantities, and discounts.

So, we should add an ID to serve as the primary key.

supplier_order: order_id, products, quantities, order_discount, product_discounts

It can be called order_id, just like the other table. There's no rules to say that the field name needs to be unique in the database. When we query these tables in SQL, we know which table the field comes from.

Result

So, our database in first normal form looks like this:

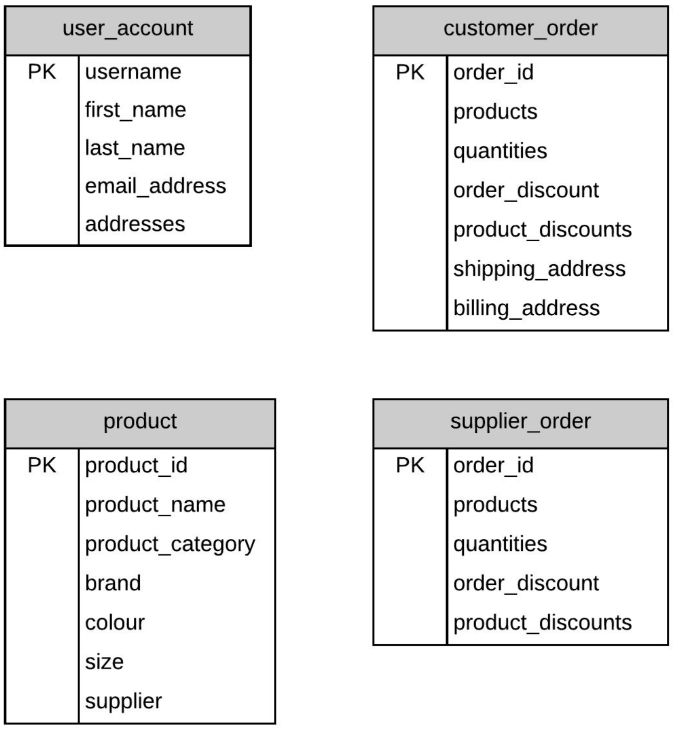
user_account: username, first_name, last_name, email_address, addresses

customer_order: order_id, products, quantities, order_discount, product_discounts, shipping_address, billing_address

product: product_id, product_name, product_category, brand, colour, size, supplier

supplier_order: order_id, products, quantities, order_discount, product_discounts

Our diagram looks like this:



Second Normal Form

We have our initial set of tables for the shopping website database, which is in first normal form.

As mentioned above, second normal form is:

1. The tables are in first normal form.
2. All of the non-key attributes are dependent on the primary key.

"All of the non-key attributes are dependent on the primary key" means that all fields that aren't the primary key are linked to or dependent on the value of the primary key.

Another way to phrase that is to say, "does this column describe what the record is?"

Let's look at each of our tables.

user_account: <u>username</u>, first_name, last_name, email_address, addresses

Are each of these fields dependent on the username value?

- first_name: Yes, and this column helps to describe what the user is.
- last_name: Yes, and this column helps to describe what the user is.

- email_address: Yes, and this column helps to describe what the user is.
- addresses: No - and here's why.

An address doesn't describe what a user is, and isn't dependent on a user.

Generally, some rules apply to addresses:

- People can be related to multiple addresses at a time (home, work, postal)
- People change their relationship to an address (e.g. they move out), rather than an address itself changing (123 Main St is always 123 Main St)
- Addresses can apply to people or to companies

I've mentioned some of these rules in my SQL Best Practices guide, and the suggested design of an address table is what I recommend here as well.

What does that mean for this database?

I would suggest creating a separate table for addresses, and relating them to a user.

This design should work if you want to have a simple setup (one address per user) or for a more complicated setup (multiple addresses over time). But the key here is that it's flexible, so if the software needs change, the database can handle it.

So, here's our new address table and our updated user table:

address: <u>address_id</u>, address

user_account: <u>username</u>, first_name, last_name, email_address, *address_ids*

The address field in the address table is expected to hold everything about an address. This can be split up - but we'll do that in third normal form!

Now, this means a user can have multiple addresses. We can't store multiple IDs in a single column.

An address can relate to multiple users, and a user can relate to multiple addresses.

We know what that means…

Time to create a joining table!

Create a new table to hold the IDs of both, and relate it to each of the other two tables.

address: <u>address_id</u>, address

user_address: <u>address_id</u>, <u>username</u>

user_account: <u>username</u>, first_name, last_name, email_address

We've set both columns in the user_address table as the primary key, for now.

Let's move on to the next table.

customer_order: <u>order_id</u>, products, quantities, order_discount, product_discounts, shipping_address, billing_address

Do each of these columns describe a customer's order?

- products: Yes, they do. But there can be multiple products on an order, which we'll address in a moment.
- quantities: Not really. The quantity is related to a product on an order, not the order itself.
- order_discount: Yes, it describes the order.
- product_discounts: No, this is to record discounts on particular products on an order.
- shipping_address: Yes, this can describe an order.
- billing_address: Yes, this can describe an order.

So, we have the concept of an order with multiple products, each of which has a quantity and an order discount.

This is a common design when dealing with order systems, and it's often implemented as an "order-order line" concept.

This means:

- An order has many order_lines
- Each order line has one product, quantity, and a discount

Here's what our tables would look like:

customer_order: order_id, order_discount, shipping_address, billing_address

customer_order_line: order_line_id, *product_id*, qty, discount, *order_id*

Now, the order_line table stores the product, quantity, and discount. It's related to the order_id in the customer_order table.

The products have been stored as product_ids as they relate to the product table.

What about price?

We could treat price two ways:

- Keep the price in the product table, which means to find the price of an order we need to look at the product table.
- Add the price to the order_line table, which means everything is on the order_line.

There are pros and cons to each way, but I would prefer having the price on the order_line. This is because past orders are not affected if a price changes.

So, our updated tables would be:

customer_order: order_id, order_discount, shipping_address, billing_address

customer_order_line: order_line_id, *product_id*, qty, discount, price, *order_id*

But… how can we tell which customer this order relates to?

We can add the primary key of the customer (the user table) into this table as a foreign key:

customer_order: <u>order_id</u>, order_discount, shipping_address, billing_address, *username*

customer_order_line: <u>order_line_id</u>, *product_id*, qty, discount, price, *order_id*

product: <u>product_id</u>, product_name, product_category, brand, colour, size, supplier

Do each of these columns describe a product?

- Product_name: Yes, this is how a product is referred to
- Product_category: Yes, this describes a product.
- Brand: Yes, this describes a product.
- Colour: Yes
- Size: Yes
- Supplier: Yes

So, our table meets second normal form as it is.

supplier_order: <u>order_id</u>, products, quantities, order_discount, product_discounts

Do each of these columns describe a supplier order?

- products: Yes, but an order has many products.
- quantities: No, because the quantity applies to a product on an order, not the whole order.
- order_discount: Yes
- product_discount: No, these apply to the products on the order.

We should take the same approach here as we did with the customer_order (split it into order and order_line):

supplier_order: <u>order_id</u>, order_discount

supplier_order_line: <u>order_line_id</u>, *product_id*, qty, discount, price, *order_id*

Now, how can we tell which supplier the order was made from? We need to store supplier information in the supplier_order table:

supplier_order: <u>order_id</u>, order_discount, supplier

Result

Here's what our set of tables now look like after second normal form has been applied:

address: address_id, address

user_address: *address_id*, *username*

user_account: username, first_name, last_name, email_address

customer_order: order_id, order_discount, shipping_address, billing_address, *username*

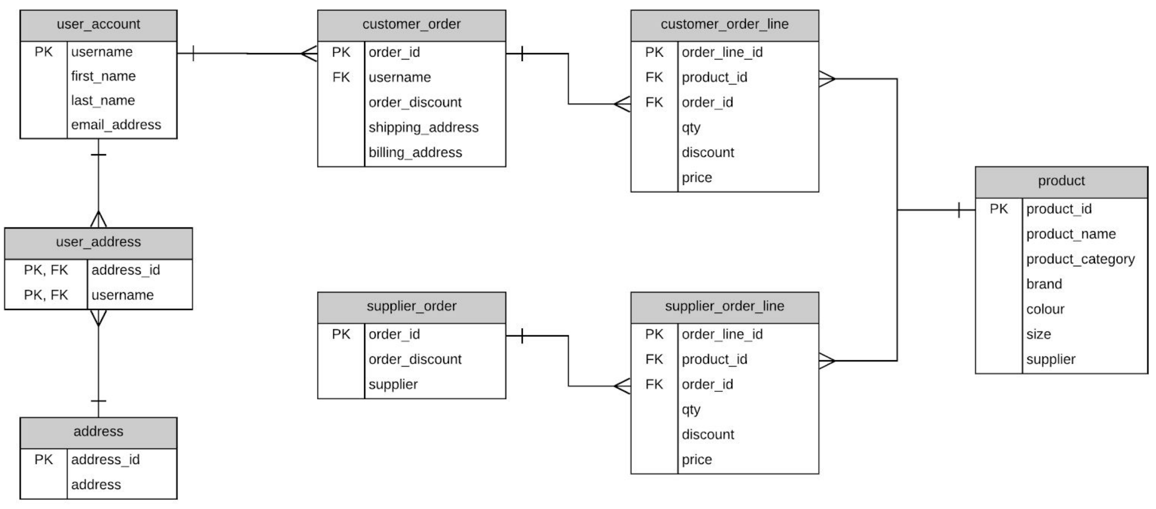
customer_order_line: order_line_id, *product_id*, qty, discount, price, *order_id*

product: product_id, product_name, product_category, brand, colour, size, supplier

supplier_order: order_id, order_discount, supplier

supplier_order_line: order_line_id, *product_id*, qty, discount, price, *order_id*

Our diagram looks like this:



We may be able to combine the order tables into one table. The only differences are that the customer_order table has a few extra columns:

- shipping_address
- billing_address
- username

We could combine these two tables, but to ensure referential integrity we would need to have a "user" record that was used to place supplier orders, and a way to record the shipping and billing address of a supplier order (which could be the address of the business). We may also want to add an order_type to define if it is a customer order or supplier order.

Or, we can keep them separate. There may be other fields that apply only to supplier orders and not customer orders (e.g. credit terms, invoice numbers).

These are worth checking with whoever gave you the requirements. It's a good idea to simplify the design if you can, but don't try to force similar entities together if they are too different.

For this example, **we'll keep them separate**. But it's an example of where you have two choices for a design decision, and either way could work.

Third Normal Form

For a database design to be in third normal form, it must meet this criteria:

1. The database meets the criteria of second normal form.
2. Every non-prime attribute of a table is not transitively dependent on the primary key of the table.

The second point means that **every attribute in your table that is not a key, is not dependent on other attributes in the table**.

Let's take a look at each of our tables.

address: <u>address_id</u>, address

Does the address depend on any other fields?

No, it doesn't.

But, the address could (and should) be broken down into the parts of an address. I recommend this so that addresses can be analysed and reported on, displayed correctly, and used better in search.

So, the address field could be split up into:

- house_unit_no
- address_line1
- address_line2
- city
- region
- postal_code
- country

Our table would look like this:

address: <u>address_id</u>, house_unit_no, address_line1, address_line2, city, region, postal_code, country

user_address: <u>address_id</u>, <u>username</u>

There are no fields in this table that aren't part of the primary key.

user_account: <u>username</u>, first_name, last_name, email_address

Do these fields depend on the username?

- first_name: Yes, the name depends on the username.
- last_name: Yes, same as the first_name.
- email_address: Yes, this depends on the username.

customer_order: <u>order_id</u>, order_discount, shipping_address, billing_address, *username*

Do these fields depend on the order_id?

- order_discount: Yes, this is dependent on the order_id.
- shipping_address: Yes, this is dependent on the order_id.
- billing_address: Yes, this is.

I've left off the username from this list as it's a foreign key.

Is there anything else we may want to capture as part of this table?

Maybe the date the order was places. We can assume that this needs to be captured (and we should check that with whoever gave us the requirements), but we can add that as a column. It also is dependent on the order_id:

customer_order: <u>order_id</u>, order_discount, shipping_address, billing_address, *username*, order_date

customer_order_line: <u>order_line_id</u>, *product_id*, qty, discount, price, *order_id*

Are these fields dependent on the order_line_id?

- qty: Yes, it's related to each order line.
- discount: Yes
- price: Yes

product: <u>product_id</u>, product_name, product_category, brand, colour, size, supplier

Are these fields dependent on the product_id?

- product_name: Yes, this is determined by the product_id.

- product_category: No, because a category can exist without a product.
- brand: No, because a brand can exist without a product.
- colour: No, because a colour can exist without a product.
- size: Yes and no. It is determined by the ID, as it can vary for each product type, but a size could exist without a product.
- supplier: No, a supplier can exist without a product.

There's quite a few fields here that we've said "no" to. Let's look at them.

Product_category can exist without a product. We want to avoid using the values in the product table as a list of categories, as there could be data entry errors, and if products are removed then the category could be removed by mistake. This is called a "delete anomaly".

To improve this, we can create a separate table to store the main list of product categories, and relate the products to it:

product_category: category_id, category_name

product: product_id, product_name, *product_category_id*, brand, colour, size, supplier

Brand will follow the same pattern. A brand can exist without a product, so we should split that into a separate table:

brand: brand_id, brand_name

product: product_id, product_name, *product_category_id*, *brand_id*, colour, size, supplier

Colours can also exist without products. This kind of attribute (as well as the Size attribute we'll look at shortly) is something that may not have any extra information.

The Brand and Product Category tables may have extra attributes (such as parent product categories, or brands that have a type of product they specialise in), which makes more sense for them to be split into separate tables.

But, with the Colour table, we will likely only store the colour name. It can still be a good idea to split this out into another table, as it will likely be easier for the applications to filter on colour, and select different colours. It also enforces a standard way of entering data (e.g. "Black" is only entered once).

So, let's create a separate table.

colour: colour_id, colour_name

product: product_id, product_name, *product_category_id*, *brand_id*, *colour_id*, size, supplier

I've called it colour with a U because I'm from Australia. You can of course call it color without a U if that's how you spell it :)

Size is an interesting one. It can exist without a product, but it's also an attribute that is likely specific to a product or set of products (e.g. shoes have different sizes to shirts, women's pants, men's pants, and other clothing).

But most sizes come from a range of values, such as numbers or S/M/L.

So, we can split it into a separate table.

size: <u>size_id</u>, size_name

product: <u>product_id</u>, product_name, *product_category_id*, *brand_id*, *colour_id*, *size_id*, supplier

Finally, a supplier can also exist outside of a product, so we should create a separate table for that and link it to the product:

supplier: <u>supplier_id</u>, supplier_name

product: <u>product_id</u>, product_name, *product_category_id*, *brand_id*, *colour_id*, *size_id*, *supplier_id*

supplier_order: <u>order_id</u>, order_discount, supplier

The order_discount field is dependent on the ID.

However, the supplier field can exist without the supplier order. We have just created a supplier table, so we should relate this order to that table using the id:

supplier_order: <u>order_id</u>, order_discount, *supplier_id*

Just like the customer_order table, we could add an order_date column here as well.

supplier_order: <u>order_id</u>, order_discount, *supplier_id*, order_date

supplier_order_line: <u>order_line_id</u>, product_id, qty, discount, price, *order_id*

Do these fields depend on the order_line_id?

- qty: Yes, it's related to each order line.
- discount: Yes
- price: Yes

Now we've looked at all of our tables and transformed them to third normal form.

Final Database Design

Let's take a look at our final database design now we've achieved third normal form:

address: <u>address_id</u>, house_unit_no, address_line1, address_line2, city, region, postal_code, country

user_address: <u>address_id</u>, <u>username</u>

user_account: <u>username</u>, first_name, last_name, email_address

customer_order: <u>order_id</u>, order_discount, shipping_address, billing_address, *username*, order_date customer_order_line: order_line_id, *product_id*, qty, discount, price, *order_id*

product_category: category_id, category_name

brand: brand_id, brand_name

colour: colour_id, colour_name

size: size_id, size_name

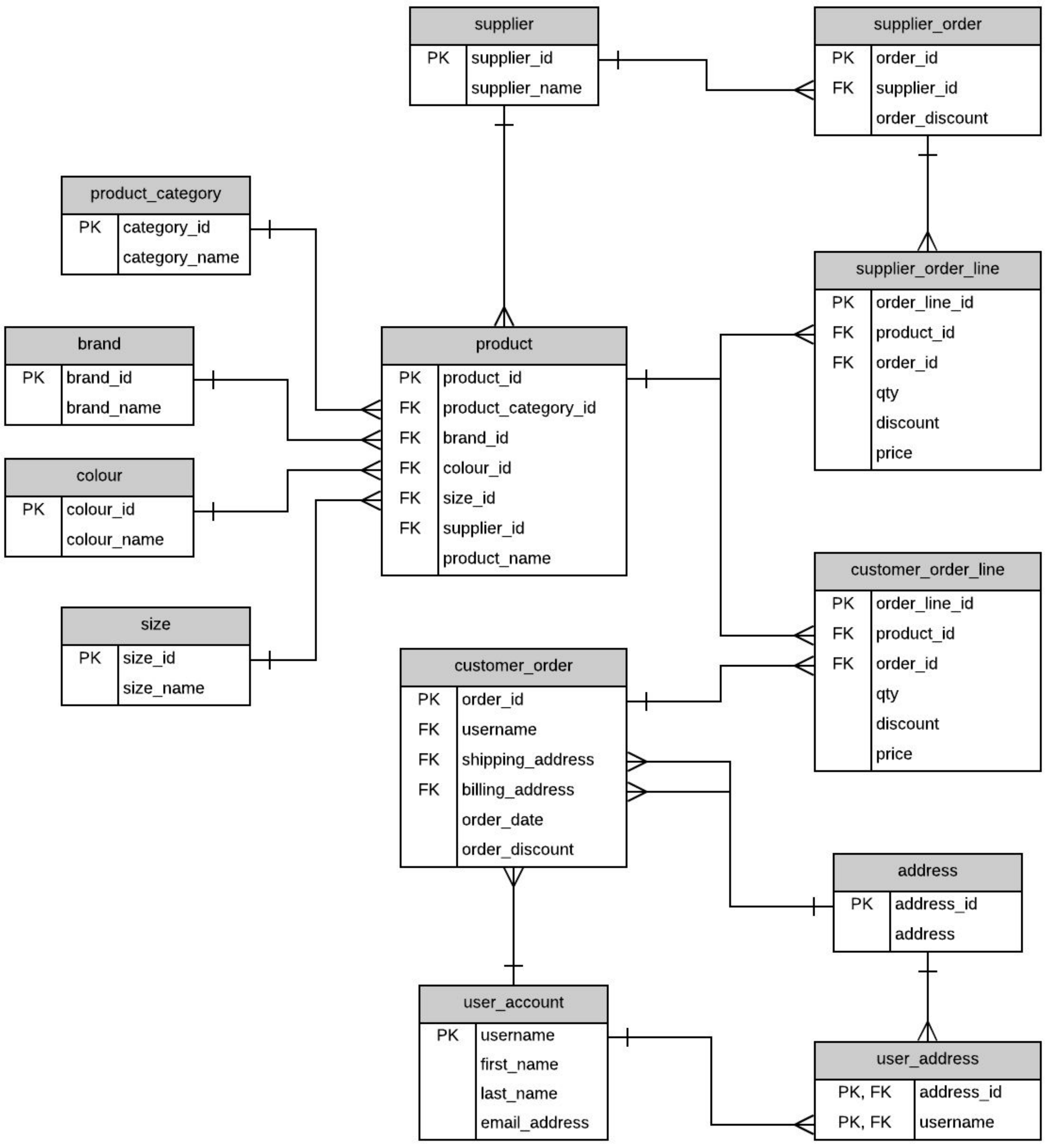
supplier: supplier_id, supplier_name

product: product_id, product_name, *product_category_id*, *brand_id*, *colour_id*, *size_id*, *supplier_id*

supplier_order: order_id, order_discount, *supplier_id*, order_date

supplier_order_line: order_line_id, *product_id*, qty, discount, price, *order_id*

Our diagram looks like this:



Sample Queries

To validate that we have a database design that works, we can look at some sample queries that we might want to run on this database.

Q: Find all orders and products and order details for a customer.

Select the orders, order lines, and related products for all orders that match a specific username.

Q: Find all addresses that a user is linked to.

Select all address fields from the address table, where the address ID matches a record in the user_address joining table for the specific username.

Q: Find all products and all of their attributes (category, brand, colour, supplier, size)

Select all product data, and the names for the category, brand, colour, supplier, and size, from each of those tables, using the related IDs in the product table.

Example 3 - Airline Reservations

This final example is probably the most complicated of the examples we've looked at in this guide.

It describes an airline reservation system.

The reason I think it's complicated is because there are many different rules that specify how things can be related and what the restrictions are, which impacts the entities we end up with.

Let's take a look at the requirements.

Requirements

The requirements for this database are:

> This database will need to store information about an airline reservation system. The purpose is to record customer bookings for flights.
>
> A flight is comprised of one or many legs, where a leg has an arrival airport and a destination airport (e.g. LAX airport to JFK airport), as well as a date and time of arrival and departure. It has a flight code and is run by an airline.
>
> Many airports around the world will be stored in this system, and a country and even a city can have multiple airports (e.g. London or New York City).
>
> A leg of a flight is performed on only one aircraft, and different legs of a flight could change aircraft. An aircraft can be a particular aircraft model, which is manufactured by a company. An aircraft is owned by an airline, and an airline can own many aircraft of the same model, or of different models.
>
> An aircraft model has a configuration of seats, which is the same for all aircrafts of that model. These seats are of several different classes, such as first class, business class, and economy.
>
> A customer books a flight, which is comprised of one or more legs, through a single airline. The booking contains the price the customer paid for each leg, as well as the class of seat they paid for. The system will store the customer's name, passport number, email address, phone number, and the country they are from.

That's quite a long list! We'll come back to these as we design our database.

Entities

First, let's start with identifying the entities. We do this by looking at all of the nouns in the requirements:

- flight
- leg
- airport
- aircraft
- manufacturer
- airline
- seat_class
- customer
- booking

This is a longer list than we've seen in the other examples.

However, the process we follow is the same.

Fields

Let's look for the fields we can use for each of these entities.

- flight: flight_code, airline
- leg: arrival_airport, departure_airport, arrival_time, departure_time, aircraft
- airport: airport_name, country, city
- aircraft: model, manufacturer, owned_by_airline
- manufacturer
- airline
- seat_class: class_name
- customer: first_name, last_name, passport_number, email_address, phone_number, country
- booking: booking_legs, price_for_each_leg, airline, seat_class, customer

Some of these entities don't have any fields. That's OK for now. We can remove them as we go through the normalisation process if we don't need them.

First Normal Form

First normal form, as mentioned earlier, means:

Each set of columns must uniquely identify a row.

Let's take a look at the columns in each of our tables to see if the columns are unique.

flight: flight_code, airline

Does the combination of these fields mean the row is unique?

Yes, an airline and a flight code would be unique.

Could the flight code change? No. We can assume it does not, but would need to check.

If we assume it doesn't, then it can be the primary key.

flight: <u>flight_code</u>, airline

leg: arrival_airport, departure_airport, arrival_time, departure_time, aircraft

Does the combination of these fields mean the row is unique?

No - there could be two aircraft departing the same airport at the same time, and arriving at the same airport at the same time.

So, let's add in a primary key field.

leg: <u>leg_id</u>, arrival_airport, departure_airport, arrival_time, departure_time, aircraft

airport: airport_name, country, city

Does a combination of these fields mean the row is unique?

Yes, if we add in the unique code given to the airport. Let's add in that code now.

airport: <u>airport_code</u>, airport_name, country, city

I've called it airport_code instead of airport_id as I feel that code is a more descriptive term for this. Airport name is kept because the name of the airport could be useful other than the code (e.g. a code of JFK has a name of "John F Kennedy Airport").

aircraft: model, manufacturer, owned_by_airline

Does a combination of these fields mean the row is unique?

I don't think so.

Here are the requirements around aircrafts and airlines and models:

> An aircraft can be a particular aircraft model, which is manufactured by a company. An aircraft is owned by an airline, and an airline can own many aircraft of the same model, or of different models.

The key part here is that “an airline can own many aircraft of the same model”.

So, if we have two rows with the same model, manufacturer, and owned_by_airline, they are not unique.

What are we really trying to store here?

- An aircraft has a model
- An airline owns many aircraft of many different models

So, perhaps our tables need to be something like this:

aircraft: <u>aircraft_model</u>, owned_by_airline

aircraft_model: manufacturer, model

This stores information about the particular model in the aircraft_model table, as well as many instances of that aircraft owned by different airlines.

Let’s add in some primary key fields:

aircraft: <u>aircraft_id</u>, aircraft_model, owned_by_airline

aircraft_model: <u>model_id</u>, manufacturer, model

manufacturer

The manufacturer table doesn’t have any fields. What do we want to capture about a manufacturer? At least a name.

manufacturer: manufacturer_name

Is the name unique? Probably.

Can the name change? It might.

Generally, if a value can change, it should not be the primary key. This is because it can relate to other tables as a foreign key.

So, let’s add in an ID.

manufacturer: <u>manufacturer_id</u>, manufacturer_name

airline

Like the manufacturer table, it has no fields. And like manufacturer, we may want to store the name of the airline, at least.

airline: airline_name

Can the airline_name change? Maybe.

Let's add a primary key ID field, just like we did with manufacturer.

airline: <u>airline_id</u>, airline_name

seat_class: class_name

Is the class_name unique? Yes, it probably is.

Can it change? It could (e.g. rename Economy to Coach).

So, let's add an ID to use as the primary key.

seat_class: <u>seat_class_id</u>, class_name

customer: first_name, last_name, passport_number, email_address, phone_number, country

Does a combination of these fields mean the row is unique?

Yes. A passport number is unique and it is used to identify a person.

However, passport numbers can change. And it's not a good idea to make fields that can change as the primary key.

So, let's create a new ID for the primary key.

customer: <u>customer_id</u>, first_name, last_name, passport_number, email_address, phone_number, country

booking: booking_legs, price_for_each_leg, airline, seat_class, customer

Does a combination of these fields mean the row is unique?

No. A customer could book the same legs for the same price, seats, and airline more than once.

So, let's add in an ID to use as the primary key.

booking: <u>booking_id</u>, booking_legs, price_for_each_leg, airline, seat_class, customer

Result

After going through our first normal form, our database design looks like this:

flight: <u>flight_code</u>, airline

leg: <u>leg_id</u>, arrival_airport, departure_airport, arrival_time, departure_time, aircraft

airport: <u>airport_code</u>, airport_name, country, city

aircraft: <u>aircraft_id</u>, aircraft_model, owned_by_airline aircraft_model: model_id, manufacturer, model

manufacturer: manufacturer_id, manufacturer_name

airline: airline_id, airline_name

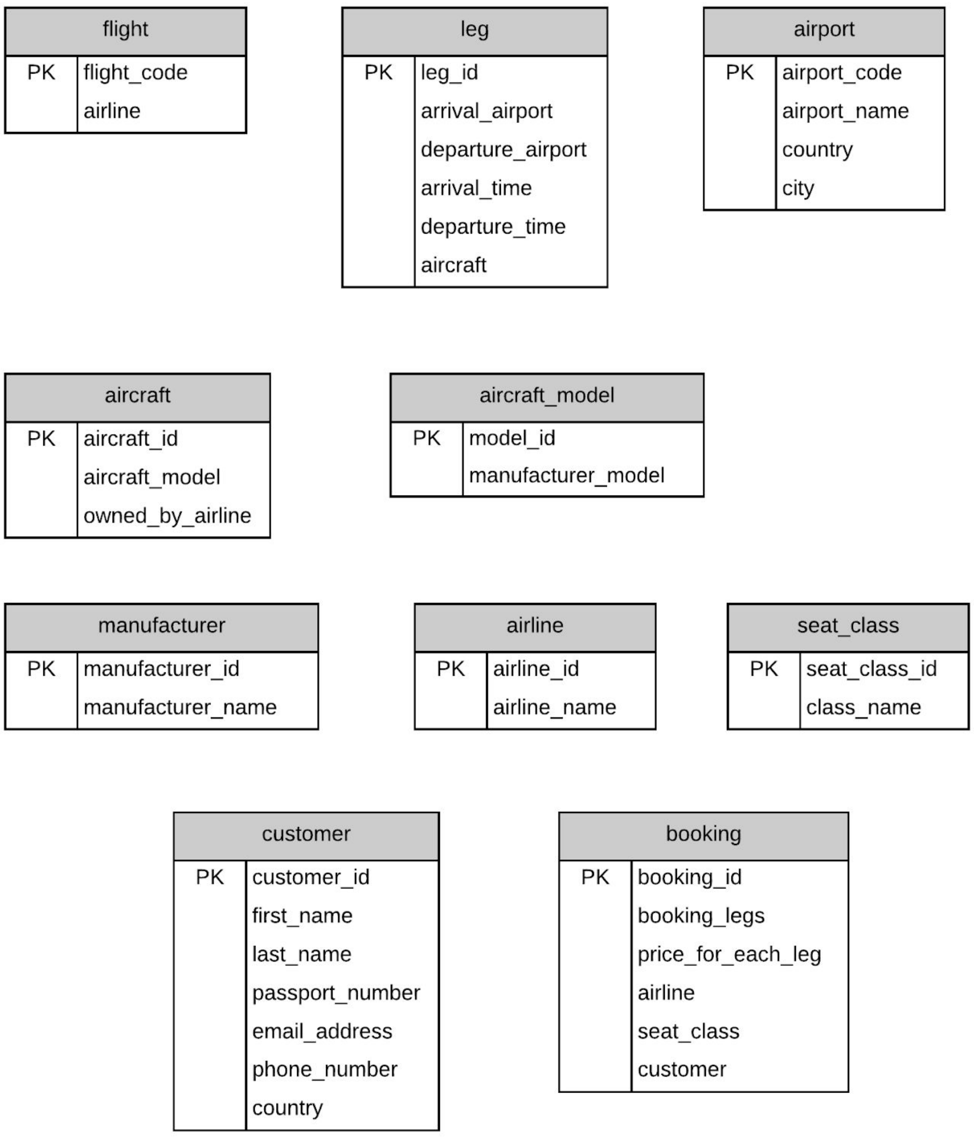
seat_class: seat_class_id, class_name

customer: customer_id, first_name, last_name, passport_number, email_address, phone_number, country

booking: booking_id, booking_legs, price_for_each_leg, airline, seat_class, customer

Each table now has some fields and a primary key.

Our diagram looks like this:



Second Normal Form

We have our initial set of tables for the airline reservation database, which is in first normal form.

As mentioned above, second normal form is:

1. The tables are in first normal form.
2. All of the non-key attributes are dependent on the primary key.

"All of the non-key attributes are dependent on the primary key" means that all fields that aren't the primary key are linked to or dependent on the value of the primary key.

Another way to phrase that is to say, "does this column describe what the record is?"

Let's look at each of our tables.

flight: <u>flight_code</u>, airline

Do each of these columns describe the flight?

- airline: Yes, it helps to describe the flight.

However, the airline field shouldn't store the name of the airline, because that's stored in the airline table. This field should be the airline_id from the airline table, which is used to link the two tables together. I've formatted it in italics to make it clear it's a foreign key.

flight: <u>flight_code</u>, airline_id

leg: leg_id, arrival_airport, departure_airport, arrival_time, departure_time, aircraft

Do each of these columns describe the leg?

- arrival_airport: Yes, it describes where the leg ends.
- departure_airport: Yes, it describes where the leg starts.
- arrival_time: Yes, it describes when it ends.
- departure_time: Yes, it describes when it starts.
- aircraft: Yes, it describes which aircraft is used for the leg.

Now, like the airline_id in the table we just looked at, both the arrival_airport and departure_airport are not the names of the airports. They should store the airport_code from the airport table. Let's update the names of the columns and mark them as foreign keys.

leg: <u>leg_id</u>, *arrival_airport_code*, *departure_airport_code*, arrival_time, departure_time, *aircraft_id*

But, how do we know which leg this flight relates to? We don't. So we need to add a column to relate this to the flight.

leg: leg_id, arrival_airport_code, departure_airport_code, arrival_time, departure_time, *aircraft_id*, *flight_code*

airport: airport_code, airport_name, country, city

Do each of these columns describe the airport?

- airport_name: Yes, this describes the airport.
- Country: Yes, it does.
- City: Yes, it's where the airport is located.

We can leave the table as it is.

aircraft: aircraft_id, aircraft_model, owned_by_airline

Do each of these columns describe the aircraft?

- aircraft_model: Yes, this is the model of the aircraft.
- owned_by_airline: Yes, this is the airline that owns the aircraft.

However, these two fields are foreign keys. The aircraft_model should be the ID from the aircraft_model table, and the owned_by_airline should be the ID from the airline table.

aircraft: aircraft_id, aircraft_model_id, owned_by_airline_id

aircraft_model: model_id, manufacturer, model

Do each of these columns describe the aircraft model?

- manufacturer: Yes, this is the manufacturer, but it should store the ID of the manufacturer table and not the name
- model: Yes, it is the name of the model. We could call it model_name to be more descriptive.

Our table will look like this:

aircraft_model: model_id, manufacturer_id, model_name

manufacturer: manufacturer_id, manufacturer_name

Does the manufacturer_name describe the manufacturer? Yes it does. So there is no change to this table.

airline: airline_id, airline_name

Does the airline_name describe the airline? Yes it does, so no change to the table.

seat_class: <u>seat_class_id</u>, class_name

Does the class_name describe the seat_class? Yes, it does, so no change to the table.

customer: *customer_id*, first_name, last_name, passport_number, email_address, phone_number, country

Do each of these columns describe the customer?

- first_name: Yes, it does.
- last_name: Yes, it does.
- passport_number: Yes, it does.
- email_address: Yes, it does.
- phone_number: Yes, it does.
- country: Yes, it does.

So, there is no change to this table.

booking: <u>booking_id</u>, booking_legs, price_for_each_leg, airline, seat_class, customer

Do each of these fields describe a booking?

- booking_legs: Yes, but there are multiple values stored here.
- price_for_each_leg: Yes, but there are also multiple values stored here.
- airline: Yes, and this should be the ID of the airline.
- seat_class: Yes, but it depends if the class of seat is the same for a booking, or it can change for each leg.
- customer: Yes, it is the customer who made the booking.

There are a few changes to make here.

First, the booking_legs and price_for_each_leg columns store information about each leg that has been booked, not the overall booking.

We don't have a table to store this information, so let's create one. It would need these two fields, as well as a way to refer to the booking.

booking_leg: <u>booking_leg_id</u>, booking_id, leg_id, price

The booking_legs and price_for_each_leg can be removed from the booking table, because they have been moved to the booking_leg table and linked using the booking_id column.

Now, can the seat class change for each leg? Let’s take a look at the requirements:

> A customer books a flight, which is comprised of one or more legs, through a single airline. **The booking contains the price the customer paid for each leg, as well as the class of seat they paid for.**

The requirements, as they are written, are not clear. They could be interpreted either way.

So, in this situation, I would take the more flexible approach. It’s more flexible to store the seat_class on the booking_leg rather than the booking. This means the customer can book different seat classes for each leg of their flight, rather than the same one for their entire booking.

But, let’s think about the requirements a bit more. Do customers pay for a seat class, or do they pay for a seat of a particular class?

There’s a slight difference there. The first option refers to an object called a seat class, and the second refers to an object called a seat.

I would assume (and it’s worth validating with whoever gave you these requirements) that customers book a seat on an aircraft, and the seat has a class. If customers make bookings and we just store the class they paid for, it will be hard to tell how many seats have been booked and how many are still available.

So, let’s update our tables.

booking: <u>booking_id</u>, airline, customer

booking_leg: <u>booking_leg_id</u>, *booking_id*, *leg_id*, *seat_id*, price

We should also change the airline and customer to the ID columns:

booking: <u>booking_id</u>, *airline_id*, *customer_id*

We don’t yet have a seat table. How does it fit in to the database?

A seat belongs to an aircraft. But the seat is actually part of an aircraft model, independent of a flight. And the seat has a class.

So, the seat table could look like this:

aircraft_seat: <u>seat_id</u>, *aircraft_id*, *seat_class_id*

Result

Our database design in second normal form is:

flight: <u>flight_code</u>, *airline_id*

leg: <u>leg_id</u>, *arrival_airport_code*, *departure_airport_code*, arrival_time, departure_time, *aircraft_id*, *flight_code*

airport: <u>airport_code</u>, airport_name, country, city

aircraft: <u>aircraft_id</u>, *aircraft_model_id*, *owned_by_airline_id* aircraft_model: model_id, *manufacturer_id*, model_name

manufacturer: manufacturer_id, manufacturer_name

airline: airline_id, airline_name

aircraft_seat: seat_id, *aircraft_id*, *seat_class_id*

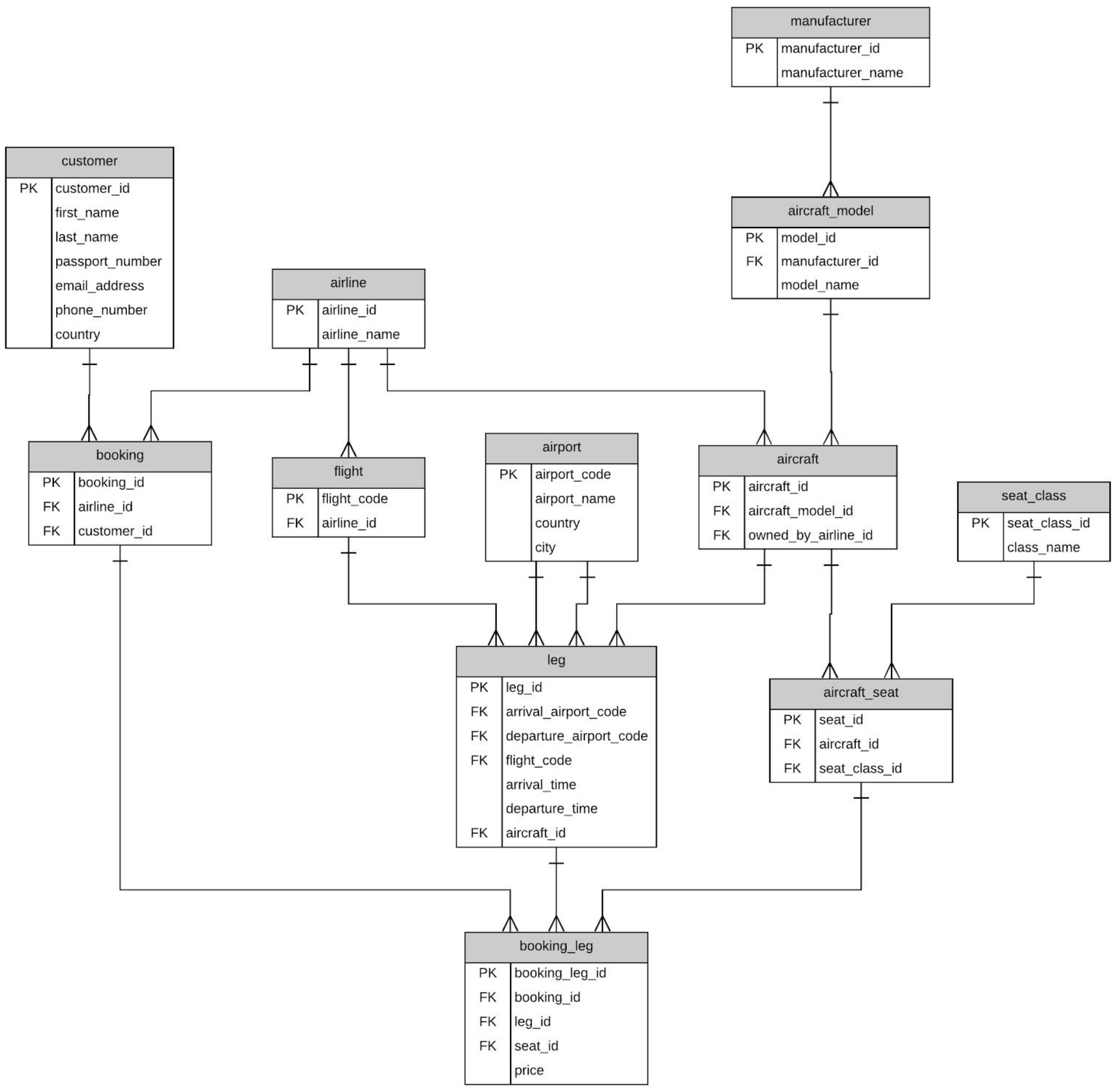
seat_class: seat_class_id, class_name

customer: customer_id, first_name, last_name, passport_number, email_address, phone_number, country

booking: booking_id, *airline_id*, *customer_id*

booking_leg: booking_leg_id, *booking_id*, *leg_id*, *seat_id*, price

Our diagram looks like this:



Third Normal Form

For a database design to be in third normal form, it must meet this criteria:

1. The database meets the criteria of second normal form.
2. Every non-prime attribute of a table is not transitively dependent on the primary key of the table.

The second point means that **every attribute in your table that is not a key, is not dependent on other attributes in the table**.

Let's take a look at each of our tables.

flight: <u>flight_code</u>, *airline_id*

Both of these fields are keys, so there is no change here.

leg: <u>leg_id</u>, *arrival_airport_code*, *departure_airport_code*, arrival_time, departure_time, *aircraft_id*, *flight_code*

Does the departure_time and arrival_time depend on any other fields? No, it doesn't.

All other fields are keys, so this table is unchanged.

airport: <u>airport_code</u>, airport_name, country, city

Do these fields depend on other fields:

- airport_name: No.
- Country: Yes, this depends on city.
- City: No, but the cities exist independently of airports.

There are a couple of changes here.

First, we should store the cities in a separate table and relate them to this table.

city: <u>city_id</u>, city_name

airport: <u>airport_code</u>, airport_name, country, *city_id*

We should also store countries in a separate table, and relate them to the city (as the country belongs to a city, which in turn is related to an airport).

airport: <u>airport_code</u>, airport_name, *city_id*

country: <u>country_id</u>, country_name city: city_id, city_name, *country_id*

This way we can find the airport, city, and country details by linking these three tables.

aircraft: aircraft_id, *aircraft_model_id*, *owned_by_airline_id*

All of these fields are keys, so there are no changes here.

aircraft_model: model_id, *manufacturer_id*, model_name

The model_name is not dependent on anything else.

manufacturer: manufacturer_id, manufacturer_name

The manufacturer name is not dependent on anything else.

airline: airline_id, airline_name

The airline name is not dependent on anything else.

aircraft_seat: seat_id, *aircraft_id*, *seat_class_id*

All of these fields are keys so there is no change here.

seat_class: seat_class_id, class_name

The class_name is not dependent on anything else.

customer: customer_id, first_name, last_name, passport_number, email_address, phone_number, country

Are these fields dependent on other fields?

- first_name: No.
- last_name: No.
- passport_number: No.
- email_address: No.
- phone_number: No.
- country: No.

However, we have a table for country already, and it would be a good idea to relate the customer to the country table, rather than store the name of the country again.

customer: <u>customer_id</u>, first_name, last_name, passport_number, email_address, phone_number, *country_id*

booking: <u>booking_id</u>, *airline_id*, *customer_id*

All of these fields are keys, so no change here.

booking_leg: <u>booking_leg_id</u>, *booking_id*, *leg_id*, *seat_id*, price

Does the price depend on any other fields? No, it doesn't.

Final Database Design

Our final database design, in third normal form, is:

flight: <u>flight_code</u>, *airline_id*

leg: <u>leg_id</u>, arrival_airport_code, departure_airport_code, arrival_time, departure_time, aircraft_id, flight_code

airport: <u>airport_code</u>, airport_name, *city_id*

country: <u>country_id</u>, country_name

city: <u>city_id</u>, city_name, *country_id*

aircraft: <u>aircraft_id</u>, *aircraft_model_id*, *owned_by_airline_id*

aircraft_model: <u>model_id</u>, *manufacturer_id*, model_name

manufacturer: <u>manufacturer_id</u>, manufacturer_name

airline: <u>airline_id</u>, airline_name

aircraft_seat: <u>seat_id</u>, *aircraft_id*, *seat_class_id*

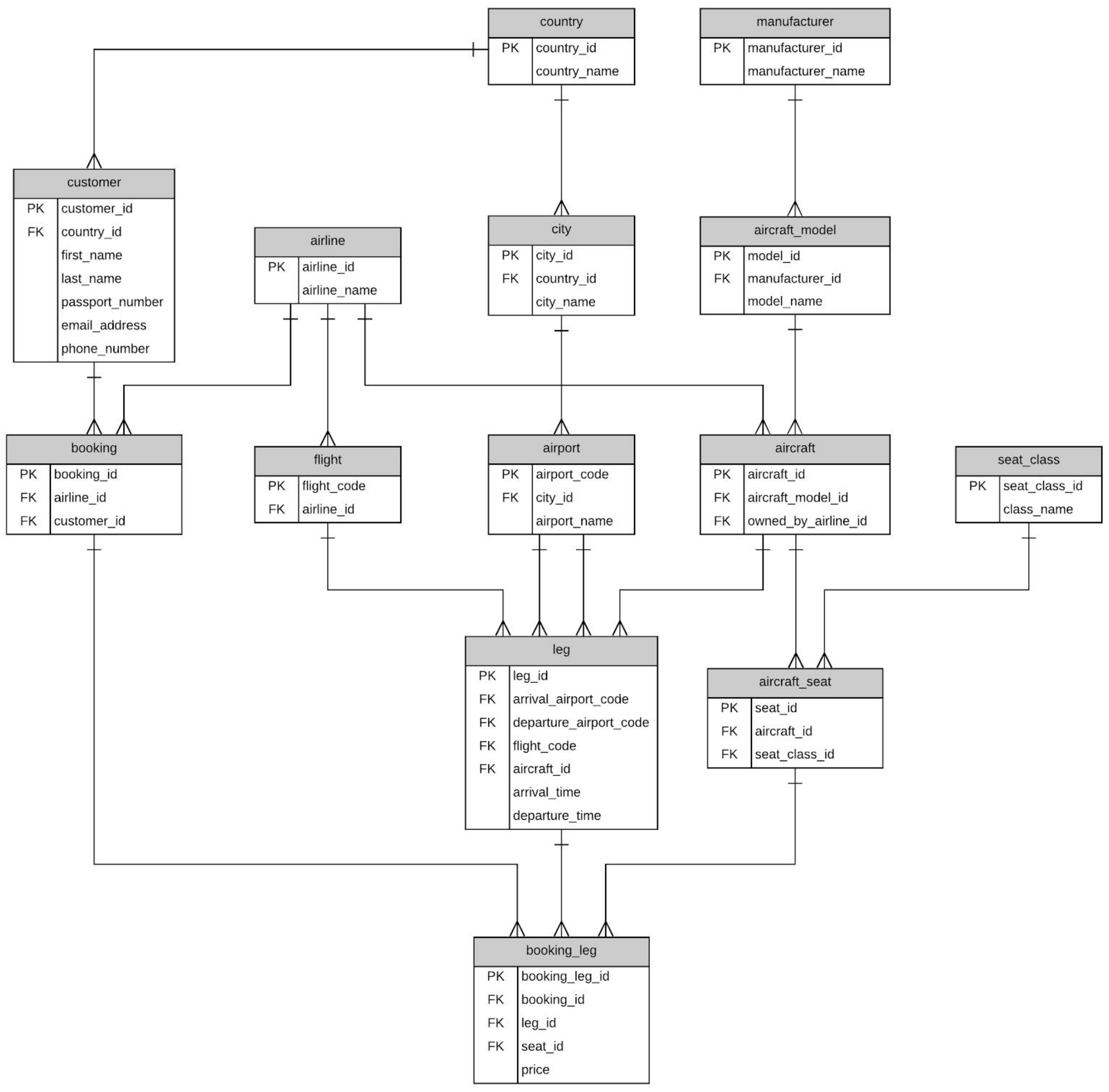
seat_class: <u>seat_class_id</u>, class_name

customer: <u>customer_id</u>, first_name, last_name, passport_number, email_address, phone_number, *country_id*

booking: <u>booking_id</u>, *airline_id*, *customer_id*

booking_leg: <u>booking_leg_id</u>, *booking_id*, *leg_id*, *seat_id*, price

Our diagram looks like this:



Sample Queries

The best way to validate the data is to think about what kind of queries would need to be run on it, and work out how they could be run.

Q: Find the flight details, seat and class details, and airport details for a particular booking.

Select all data from the flight, aircraft_seat, seat_class, booking_leg, booking, leg, and airport tables, for a specific booking reference.

Q: Find all the manufacturer, model, and the number of each, for all aircraft owned by an airline

Select the manufacturer name, the model name, and the airline name from the manufacturer, aircraft_model, aircraft, and airline tables, along with a count of each manufacturer and model

Q: Find all flights arriving in a certain airport on a certain day.

Select all information from the leg table, joined to the airport table, where the airport is a specific code.

Conclusion

These three examples have explained how to turn a few sentences of requirements into a database design that meets third normal form.

They have covered different issues that you can have when designing a database.

I hope they have been helpful for you!

If you have any questions on this guide, let me know at ben@databasestar.com

Thanks!

Ben Brumm

www.DatabaseStar.com