The 3 Normal Forms:

A Tutorial

by Fred Coulson

Table of Contents

Introduction

This is meant to be a very brief tutorial aimed at beginners who want to get a conceptual grasp on the database normalization process. I find it very difficult to visualize these concepts using words alone, so I shall rely as much as possible upon pictures and diagrams.

To demonstrate the main principles involved, we will take the classic example of an **Invoice** and level it to the Third Normal Form. We will also construct an **Entity Relationship Diagram** (ERD) of the database as we go.

Important Note: This is *not* a description of how you would actually design and implement a database. The sample database screenshots are not meant to be taken literally, but merely as visual aids to show how the raw data gets shuffled about as the table structure becomes increasingly normalized.

Purists and academics may not be interested in this treatment. I will not cover issues such as the benefits and drawbacks of normalization. For those who wish to pursue the matter in greater depth, a list of references for further reading is provided at the end.

For the most part, the first three normal forms are common sense. When people sit down to design a database, they often already have a partially-normalized structure in mind—normalization is a natural way of perceiving relationships between data and no special skill in mathematics or set theory is required.

In fact, it usually takes quite a bit of work to *de*-normalize a database (that is, remove the natural efficient relationships that a normalized data structure provides). Denormalization is a fairly common task, but it is beyond the scope of this presentation.

To begin: First, memorize the 3 normal forms so that you can recite them in your sleep. The meaning will become clear as we go. Just memorize them for now:

1. No repeating elements or groups of elements
2. No partial dependencies on a concatenated key
3. No dependencies on non-key attributes

The Problem:
Keeping Track of a Stack of Invoices

Consider a typical invoice (**Figure A**).

Figure A: Invoice

Those of us who have an ordered mind but aren't quite aware of relational databases might try to capture the Invoice data in a spreadsheet, such as Microsoft Excel.

Figure A-1: orders spreadsheet

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Invoice No.	Date	Cust. No.	Cust. Name	Cust. Address	Cust. City	Cust. State	Item ID	Item Descri	Item Qty.	Item Price	Item Total	Order Total Price
2	125	9/13/2002	56	Foo, Inc.	23 Main St., Thorpleburg	Thorpleburg	TX	563	56" Blue Fre	4	$ 3.50	$ 14.00	$ 82.00
3								851	Spline End i	32	$ 0.25	$ 8.00	$ 82.00
4								652	3" Red Free	5	$ 12.00	$ 60.00	$ 82.00
5	126	9/14/2002	2	Freens R Us	1600 Pennsylvania Avenu	Washington	DC	563	56" Blue Fre	500	$ 3.50	$1,750.00	$ 10,750.00
6								652	3" Red Free	750	$ 12.00	$9,000.00	$ 10,750.00

This isn't a bad approach, since it records every purchase made by every customer. But what if you started to ask complicated questions, such as:

- How many 3" Red Freens did *Freens R Us* order in 2002?
- What are total sales of 56" Blue Freens in the state of Texas?
- What items were sold on July 14, 2003?

As the spreadsheet grows it becomes increasingly difficult to ask the spreadsheet these questions. In an attempt to put the data into a state where we can reasonably expect to answer such questions, we begin the **normalization** process.

First Normal Form:
No Repeating Elements or Groups of Elements

Take a look at rows 2, 3 and 4 on the spreadsheet in Figure A-1. These represent all the data we have for a single invoice (Invoice #125).

In database lingo, this group of rows is referred to as a single *database row*. Never mind the fact that one database row is made up here of three spreadsheet rows: It's an unfortunate ambiguity of language. Academic database theoreticians have a special word that helps a bit with the ambiguity: they refer to the "thing" encapsulated by rows 2, 3 and 4 as a **tuple** (pronounced *tu'ple* or *too'ple*). We're not going to use that word here (and if you're lucky, you'll never hear it again for the rest of your database career). Here, we will refer to this thing as a ***row***.

So, First Normal Form (NF1) wants us to get rid of **repeating elements**. What are those?

Again we turn our attention to the first invoice (#125) in Figure A-1. Cells H2, H3, and H4 contain a list of Item ID numbers. This is a *column* within our first database row. Similarly, I2-I4 constitute a single column; same with J2-J4, K2-K4, L2-L4, and M2-M4. Database columns are sometimes referred to as **attributes** (rows/columns are the same as tuples/attributes).

You will notice that each of these columns contains a list of values. It is precisely these lists that NF1 objects to: NF1 abhors lists or arrays within a single database column. NF1 craves **atomicity**: the indivisibility of an attribute into similar parts.

Therefore it is clear that we have to do something about the repeating **item information** data within the row for Invoice #125. On Figure A-1, that is the following cells:

- H2 through M2
- H3 through M3
- H4 through M4

Similar (though not necessarily *identical*) data repeats within Invoice #125's row. We can satisfy NF1's need for atomicity quite simply: by separating each item in these lists into its own row.

Figure A-2: flattened orders spreadsheet

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Invoice No.	Date	Cust. No.	Cust. Name	Cust. Address	Cust. City	Cust. State	Item ID	Item Descri	Item Qty.	Item Price	Item Total	Order Total Price
2	125	9/13/2002	56	Foo, Inc.	23 Main St., Thorpleburg	Thorpleburg	TX	563	56" Blue Fre	4	$ 3.50	$ 14.00	$ 82.00
3	125	9/13/2002	56	Foo, Inc.	23 Main St., Thorpleburg	Thorpleburg	TX	851	Spline End	32	$ 0.25	$ 8.00	$ 82.00
4	125	9/13/2002	56	Foo, Inc.	23 Main St., Thorpleburg	Thorpleburg	TX	652	3" Red Free	5	$ 12.00	$ 60.00	$ 82.00
5	126	9/14/2002	2	Freens R Us	1600 Pennsylvania Avenu	Washington	DC	563	56" Blue Fre	500	$ 3.50	$1,750.00	$ 10,750.00
6	126	9/14/2002	2	Freens R Us	1600 Pennsylvania Avenu	Washington	DC	652	3" Red Free	750	$ 12.00	$9,000.00	$ 10,750.00

I can hear everyone objecting: We were trying to reduce the amount of duplication, and here we have introduced *more*! Just look at all that duplicated customer data!

Don't worry. The kind of duplication that we introduce at this stage will be addressed when we get to the **Third Normal Form**. Please be patient; this is a necessary step in the process.

We have actually only told half the story of NF1. Strictly speaking, NF1 addresses two issues:

1. A row of data cannot contain repeating groups of similar data (**atomicity**)
2. Each row of data must have a unique identifier (or **Primary Key**)

We have already dealt with atomicity. But to make the point about Primary Keys, we shall bid farewell to the spreadsheet and move our data into a relational database management system (RDBMS). Here we shall use Microsoft Access to create the **orders** table, as in **Figure B**:

Figure B: orders table

order_id	order_date	customer_id	customer_name	customer_addre	customer_city	customer_	item_id	item_description	item_qty	item_price	item_total_price	order_total_price
125	9/13/2002	56	Foo, Inc.	23 Main St., Th	Thorpleburg	TX	563	56" Blue Freen	4	$3.50	$14.00	$82.00
125	9/13/2002	56	Foo, Inc.	23 Main St., Th	Thorpleburg	TX	851	Spline End (Xtra	32	$0.25	$8.00	$82.00
125	9/13/2002	56	Foo, Inc.	23 Main St., Th	Thorpleburg	TX	652	3" Red Freen	5	$12.00	$60.00	$82.00
126	9/14/2002	2	Freens R Us	1600 Pennsylva	Washington	DC	563	56" Blue Freen	500	$3.50	$1,750.00	$10,750.00
126	9/14/2002	2	Freens R Us	1600 Pennsylva	Washington	DC	652	3" Red Freen	750	$12.00	$9,000.00	$10,750.00

Record: |◄ ◄| 6 | ► ►| ►* | of 6

This looks pretty much the same as the spreadsheet, but the difference is that within an RDBMS we can identify a **primary key**. A primary key is a column (or group of columns) that uniquely identifies *each row*.

As you can see from Figure B, there is no single column that uniquely identifies each row. However, if we put a number of columns together, we can satisfy this requirement.

The two columns that together uniquely identify each row are **order_id** and **item_id**: no two rows have the same combination of order_id and item_id. Therefore, together they qualify to be used as the table's primary key. Even though they are in two different table columns, they are treated as a single entity. We call them **concatenated**.

*A value that uniquely identifies a row is called a **primary key**.*

*When this value is made up of two or more columns, it is referred to as a **concatenated primary key**.*

The underlying structure of the orders table can be represented as **Figure C**:

We identify the columns that make up the primary key with the **PK** notation. Figure C is the beginning of our **Entity Relationship Diagram** (or ERD).

Our database schema now satisfies the two requirements of First Normal Form: **atomicity** and **uniqueness**. Thus it fulfills the most basic criteria of a *relational database*.

What's next?

Figure C: orders table structure

orders
order_id (PK)
order_date
customer_id
customer_name
customer_address
customer_city
customer_state
item_id (PK)
item_description
item_qty
item_price
item_total_price
order_total_price

Second Normal Form:
No Partial Dependencies on a Concatenated Key

Next we test each table for **partial dependencies on a concatenated key**. This means that for a table that has a concatenated primary key, each column in the table that is not part of the primary key *must* depend upon the entire concatenated key for its existence. If any column only depends upon one part of the concatenated key, then we say that the entire table has failed Second Normal Form and we must create another table to rectify the failure.

Still not clear? To try and understand this, let's take apart the **orders** table column by column. For each column we will ask the question,

> *Can this column exist **without** one or the other part of the concatenated primary key?*

If the answer is "yes" — even once — then the table fails Second Normal Form.

Refer to **Figure C** again to remind us of the orders table structure.

First, recall the meaning of the two columns in the primary key:

orders
order_id (PK)
order_date
customer_id
customer_name
customer_address
customer_city
customer_state
item_id (PK)
item_description
item_qty
item_price
item_total_price
order_total_price

- **order_id** identifies the invoice that this item comes from.
- **item_id** is the inventory item's unique identifier. You can think of it as a part number, inventory control number, SKU, or UPC code.

We don't analyze these columns (since they are part of the primary key). Now consider the remaining columns...

order_date is the date on which the order was made. Obviously it relies on **order_id**; an order date has to have an order, otherwise it is only a date. But can an order date exist without an **item_id**?

The short answer is yes: order_date relies on order_id, not item_id. Some of you might object, thinking that this means you could have a dated order with no items (an empty invoice, in effect). But this is not what we are saying at all: All we are trying to establish here is whether a *particular* order on a *particular* date relies on a particular item. Clearly, it does not. The problem of how to prevent empty

orders falls under a discussion of "business rules" and could be resolved using check constraints or application logic; it is not an issue for Normalization to solve.

Therefore: order_date *fails* Second Normal Form. ✖

So voilá, our table has already failed Second Normal Form. But let's continue with testing the other columns. We have to find all the columns that fail the test, and then we do something special with them.

customer_id is the ID number of the customer who placed the order. Does it rely on **order_id**? No: a customer can exist without placing any orders. Does it rely on **item_id**? No: for the same reason. This is interesting: customer_id (along with the rest of the customer_* columns) does not rely on *either* member of the primary key. What do we do with these columns?

We don't have to worry about them until we get to Third Normal Form. We mark them as "unknown" for now. ❓

item_description is the next column that is not itself part of the primary key. This is the plain-language description of the inventory item. Obviously it relies on **item_id**. But can it exist without an **order_id**?

Yes! An inventory item (together with its "description") could sit on a warehouse shelf forever, and never be purchased... It can exist independent of an order. item_description **fails** the test. ✖

item_qty refers to the number of items purchased on a particular invoice. Can this quantity exist without an item_id? Impossible: we cannot talk about the "amount of nothing" (at least not in database design). Can the quantity exist without an order_id? No: a quantity that is purchased with an invoice is meaningless without an invoice. So this column does not violate Second Normal Form: item_qty depends on **both** parts of our concatenated primary key. ✔

item_price is similar to **item_description**. It depends on the **item_id** but not on the **order_id**, so it *does* violate Second Normal Form. ✖

item_total_price is a tricky one. On the one hand, it seems to depend on both **order_id** and **item_id**, in which case it passes Second Normal Form. On the other hand, it is a *derived value*: it is merely the product of item_qty and item_price. What to do with this field?

In fact, this field does not belong in our database at all. It can easily be reconstructed outside of the database proper; to include it would be redundant (and could quite possibly introduce corruption). Therefore we will discard it and speak of it no more.

order_total_price, the sum of all the item_total_price fields for a particular order, is another derived value. We discard this field too.

Here is the markup from our NF2 analysis of the **orders** table:

Figure C (revised):

What do we do with a table that fails Second Normal Form, as this one has? First we take out the second half of the concatenated primary key (**item_id**) and put it in its own table.

All the columns that depend on item_id - whether in whole or in part - follow it into the new table. We call this new table **order_items** (see **Figure D**).

The other fields — those that rely on just the first half of the primary key (order_id) and those we aren't sure about — stay where they are.

Figure D: orders and order_items tables

There are several things to notice:

1. We have brought a copy of the **order_id** column over into the **order_items** table. This allows each order_item to "remember" which order it is a part of.
2. The **orders** table has fewer rows than it did before.
3. The **orders** table no longer has a concatenated primary key. The primary key now consists of a single column, **order_id**.
4. The **order_items** table *does* have a concatenated primary key.

Here is the table structure (**Figure E**):

Figure E: orders and order_items table structure
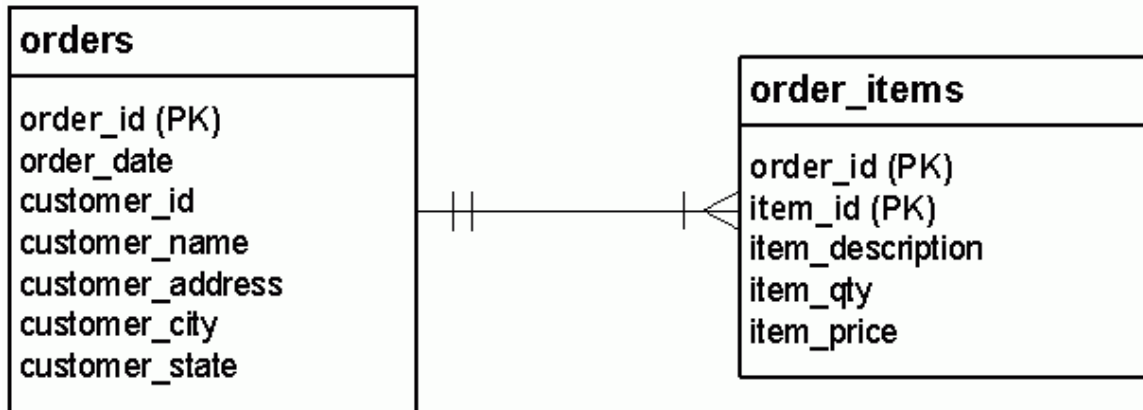
If you are new to Entity Relationship Diagrams, pay close attention to the line that connects these two tables. This line means, in English,

- *each order can be associated with any number of order-items, but **at least** one;*
- *each order-item is associated with one order, and **only** one.*
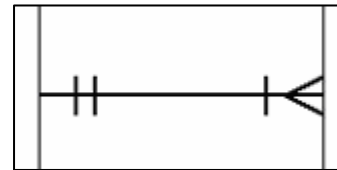
There are other ways of depicting these table-to-table relationships; here I am using one of many standard conventions.

Second Normal Form: Phase II

But wait, there's more!

Remember, NF2 only applies to tables with a concatenated primary key. Now that **orders** has a single-column primary key, it has passed Second Normal Form. Congratulations!

order_items, however, still has a concatenated primary key. We have to pass it through the NF2 analysis again, and see if it measures up. We ask the same question we did before:

> Can this column exist **without** one or the other part of the concatenated primary key?

First, refer to **Figure F**, to remind us of the **order_items** table structure.

Figure F:

Now consider the columns that are not part of the primary key...

item_description relies on **item_id**, but not order_id. So (surprise), this column once again fails NF2. ✖

item_qty relies on both members of the primary key. It does *not* violate NF2. ✔

item_price relies on the **item_id** but not on the **order_id**, so it *does* violate Second Normal Form. ✖

We should be getting good at this now. Here is the marked up table diagram:

Figure F (revised):

So, we take the fields that fail NF2 and create a new table. We call this new table **items**:

Figure G: order_items and items table

order_id	item_id	item_qty
125	563	4
125	851	32
125	652	5
126	563	500
126	652	750

| item_id	item_description	item_price
563	56" Blue Freen	$3.50
851	Spline End (Xtra Large)	$0.25
652	3" Red Freen	$12.00

But wait, something's wrong. When we did our first pass through the NF2 test, we took out *all* the fields that relied on item_id and put them into the new table. This time, we are only taking the fields that failed the test: in other words, **item_qty** stays where it is. Why? What's different this time?

The difference is that in the first pass, we removed the **item_id** key from the **orders** table altogether, because of the one-to-many relationship between orders and order-items. Therefore the **item_qty** field *had* to follow **item_id** into the new table.

In the second pass, **item_id** was not removed from the **order_items** table because of the *many-to-one* relationship between order-items and items. Therefore, since item_qty does not violate NF2 this time, it is permitted to stay in the table with the two primary key parts that it relies on.

This should be clearer with a new ERD. Here is how the **items** table fits into the overall database schema:

Figure H:
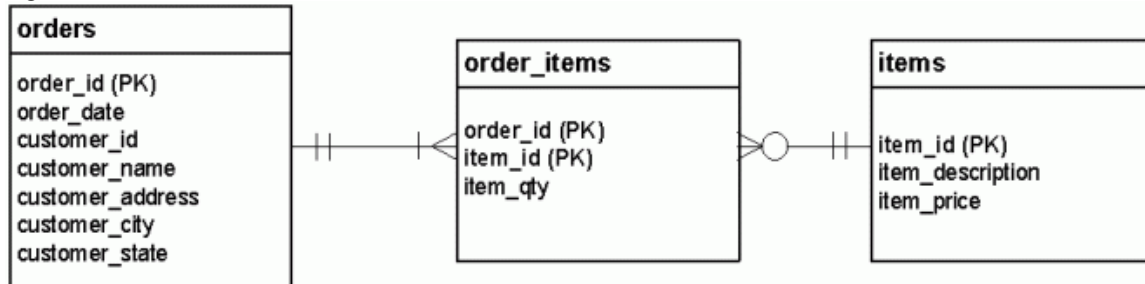
The line that connects the **items** and **order_items** tables means the following:

- *Each item can be associated with any number of lines on any number of invoices, including zero;*
- *each order-item is associated with one item, and **only** one.*

These two lines are examples of one-to-many relationships. This three-table structure, considered in its entirety, is how we express a many-to-many relationship:

> *Each order can have many items; each item can belong to many orders.*

Notice that this time, we did not bring a copy of the order_id column into the new table. This is because individual items do not need to have knowledge of the orders they are part of. The order_items table takes care of remembering this relationship via the order_id and item_id columns. Taken together these columns comprise the primary key of order_items, but taken separately they are foreign keys or pointers to rows in other tables. More about foreign keys when we get to Third Normal Form.

Notice, too, that our new table does not have a concatenated primary key, so it passes NF2. At this point, we have succeeded in attaining Second Normal Form!

Third Normal Form:
No Dependencies on Non-Key Attributes

At last, we return to the problem of the repeating Customer information. As our database now stands, if a customer places more than one order then we have to input all of that customer's contact information again. This is because there are columns in the **orders** table that rely on "non-key attributes".

To better understand this concept, consider the **order_date** column. Can it exist independent of the **order_id** column? *No*: an "order date" is meaningless without an order. **order_date** is said to depend on a key attribute (order_id is the "key attribute" because it is the primary key of the table).

What about **customer_name** — can it exist on its own, outside of the **orders** table?

Yes. It is meaningful to talk about a customer name without referring to an order or invoice. The same goes for **customer_address**, **customer_city**, and **customer_state**. These four columns actually rely on **customer_id**, which is not a key in this table (it is a *non-key attribute*).

These fields belong in their own table, with **customer_id** as the primary key (see Figure I).

Figure I:
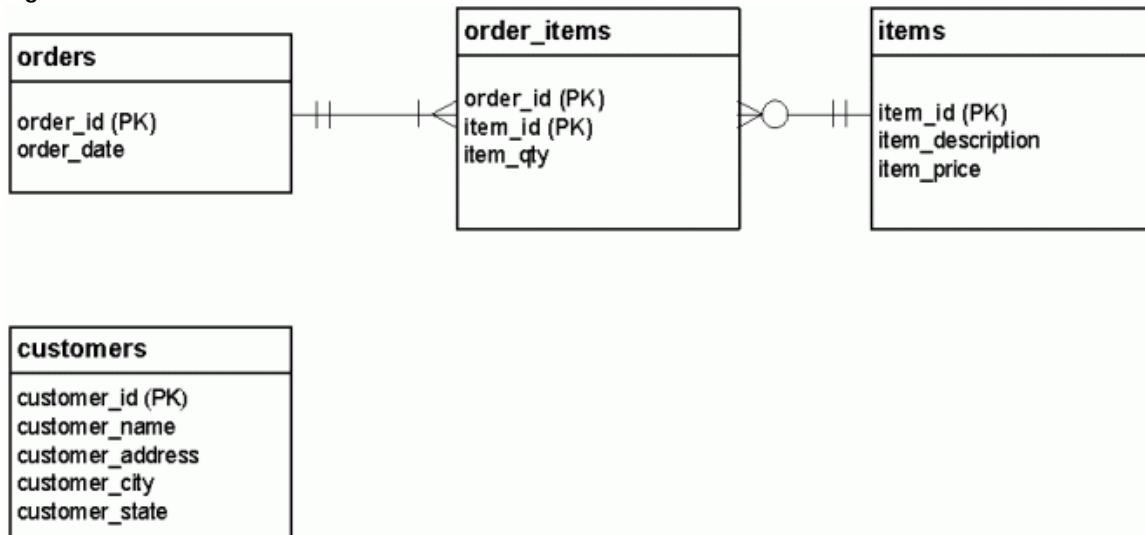
However, you will notice in Figure I that we have severed the relationship between the orders table and the Customer data that used to inhabit it.

This won't do at all.

We have to restore the relationship by creating an entity called a **foreign key** (indicated in our diagram by **(FK)**) in the orders table. A foreign key is essentially a column that points to the primary key in another table. **Figure J** describes this relationship, and shows our completed ERD:

Figure J: Final ERD
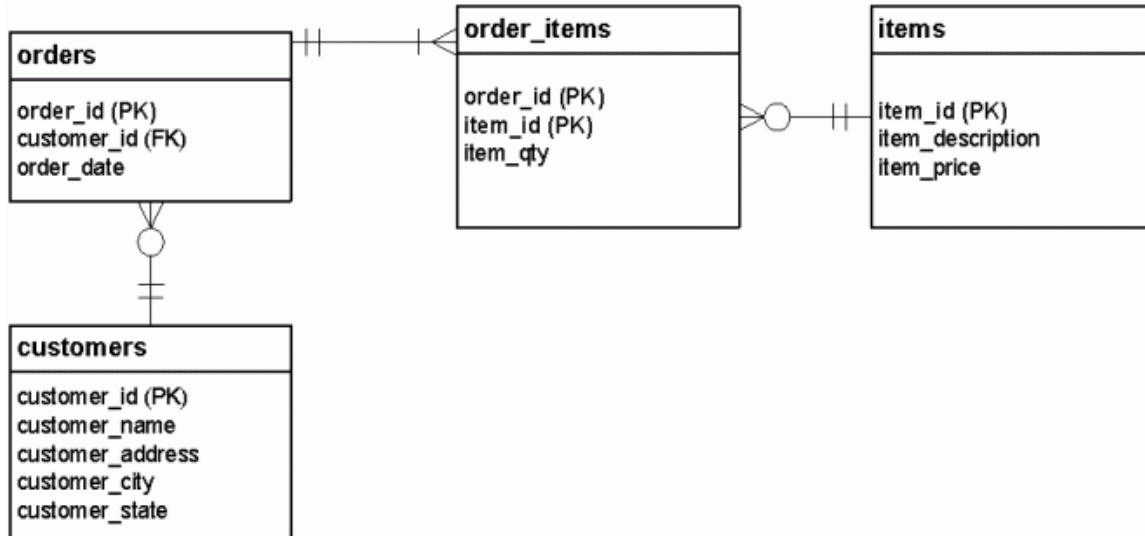
The relationship that has been established between the **orders** and **customers** table may be expressed in this way:

- *each order is made by one, and **only** one customer;*
- *each customer can make any number of orders, including zero.*

One final refinement...

You will notice that the **order_id** and **item_id** columns in order_items perform a dual purpose: not only do they function as the (concatenated) primary key for **order_items**, they also individually serve as foreign keys to the **order** table and **items** table respectively.

Figure J.1 documents this fact, and shows our completed ERD:

Figure J: Final ERD
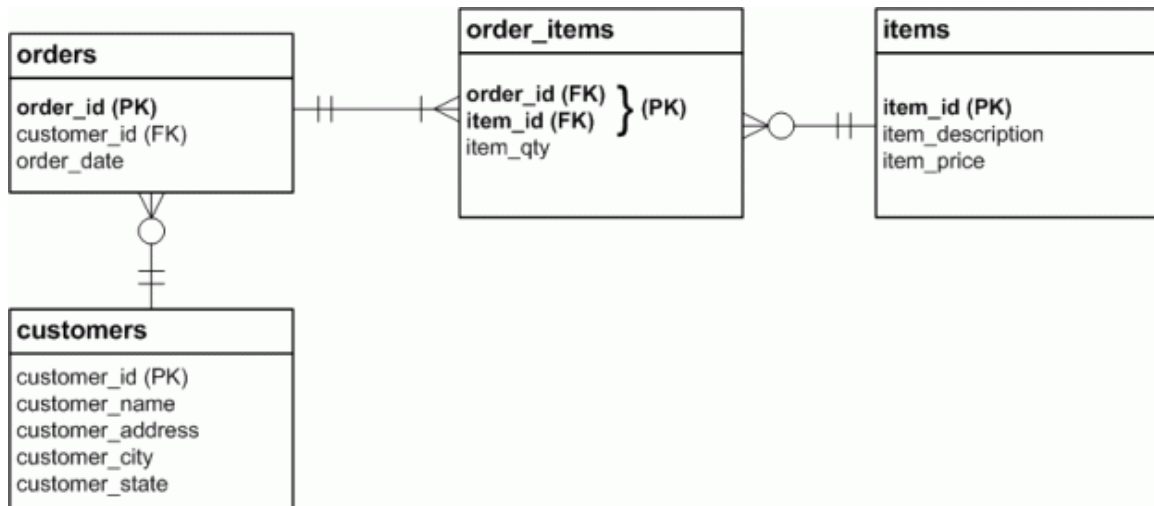
And finally, here is what the data in each of the four tables looks like. Notice that NF3 removed **columns** from a table, rather than **rows**.

Figure K:

References for Further Reading

Needless to say, there's a lot more to it than this. If you want to read more about the theory and practice of the 3 Normal Forms, here are some suggestions:

- [The Art of Analysis](#), by Dr. Art Langer, devotes considerable space to normalization. Springer-Verlag Telos (January 15, 1997)
  ISBN: 0387949720
- Dr. Codd's seminal 1969 paper on database normalization:
  www.acm.org/classics/nov95
- The [Wikipedia article on normalization](#) discusses all five normal forms:
  en.wikipedia.org/wiki/Database_normalization