Unit 4.0 – Normalization

Introduction

Database normalization is process used to organize a database into tables and columns. The idea is that a table should be about a specific topic and that only those columns which support that topic are included. For example, a spreadsheet containing information about sales people and customers serves several purposes:

- Identify sales people in your organization
- List all customers your company calls upon to sell product
- Identify which sales people call on specific customers.

By limiting a table to one purpose you reduce the number of duplicate data that is contained within your database, which helps eliminate some issues stemming from database modifications. To assist in achieving these objectives, some rules for database table organization have been developed. The stages of organization are called normal forms; there are three normal forms most databases adhere to using. As tables satisfy each successive normalization form, they become less prone to database modification anomalies and more focused toward a sole purpose or topic. Before we move on be sure you understand the definition of a database table.

Reasons for Normalization

There are three main reasons to normalize a database. The first is to minimize duplicate data, the second is to minimize or avoid data modification issues, and the third is to simplify queries. As we go through the various states of normalization we'll discuss how each form addresses these issues, but to start, let's look at some data which hasn't been normalized and discuss some potential pitfalls. Once these are understood, I think you'll better appreciate the reason to normalize the data. Consider the following table:

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Note: The primary key columns are underlined

The first thing to notice is this table serves many purposes including:

- Identifying the organization's salespeople
- Listing the sales offices and phone numbers
- Associating a salesperson with an sales office
- Showing each salesperson's customers

As a DBA this raises a red flag. In general, I like to see tables that have one purpose. Having the table serve many purposes introduces many of the challenges; namely, data duplication, data update issues, and increased effort to query data.

Data Duplication and Modification Anomalies

Notice that for each SalesPerson we have listed both the SalesOffice and OfficeNumber. This information is duplicated for each SalesPerson. Duplicated information presents two problems:

- It increases storage and decrease performance.
- It becomes more difficult to maintain data changes.

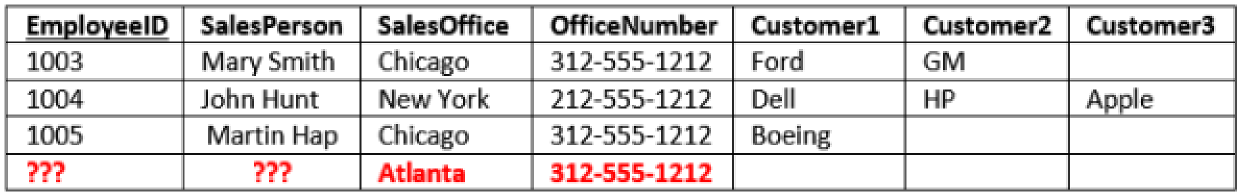
For example:

- Consider if we move the Chicago office to Evanston, IL. To properly reflect this in our table, we need to update the entries for all the SalesPersons currently in Chicago. Our table is a small example, but you can see if it were larger, that potentially this could involve hundreds of updates.
- Also consider what would happen if John Hunt quits. If we remove his entry, then we lose the information for New York.

These situations are modification anomalies. There are three modification anomalies that can occur:

Insert Anomaly

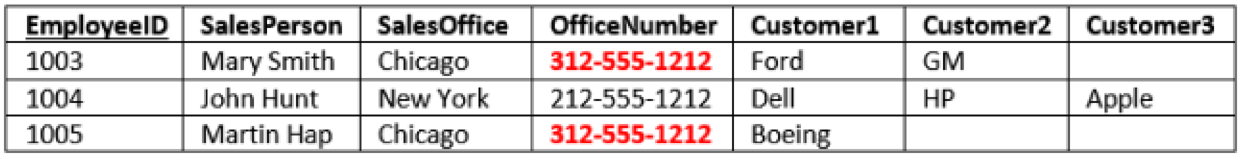
There are facts we cannot record until we know information for the entire row. In our example we cannot record a new sales office until we also know the sales person. Why? Because in order to create the record, we need provide a primary key. In our case this is the EmployeeID.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	**???**	**Atlanta**	**312-555-1212**			

Update Anomaly

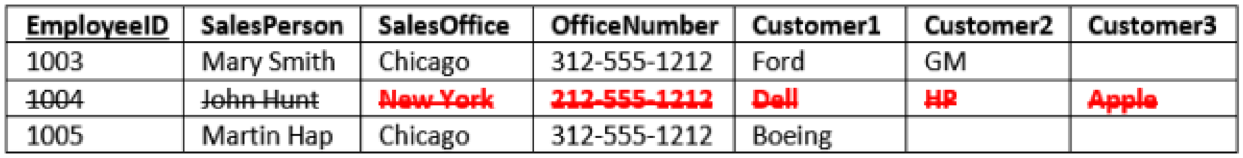
The same information is recorded in multiple rows. For instance if the office number changes, then there are multiple updates that need to be made. If these updates are not successfully completed across all rows, then an inconsistency occurs.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	**312-555-1212**	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	**312-555-1212**	Boeing		

Deletion Anomaly

Deletion of a row can cause more than one set of facts to be removed. For instance, if John Hunt retires, then deleting that row cause use to lose information about the New York office.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Search and Sort Issues

The last reason we'll consider is making it easier to search and sort your data. In the SalesStaff table if you want to search for a specific customer such as Ford, you would have to write a query like

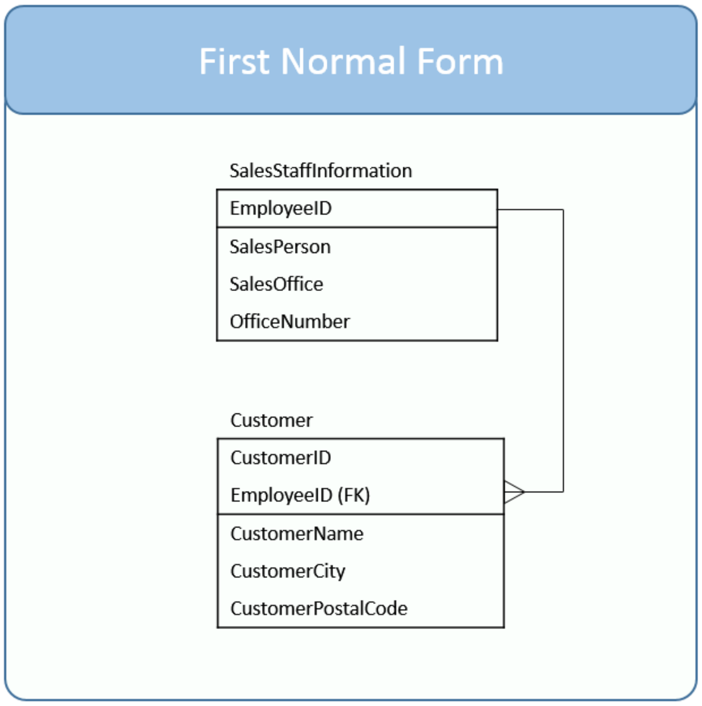
```
SELECT SalesOffice
FROM SalesStaff
WHERE Customer1 = 'Ford' OR
      Customer2 = 'Ford' OR
      Customer3 = 'Ford'
```

Clearly if the customer were somehow in one column our query would be simpler. Also, consider if you want to run a query and sort by customer. The way the table is currently defined, this isn't possible, unless you use three separate queries with a UNION. These anomalies can be eliminated or reduced by properly separating the data into different tables, to house the data in tables which serve a single purpose. The process to do this is called normalization, and the various stages you can achieve are called the normal forms.

Definition of Normalization

There are three common forms of normalization: 1st, 2nd, and 3rd normal form. There are several additional forms, such as BCNF, but I consider those advanced, and not too necessary to learn in the beginning. The forms are progressive, meaning that to qualify for 3rd normal form a table must first satisfy the rules for 2nd normal form, and 2nd normal form must adhere to those for 1st normal form. Before we discuss the various forms and rules in details, let's summarize the various forms:

- **First Normal Form** – The information is stored in a relational table and each column contains atomic values, and there are not repeating groups of columns.
- **Second Normal Form** – The table is in first normal form and all the columns depend on the table's primary key.
- **Third Normal Form** – the table is in second normal form and all of its columns are not transitively dependent on the primary key



First Normal Form

The first steps to making a proper SQL table is to ensure the information is in first normal form. Once a table is in first normal form it is easier to search, filter, and sort the information. The rules to satisfy 1st normal form are:

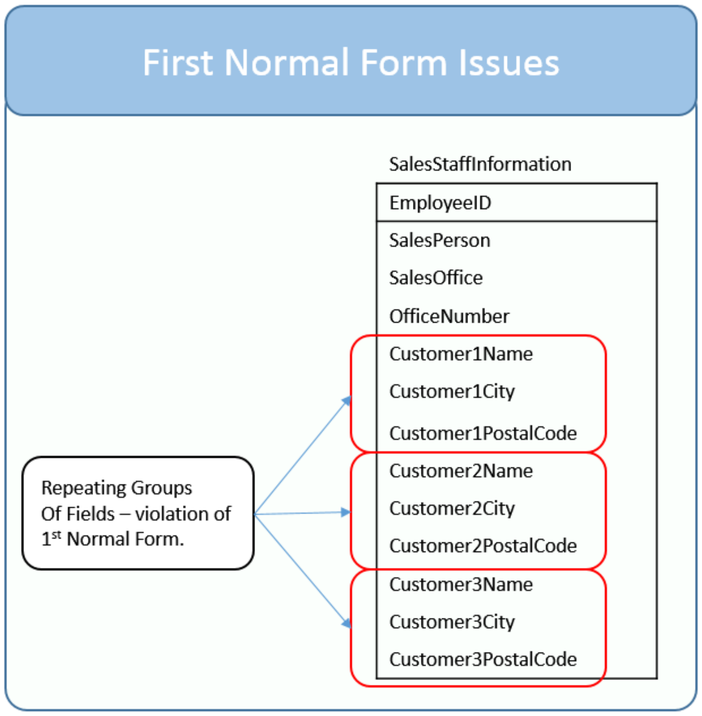
- That the data is in a database table. The table stores information in rows and columns where one or more columns, called the primary key, uniquely identify each row.
- Each column contains atomic values, and there are not repeating groups of columns.

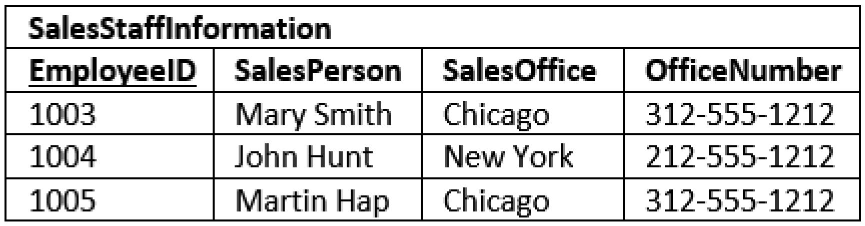
Tables in first normal form cannot contain sub columns. That is, if you are listing several cities, you cannot list them in one column and separate them with a semi-colon.

When a value is atomic, the values cannot be further subdivided. For example, the value "Chicago" is atomic; whereas "Chicago; Los Angeles; New York" is not. Related to this requirement is the concept that a table should not contain repeating groups of columns such as Customer1Name, Customer2Name, and Customer3Name.

Our example table is transformed to first normal form by placing the repeating customer related columns into their own table. This is shown to the right.

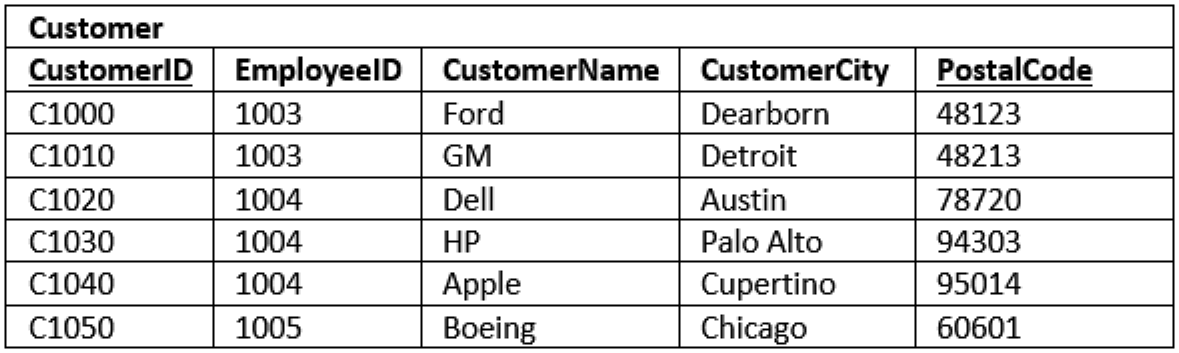
The repeating groups of columns now become separate rows in the Customer table linked by the EmployeeID foreign key. As mentioned in the lesson on Data Modeling, a foreign key is a value which matches back to another table's primary key. In this case, the customer table contains the corresponding EmployeeID for the SalesStaffInformation row. Here is our data in first normal form.





SalesStaffInformation			
EmployeeID	SalesPerson	SalesOffice	OfficeNumber
1003	Mary Smith	Chicago	312-555-1212
1004	John Hunt	New York	212-555-1212
1005	Martin Hap	Chicago	312-555-1212

Note: Primary Key: EmployeeID



Customer				
CustomerID	EmployeeID	CustomerName	CustomerCity	PostalCode
C1000	1003	Ford	Dearborn	48123
C1010	1003	GM	Detroit	48213
C1020	1004	Dell	Austin	78720
C1030	1004	HP	Palo Alto	94303
C1040	1004	Apple	Cupertino	95014
C1050	1005	Boeing	Chicago	60601

This design is superior to our original table in several ways:

1. The original design limited each SalesStaffInformation entry to three customers. In the new design, the number of customers associated to each design is practically unlimited.
2. It was nearly impossible to Sort the original data by Customer. You could, if you used the UNION statement, but it would be cumbersome. Now, it is simple to sort customers.
3. The same holds true for filtering on the customer table. It is much easier to filter on one customer name related column than three.
4. The insert and deletion anomalies for Customer have been eliminated. You can delete all the customer for a SalesPerson without having to delete the entire SalesStaffInformaiton row.

Modification anomalies remain in both tables, but these are fixed once we reorganize them as 2nd normal form.

Second Normal Form

I like to think the reason we place tables in 2nd normal form is to narrow them to a single purpose. Doing so bring's clarity to the database design, makes it easier for us to describe and use a table, and tends to eliminate modification anomalies.

This stems from the primary key identifying the main topic at hand, such as identifying buildings, employees, or classes, and the columns, serving to add meaning through descriptive attributes.

An EmployeeID isn't much on its own, but add a name, height, hair color and age, and now you're starting to describe a real person.

2NF – Second Normal Form Definition

A table is in Second Normal Form if:

- The table is in 1st normal form, and
- All the non-key columns are dependent on the table's primary key.

The primary key provides a means to uniquely identify each row in a table. When we talk about columns depending on the primary key, we mean, that in order to find a particular value, such as what color is Kris' hair, you would first have to know the primary key, such as an EmployeeID, to look up the answer.

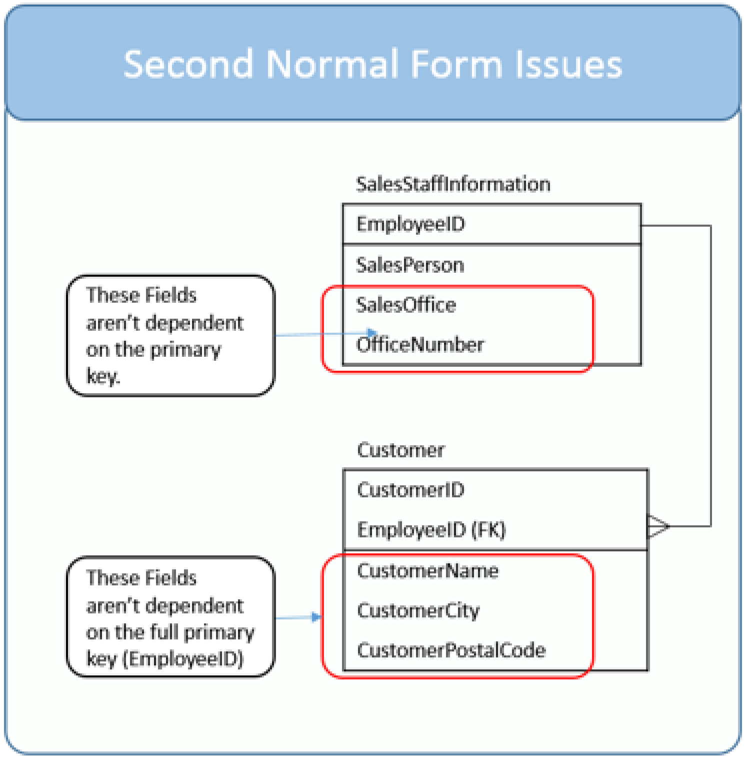
Once you identify a table's purpose, then look at each of the table's columns and ask yourself, "Does this column serve to describe what the primary key identifies?"

- If you answer "yes," then the column is dependent on the primary key and belongs in the table.
- If you answer "no," then the column should be moved different table.

When all the columns relate to the primary key, they naturally share a common purpose, such as describing an employee. That is why I say that when a table is in second normal form, it has a single purpose, such as storing employee information.

So far, we have taken our example to the first normal form, and it has several issues.

The first issue is the SalesStaffInformation table has two columns which aren't dependent on the EmployeeID. Though they are used to describe which office the SalesPerson is based out of, the SalesOffice and OfficeNumber columns themselves don't serve to describe who the employee is.


Second Normal Form Issues
SalesStaffInformation
EmployeeID
SalesPerson
SalesOffice
OfficeNumber
These Fields aren't dependent on the primary key.
Customer
CustomerID
EmployeeID (FK)
CustomerName
CustomerCity
CustomerPostalCode
These Fields aren't dependent on the full primary key (EmployeeID)

The second issue is that there are several attributes which don't completely rely on the entire Customer table primary key. For a given customer, it doesn't make sense that you should have to know both the CustomerID and EmployeeID to find the customer.

It stands to reason you should only need to know the CustomerID. Given this, the Customer table isn't in 2nd normal form as there are columns that aren't dependent on the full primary key. They should be moved to another table.

These issues are identified on the right in red.

Fix the Model to 2NF Standards

Since the columns identified in red aren't completely dependent on the table's primary key, it stands to reason they belong elsewhere. In both cases, the columns are moved to new tables.

In the case of SalesOffice and OfficeNumber, a SalesOffice was created. A foreign key was then added to SalesStaffInformaiton so we can still describe in which office a sales person is based.

The changes to make Customer a second normal form table are a little trickier. Rather than move the offending columns CustomerName, CustomerCity, and CustomerPostalCode to new table, recognize that the issue is EmployeeID! The three columns don't depend on this part of the key. Really this table is trying to serve two purposes:

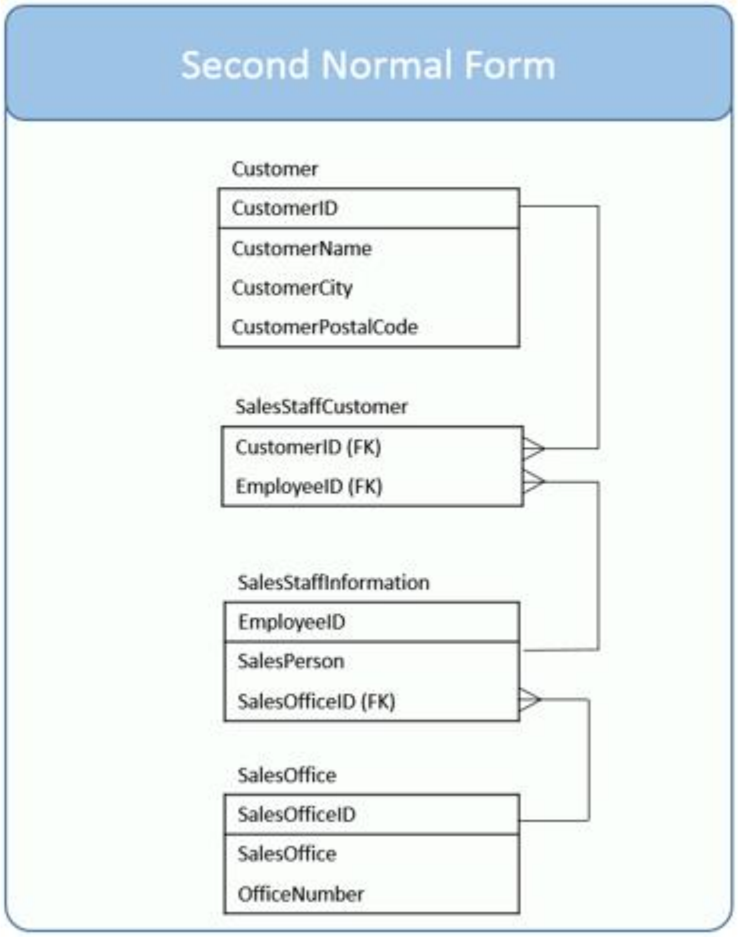
- To indicate which customers are called upon by each employee
- To identify customers and their locations.

For the moment remove EmployeeID from the table. Now the table's purpose is clear, it is to identify and describe each customer.

Now let's create a table named SalesStaffCustomer to describe which customers a sales person calls upon. This table has two columns CustomerID and EmployeeID. Together, they form a primary key. Separately, they are foreign keys to the Customer and SalesStaffInformation tables respectively.

With these changes made the data model, in second normal form, is shown to the right.

To better visualize this, here are the tables with data.



Customer			
CustomerID	CustomerName	CustomerCity	CustomerPostalCode
C1000	Ford	Dearborn	48123
C1010	GM	Detroit	48213
C1020	Dell	Austin	78720
C1030	HP	Palo Alto	94303
C1040	Apple	Cupertino	95014
C1050	Boeing	Chicago	60601

As you review the data in the tables notice that the redundancy is mostly eliminated. Also, see if you can find any update, insert, or deletion anomalies. Those too are gone. You can now eliminate all the sales people, yet retain customer records. Also, if all the SalesOffices close, it doesn't mean you have to delete the records containing sales people.

SalesStaffCustomer	
CustomerID	EmployeeID
C1000	1003
C1010	1003
C1020	1004
C1030	1004
C1040	1004
C1050	1005

The SalesStaffCustomer table is a strange one. It's just all keys! This type of table is called an associative table. An associative table is useful when you need to model a many-to-many relationship.

Each column is a foreign key. If you look at the data model you'll notice that there is a one to many relationship to this table from SalesStaffInformation and another from Customer. In effect the table allows you to bridge the two tables together.

SalesStaffInformation		
EmployeeID	SalesPerson	SalesOffice
1003	Mary Smith	S10
1004	John Hunt	S20
1005	Martin Hap	S10

SalesOffice		
SalesOfficeID	SalesOffice	OfficeNumber
S10	Chicago	312-555-1212
S20	New York	212-555-1212

For all practical purposes this is a pretty workable database. Three out of the four tables are even in third normal form, but there is one table which still has a minor issue, preventing it from being so.

Third Normal Form

Once a table is in second normal form, we are guaranteed that every column is dependent on the primary key, or as I like to say, the table serves a single purpose. But what about relationships among the columns? Could there be dependencies between columns that could cause an inconsistency?

A table containing both columns for an employee's age and birth date is spelling trouble, there lurks an opportunity for a data inconsistency!

How are these addressed? By the third normal form.

3NF – Third Normal Form Definition

A table is in third normal form if:

- A table is in 2nd normal form.
- It contains only columns that are non-transitively dependent on the primary key

Wow! That's a mouthful. What does non-transitively dependent mean? Let's break it down.

Transitive

When something is transitive, then a meaning or relationship is the same in the middle as it is across the whole. If it helps think of the prefix trans as meaning "across." When something is transitive, then if something applies from the beginning to the end, it also applies from the middle to the end.

Since ten is greater than five, and five is greater than three, you can infer that ten is greater than three.

In this case, the greater than comparison is transitive. In general, if A is greater than B, and B is greater than C, then it follows that A is greater than C.

If you're having a hard time wrapping your head around "transitive" I think for our purpose it is safe to think "through" as we'll be reviewing to see how one column in a table may be related to others, through a second column.

Dependence

An object has a dependence on another object when it relies upon it. In the case of databases, when we say that a column has a dependence on another column, we mean that the value can be derived from the other. For example, my age is dependent on my birthday. Dependence also plays an important role in the definition of the second normal form.

Transitive Dependence

Now let's put the two words together to formulate a meaning for transitive dependence that we can understand and use for database columns.

I think it is simplest to think of transitive dependence to mean a column's value relies upon another column through a second intermediate column.

Consider three columns: AuthorNationality, Author, and Book. Column values for AuthorNationality and Author rely on the Book; once the book is known, you can find out the Author or AuthorNationality. But also notice that the AuthorNationality relies upon Author. That is, once you know the Author, you can determine their nationality. In this sense then, the AuthorNationality relies upon Book, via Author. This is a transitive dependence.

This can be generalized as being three columns: A, B and PK. If the value of A relies on PK, and B relies on PK, and A also relies on B, then you can say that A relies on PK though B. That is A is transitively dependent on PK.

Let's look at some examples to understand further.

Primary Key (PK)	**Column A**	**Column B**	**Transitive Dependence?**
PersonID	FirstName	LastName	No, In Western cultures a person's last name is based on their father's LastName, whereas their FirstName is given to them.
PersonID	BodyMassIndex	IsOverweight	Yes, BMI over 25 is considered overweight.It wouldn't make sense to have the value IsOverweight be true when the BodyMassIndex was < 25.
PersonID	Weight	Sex	No:There is no direct link between the weight of a person and their sex.
VehicleID	Model	Manufacturer	Yes:Manufacturers make specific models. For instance, Ford creates the Fiesta; whereas, Toyota manufacturers the Camry.

To be non-transitively dependent, then, means that all the columns are dependent on the primary key (a criterion for 2nd normal form) and no other columns in the table.

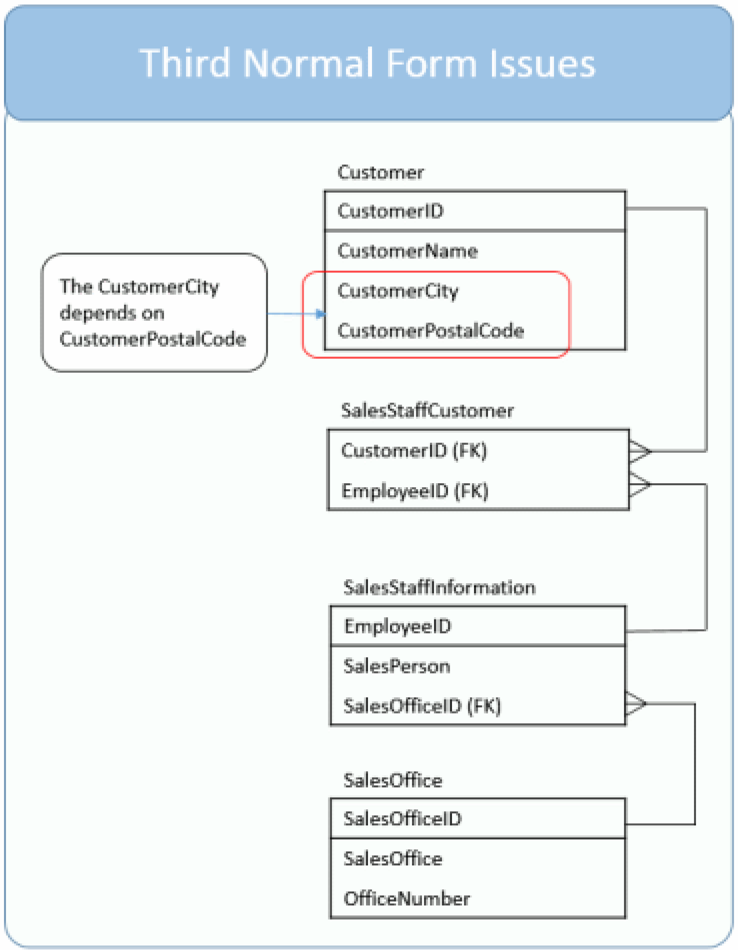
Issues with our Example Data Model

Let's review what we have done so far with our database. You'll see that I've found one transitive dependency:

CustomerCity relies on CustomerPostalCode which relies on CustomerID

A postal code applies to one city. Although all the columns are dependent on the primary key, CustomerID, there is an opportunity for an update anomaly as you could update the CustomerPostalCode without making a corresponding update to the CustomerCity.

We've identified this issue in red.



Fix the Model to 3NF Standards

For our model to be in third normal form, we need to remove the transitive dependencies. As we stated our dependency is:

CustomerCity relies on CustomerPostalCode which relies on CustomerID

It is OK that CustomerPostalCode relies on CustomerID; however, we break 3NF by including CustomerCity in the table. To fix this we'll create a new table, PostalCode, which includes PostalCode as the primary key and City as its sole column.

The CustomerPostalCode remains in the customer table. The CustomerPostalCode can then be designated a foreign key. In this way, through the relation, the city and postal code is still known for each customer. In addition, we've eliminated the update anomaly.