Normalization

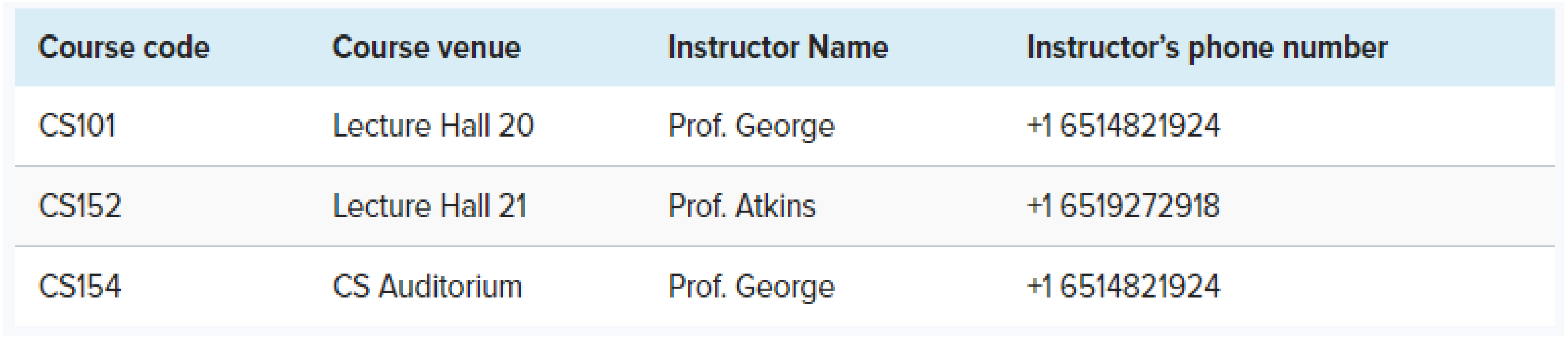
Why Normalization?

- When developing the schema of a relational database, one of the most important aspects to be taken into account is to ensure that the duplication is minimized.
- This is done for 2 purposes:
  1. Reducing the amount of storage needed to store the data.
  2. Avoiding unnecessary data conflicts that may creep in because of multiple copies of the same data getting stored.

Normalization in DBMS

- Database Normalization is a technique that helps in designing the schema of the database in an optimal manner so as to ensure the above points.
- The core idea of database normalization is to divide the tables into smaller subtables and store pointers to data rather than replicating it.
- For a better understanding of what we just said, here is a simple DBMS Normalization example:

- To understand (DBMS)normalization in the database with example tables, let's assume that we are supposed to store the details of courses and instructors in a university. Here is what a sample database could look like:



Course code	Course venue	Instructor Name	Instructor's phone number
CS101	Lecture Hall 20	Prof. George	+1 6514821924
CS152	Lecture Hall 21	Prof. Atkins	+1 6519272918
CS154	CS Auditorium	Prof. George	+1 6514821924

- Here, the data basically stores the course code, course venue, instructor name, and instructor’s phone number. At first, this design seems to be good. However, issues start to develop once we need to modify information. For instance, suppose, if Prof. George changed his mobile number. In such a situation, we will have to make edits in 2 places. What if someone just edited the mobile number against CS101, but forgot to edit it for CS154? This will lead to stale/wrong information in the database.

- This problem, however, can be easily tackled by dividing our table into 2 simpler tables:

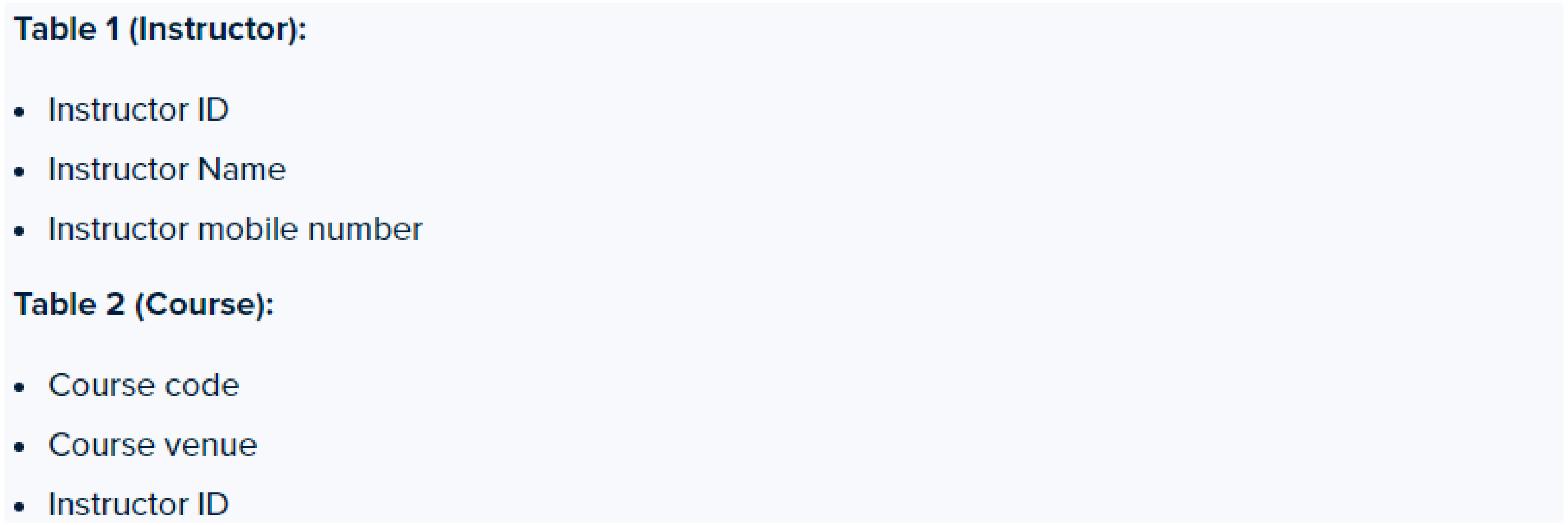


Table 1 (Instructor):

- Instructor ID
- Instructor Name
- Instructor mobile number

Table 2 (Course):

- Course code
- Course venue
- Instructor ID

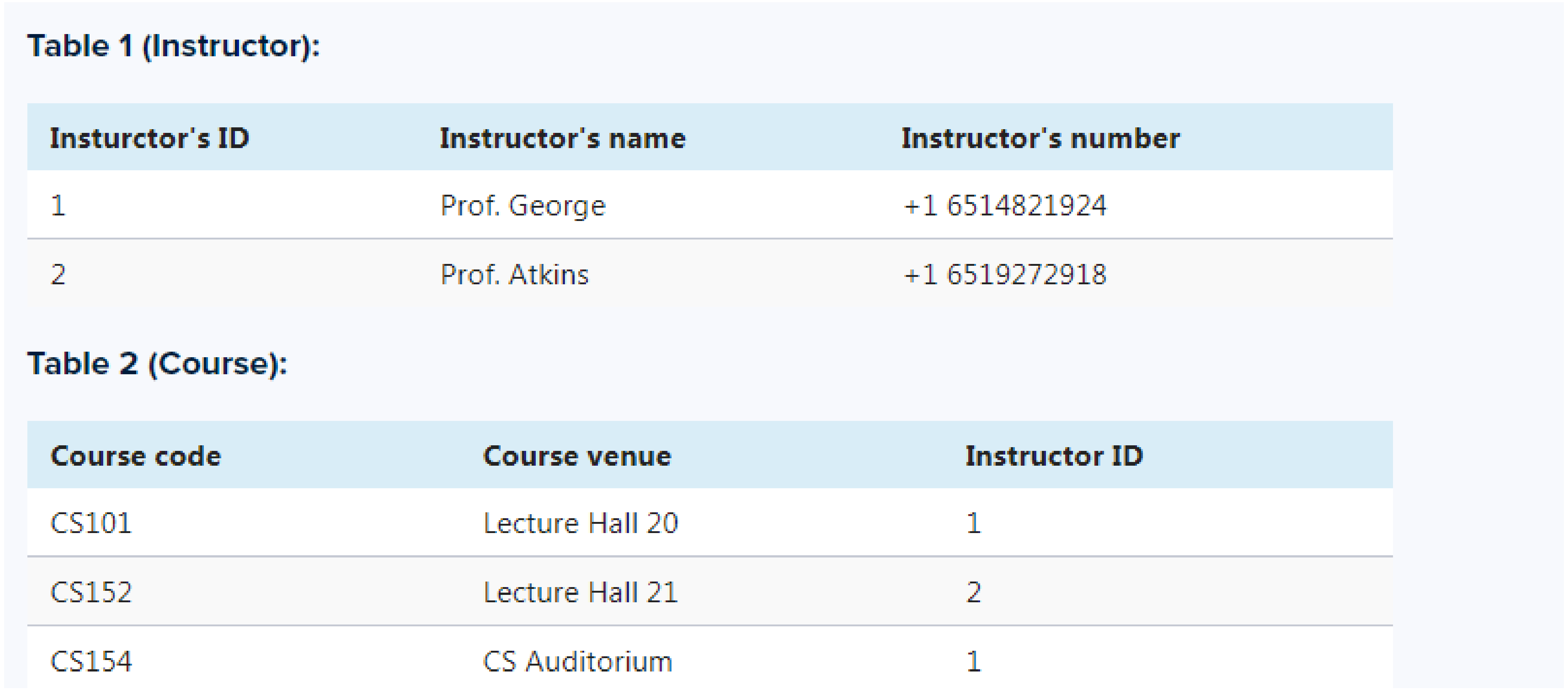


Table 1 (Instructor):

Insturctor's ID	Instructor's name	Instructor's number
1	Prof. George	+1 6514821924
2	Prof. Atkins	+1 6519272918

Table 2 (Course):

Course code	Course venue	Instructor ID
CS101	Lecture Hall 20	1
CS152	Lecture Hall 21	2
CS154	CS Auditorium	1

- Basically, we store the instructors separately and in the course table, we do not store the entire data of the instructor. We rather store the ID of the instructor. Now, if someone wants to know the mobile number of the instructor, he/she can simply look up the instructor table. Also, if we were to change the mobile number of Prof. George, it can be done in exactly one place. This avoids the stale/wrong data problem.

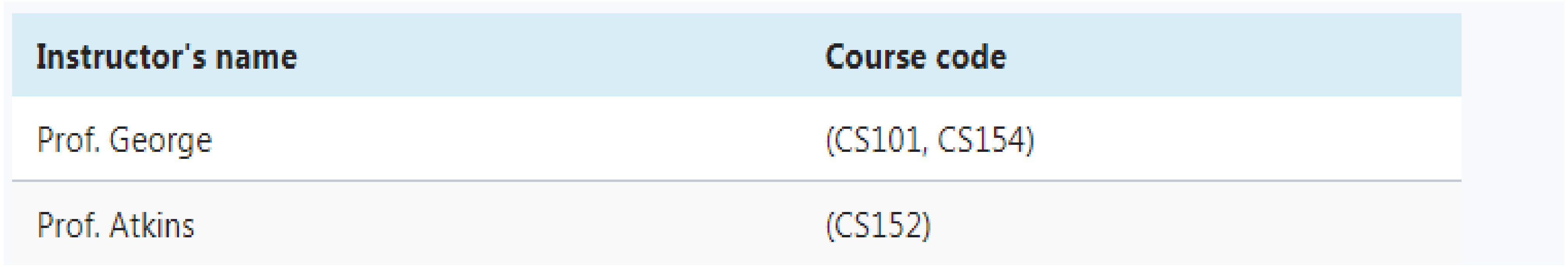
- Further, if you observe, the mobile number now need not be stored 2 times. We have stored it at just 1 place. This also saves storage. This may not be obvious in the above simple example. However, think about the case when there are hundreds of courses and instructors and for each instructor, we have to store not just the mobile number, but also other details like office address, email address, specialization, availability, etc. In such a situation, replicating so much data will increase the storage requirement unnecessarily.
- The above is a simplified example of how database normalization works.

Types of DBMS Normalization

- There are various database “Normal” forms. Each normal form has an importance which helps in optimizing the database to save storage and to reduce redundancies.

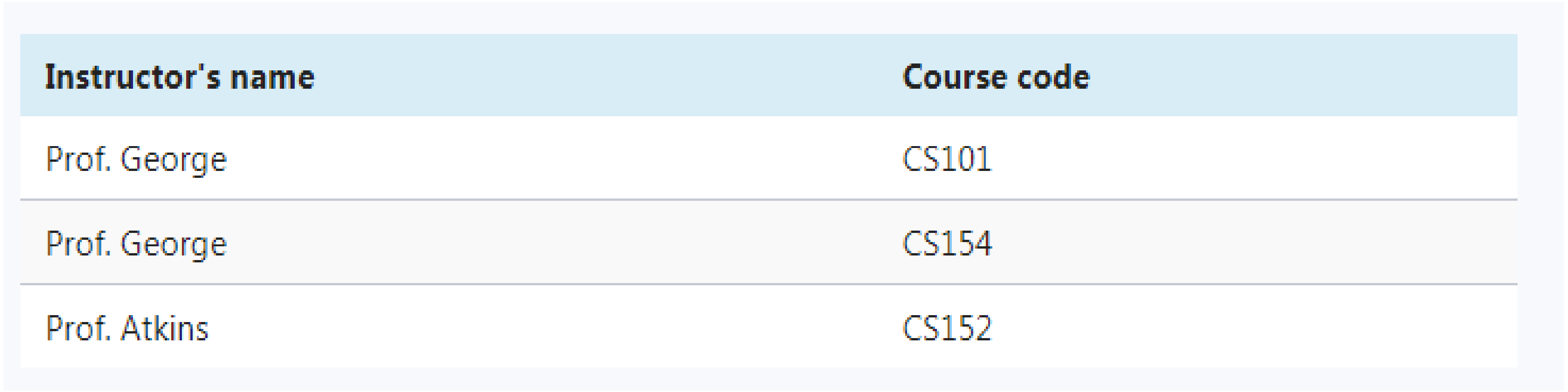
First Normal Form (1NF)

- The First normal form simply says that each cell of a table should contain exactly one value. Let us take an example. Suppose we are storing the courses that a particular instructor takes, we can store it like this:



Instructor's name	Course code
Prof. George	(CS101, CS154)
Prof. Atkins	(CS152)

- Here, the issue is that in the first row, we are storing 2 courses against Prof. George. This isn't the optimal way since that's now how SQL databases are designed to be used. A better method would be to store the courses separately. For instance:

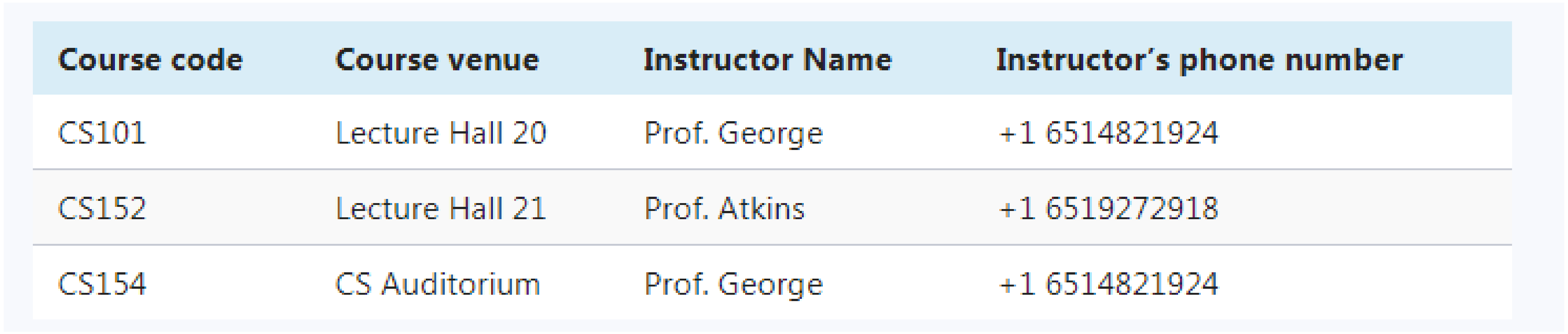


Instructor's name	Course code
Prof. George	CS101
Prof. George	CS154
Prof. Atkins	CS152

- This way, if we want to edit some information related to CS101, we do not have to touch the data corresponding to CS154. Also, observe that each row stores unique information. There is no repetition. This is the First Normal Form.

Second Normal Form (2NF)

- For a table to be in second normal form, the following 2 conditions are to be met:
  1. The table should be in the first normal form.
  2. The primary key of the table should compose of exactly 1 column.
- The first point is obviously straightforward since we just studied 1NF. Let us understand the first point - 1 column primary key. Well, a primary key is a set of columns that uniquely identifies a row. Basically, no 2 rows have the same primary keys. Let us take an example.



Course code	Course venue	Instructor Name	Instructor's phone number
CS101	Lecture Hall 20	Prof. George	+1 6514821924
CS152	Lecture Hall 21	Prof. Atkins	+1 6519272918
CS154	CS Auditorium	Prof. George	+1 6514821924

Here, in this table, the course code is unique. So, that becomes our primary key. Let us take another example of storing student enrollment in various courses. Each student may enroll in multiple courses. Similarly, each course may have multiple enrollments.

- A sample table may look like this (student name and course code):
  - Course code
  - Course code, professor name
  - Course code, professor mobile number
- A superkey whose size (number of columns) is the smallest is called as a candidate key. For instance, the first superkey above has just 1 column. The second one and the last one have 2 columns. So, the first superkey (Course code) is a candidate key.

- Boyce-Codd Normal Form says that if there is a functional dependency A → B, then either A is a superkey or it is a trivial functional dependency. A trivial functional dependency means that all columns of B are contained in the columns of A. For instance, (course code, professor name) → (course code) is a trivial functional dependency because when we know the value of course code and professor name, we do know the value of course code and so, the dependency becomes trivial.

- **Let us understand what’s going on:**

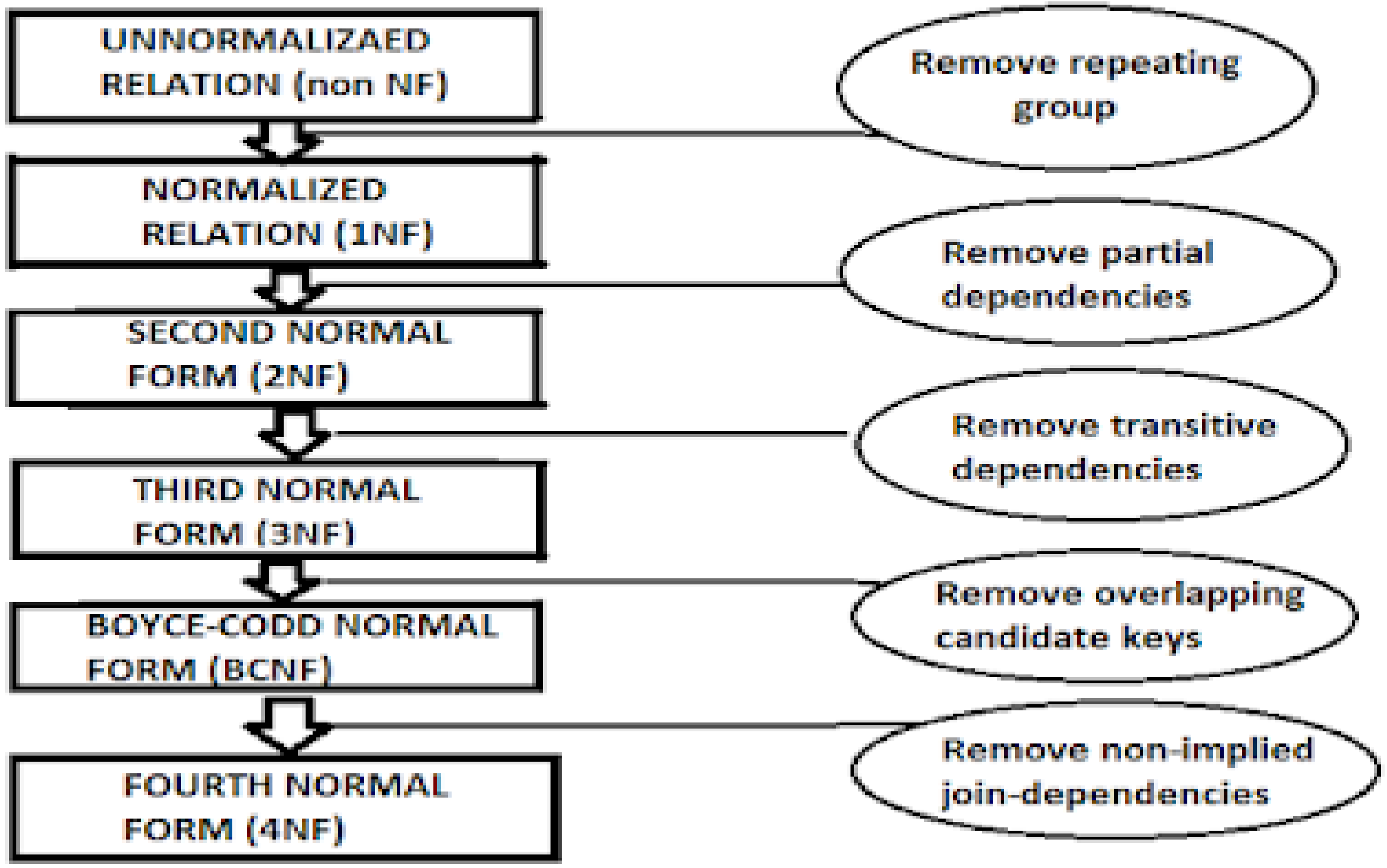
- A is a superkey: this means that only and only on a superkey column should it be the case that there is a dependency of other columns. Basically, if a set of columns (B) can be determined knowing some other set of columns (A), then A should be a superkey. Superkey basically determines each row uniquely.

- It is a trivial functional dependency: this means that there should be no non-trivial dependency.
- For instance, we saw how the professor's department was dependent on the professor's name.
- This may create integrity issues since someone may edit the professor's name without changing the department. This may lead to an inconsistent database.
- There are also 2 other normal forms:

- **Fourth normal form**
- A table is said to be in fourth normal form if there is no two or more, independent and multivalued data describing the relevant entity.

- **Fifth normal form**
- A table is in fifth Normal Form if:
- It is in fourth normal form.
- It cannot be subdivided into any smaller tables without losing some form of information.

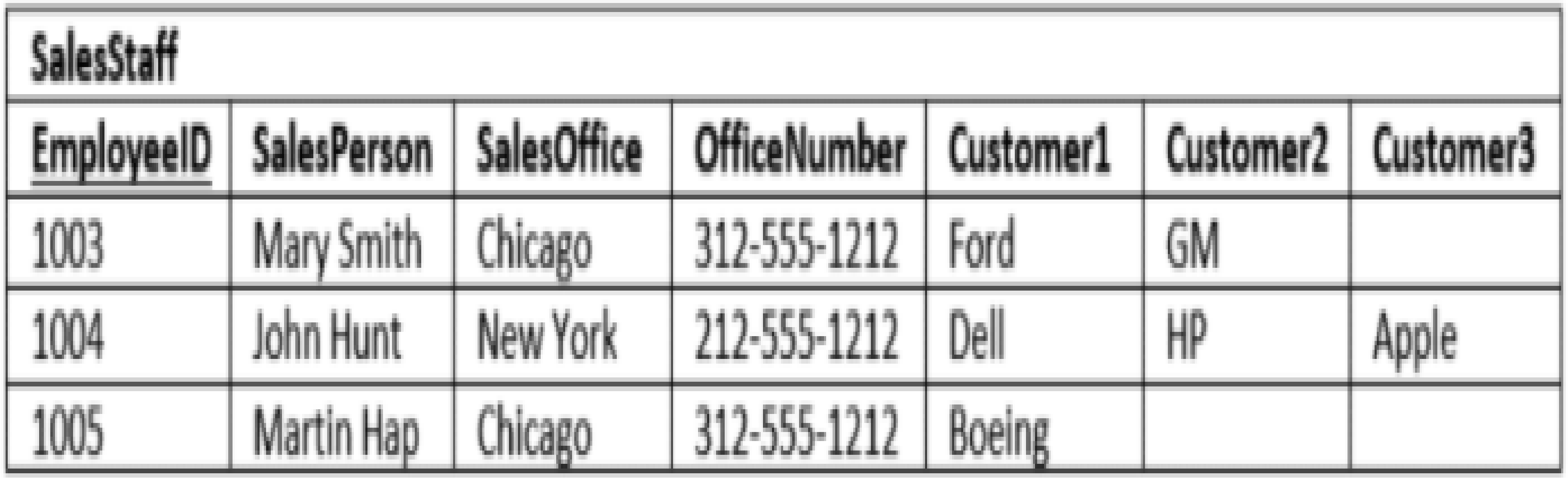
Summary


UNNORMALIZAED RELATION (non NF)
NORMALIZED RELATION (1NF)
SECOND NORMAL FORM (2NF)
THIRD NORMAL FORM (3NF)
BOYCE-CODD NORMAL FORM (BCNF)
FOURTH NORMAL FORM (4NF)
Remove repeating group
Remove partial dependencies
Remove transitive dependencies
Remove overlapping candidate keys
Remove non-implied join-dependencies

- The various forms of database normalization are useful while designing the schema of a database in such a way that there is no data replication which may possibly lead to inconsistencies. While designing the schema for applications, we should always think about how can we make use of these forms.

Practice

- Normalize the following table :

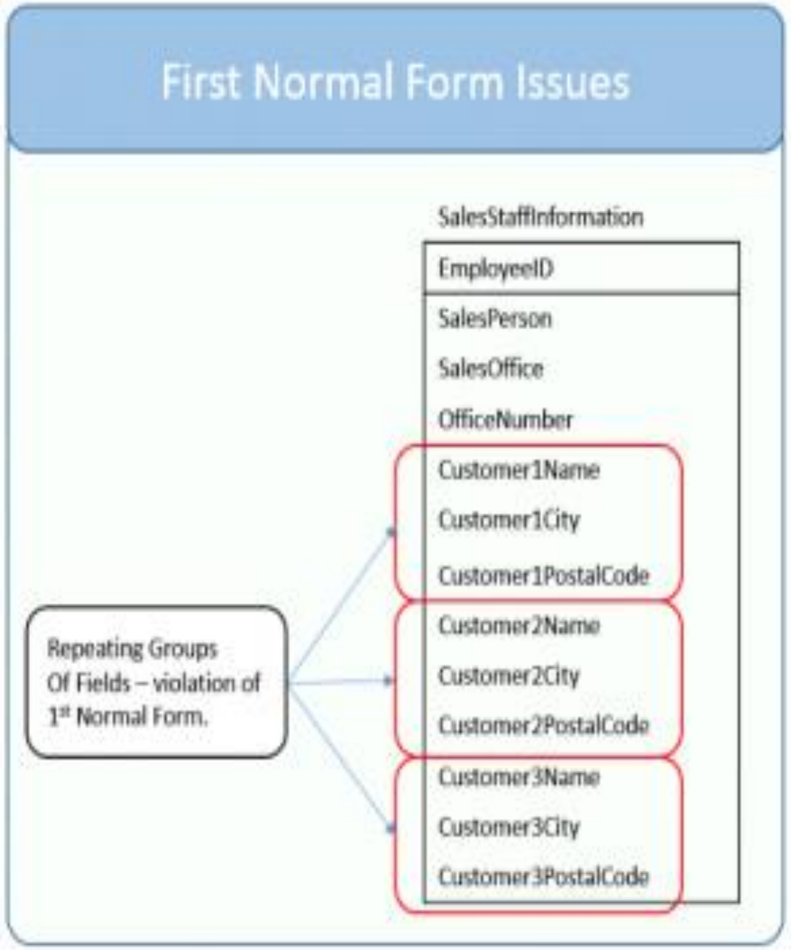


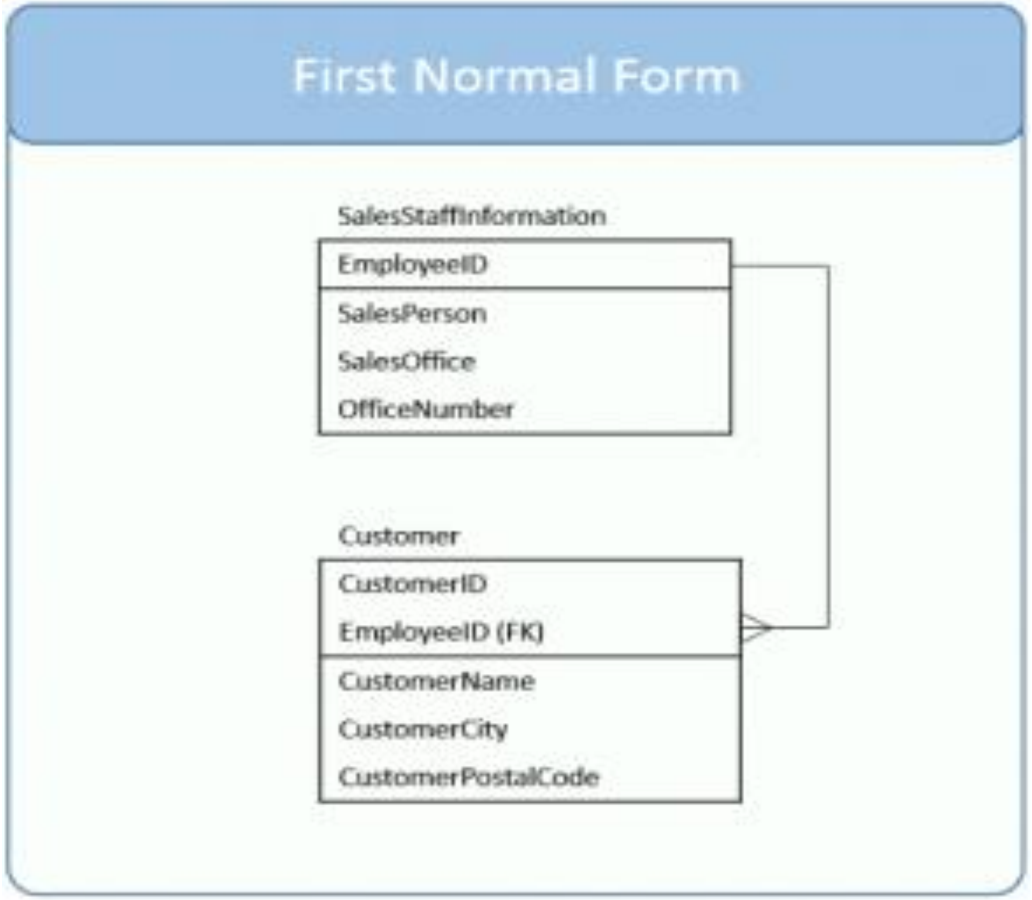
SalesStaff

<u>EmployeeID</u>	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		

Solution

- 1NF





- In this case, the customer table contains the corresponding EmployeeID for the SalesStaffInformation row. Here is our data in the 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

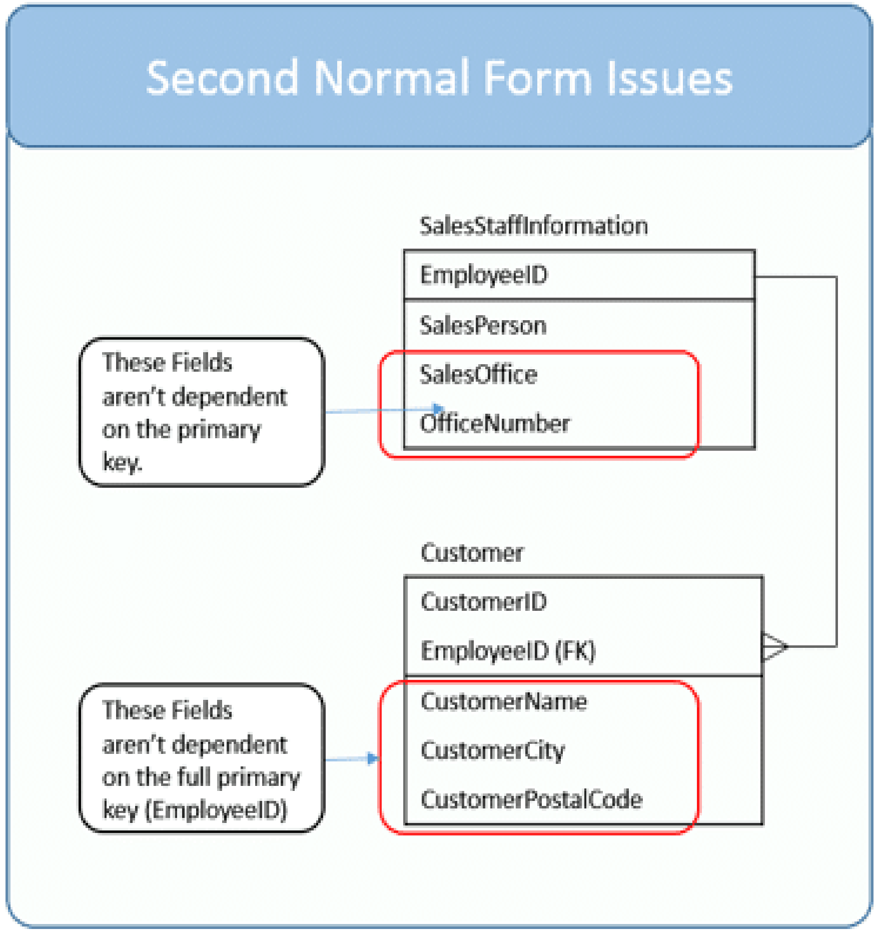
Note: Primary Key: EmployeeID

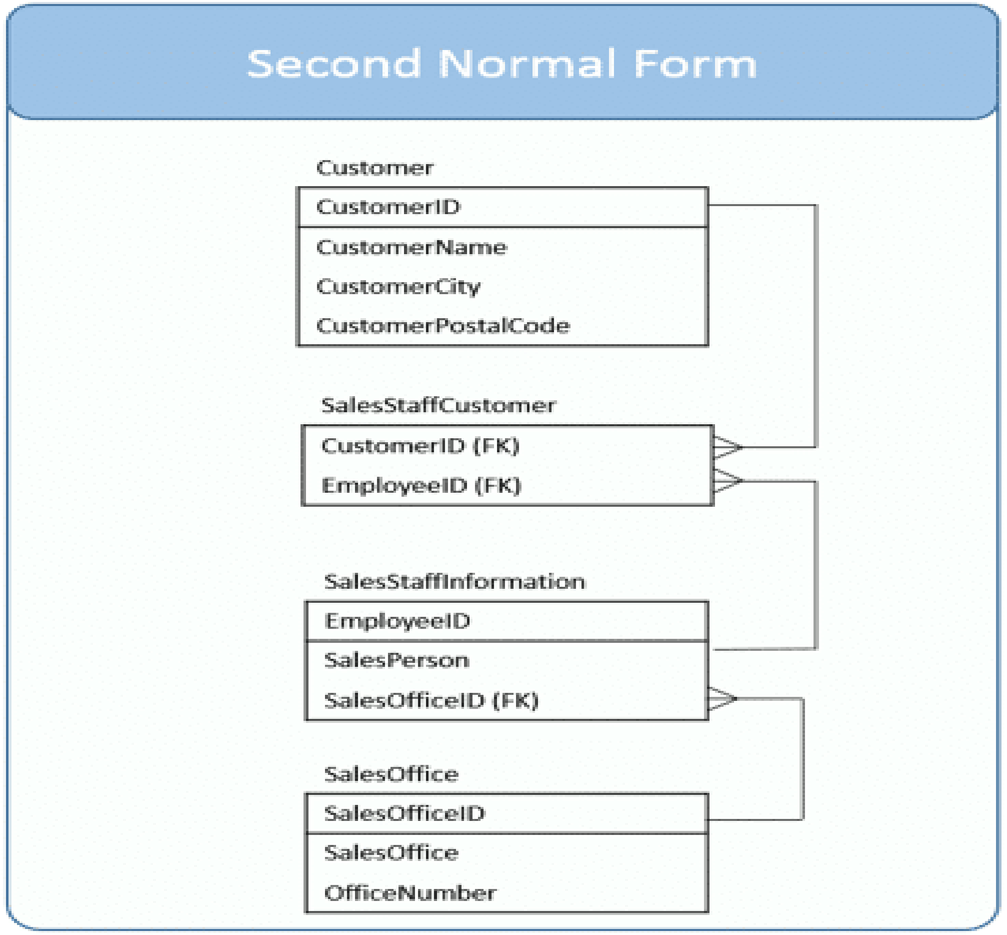
Customer				
CustomerID	**EmployeeID**	**CustomerName**	**CustomerCity**	**PostalCode**
C1000	1003	Ford	Dearborn	48123
C1010	1003	GM	Detroit	48213
C1020	1004	Dell	Austin	78720
C1030	1004	HP	Palo Alto	94303
C1040	1004	Apple	Cupertino	95014
C1050	1005	Boeing	Chicago	60601

- This design is superior to our original table in several ways:
- 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.
- The Customer, which is our original data, is nearly impossible to sort. You could, if you used the UNION statement, but it would be cumbersome. Now, it is simple to sort customers.
- The same holds true for filtering on the customer table. It is much easier to filter on one customer name related column than three.

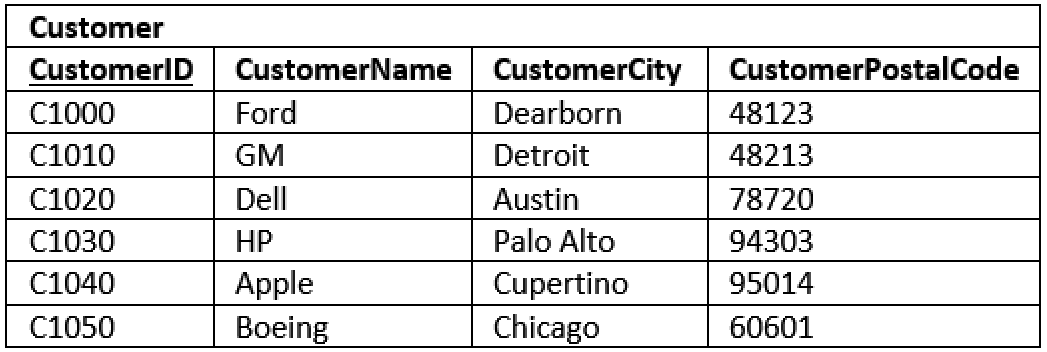
- The insert and deletion anomalies for Customer have been eliminated. You can delete all the customer for a SalesPerson without having to delete the entire SalesStaffInformaiton row.
- Modification anomalies remain in both tables, but these are fixed once we reorganize them as 2nd normal form.

2ND NF

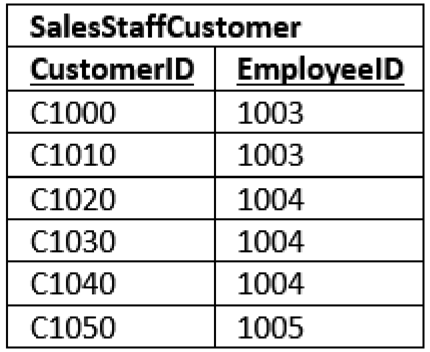

Second Normal Form Issues
SalesStaffInformation
EmployeeID
SalesPerson
SalesOffice
OfficeNumber
These Fields aren't dependent on the primary key.
Customer
CustomerID
EmployeeID (FK)
CustomerName
CustomerCity
CustomerPostalCode
These Fields aren't dependent on the full primary key (EmployeeID)


Second Normal Form
Customer
CustomerID
CustomerName
CustomerCity
CustomerPostalCode
SalesStaffCustomer
CustomerID (FK)
EmployeeID (FK)
SalesStaffInformation
EmployeeID
SalesPerson
SalesOfficeID (FK)
SalesOffice
SalesOfficeID
SalesOffice
OfficeNumber

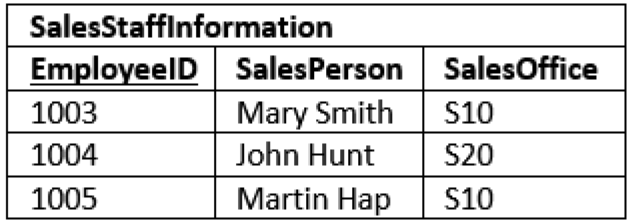
- To better visualize this, here are the tables with data.



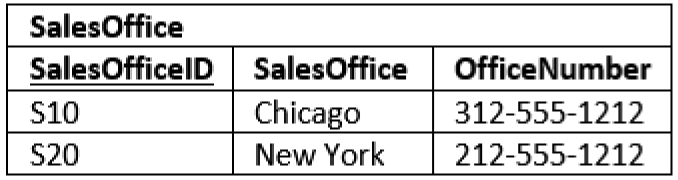
Customer			
CustomerID	CustomerName	CustomerCity	CustomerPostalCode
C1000	Ford	Dearborn	48123
C1010	GM	Detroit	48213
C1020	Dell	Austin	78720
C1030	HP	Palo Alto	94303
C1040	Apple	Cupertino	95014
C1050	Boeing	Chicago	60601



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



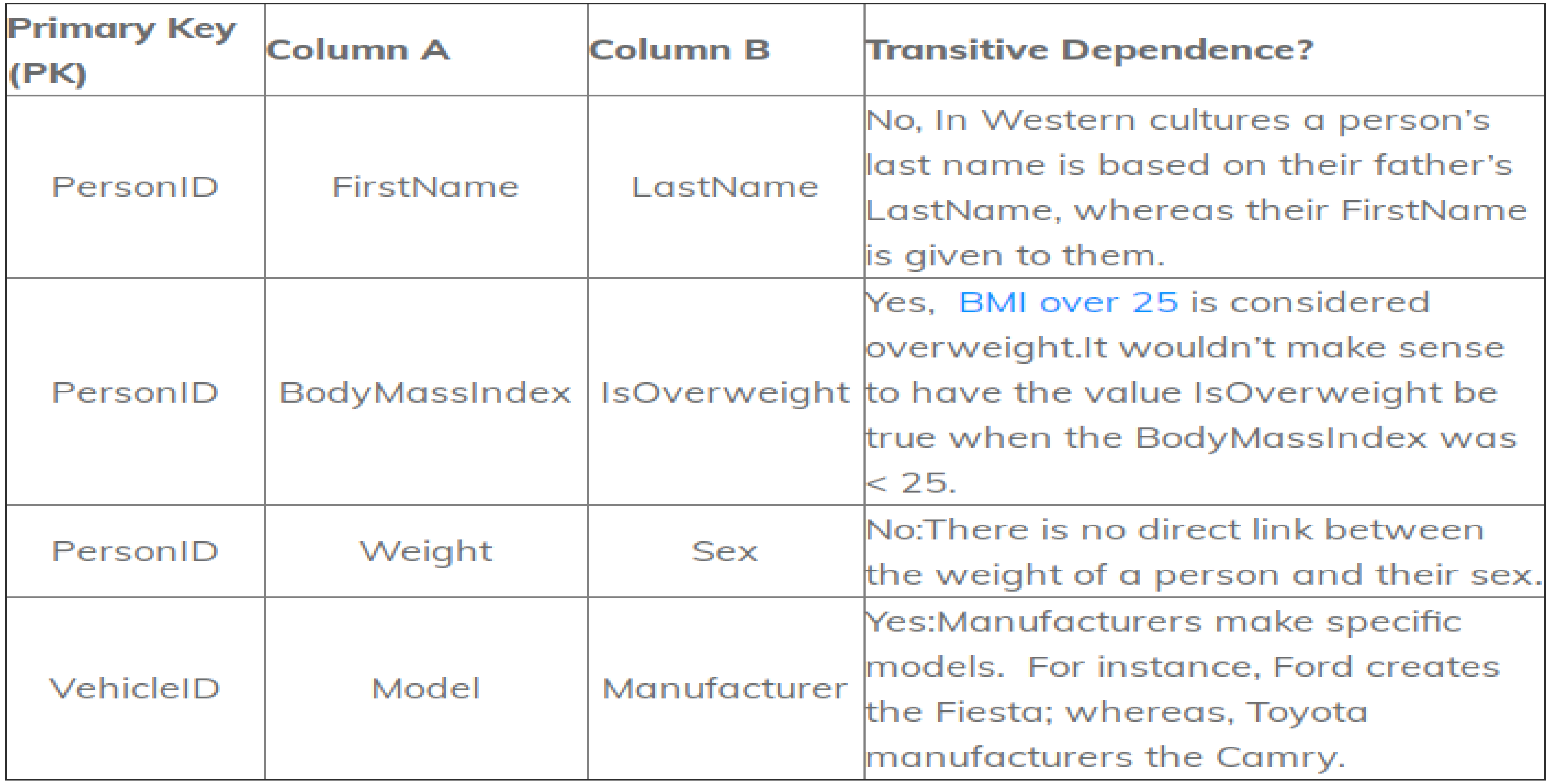
SalesStaffInformation		
EmployeeID	SalesPerson	SalesOffice
1003	Mary Smith	S10
1004	John Hunt	S20
1005	Martin Hap	S10



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

Transitive Dependency

Let's look at some examples to understand further.

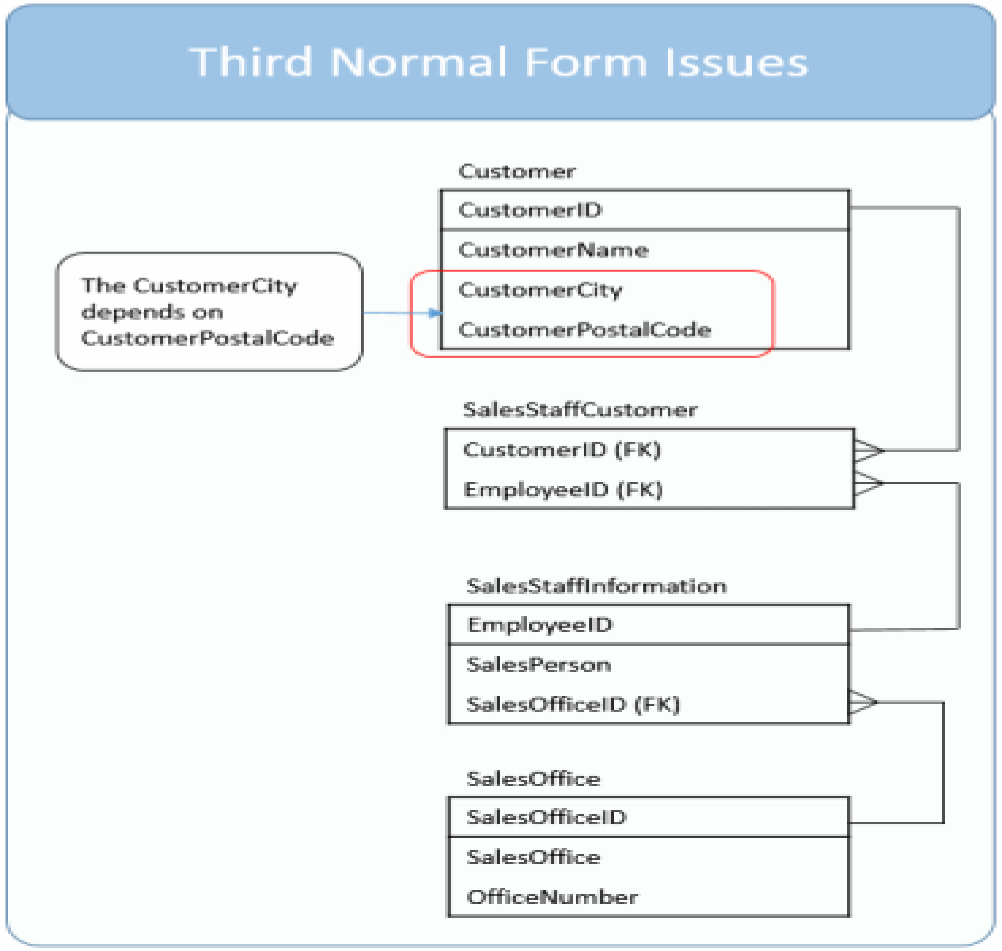


Primary Key (PK)	**Column A**	**Column B**	**Transitive Dependence?**
PersonID	FirstName	LastName	No, In Western cultures a person's last name is based on their father's LastName, whereas their FirstName is given to them.
PersonID	BodyMassIndex	IsOverweight	Yes, BMI over 25 is considered overweight.It 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.

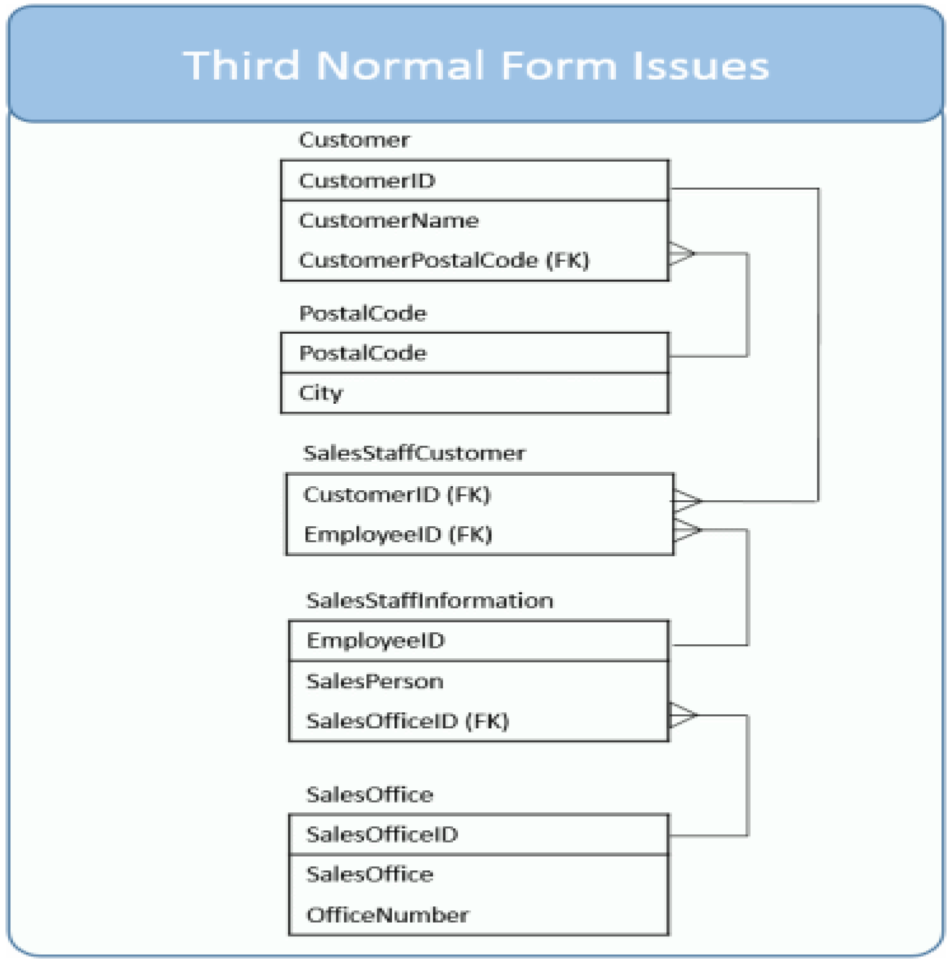
- To be non-transitively dependent, then, means that all the columns are dependent on the primary key (a criteria for 2nd normal form) and no other columns in the table.

3RD NF

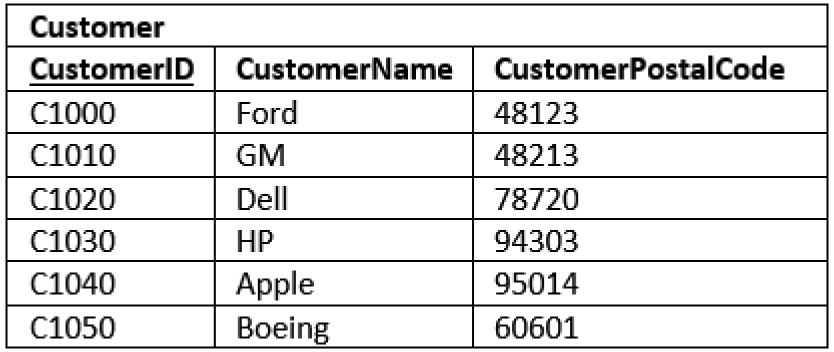
- **CustomerCity relies on CustomerPostalCode which relies on CustomerID**
- Generally speaking a postal code applies to one city. Although all the columns are dependent on the primary key, CustomerID, there is an opportunity for an update anomaly as you could update the CustomerPostalCode without making a corresponding update to the CustomerCity.
- We've identified this issue in red.


Third Normal Form Issues
Customer
CustomerID
CustomerName
CustomerCity
CustomerPostalCode
The CustomerCity depends on CustomerPostalCode
SalesStaffCustomer
CustomerID (FK)
EmployeeID (FK)
SalesStaffInformation
EmployeeID
SalesPerson
SalesOfficeID (FK)
SalesOffice
SalesOfficeID
SalesOffice
OfficeNumber

- **Fix the Model to 3NF Standards**
- In order 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**

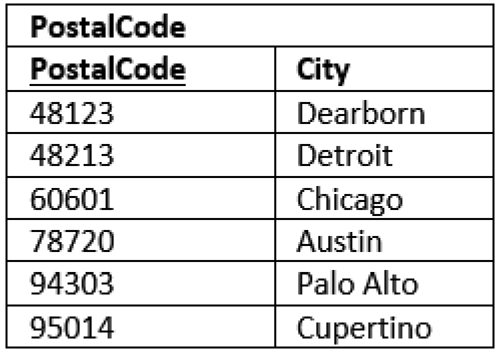

Third Normal Form Issues
Customer
CustomerID
CustomerName
CustomerPostalCode (FK)
PostalCode
PostalCode
City
SalesStaffCustomer
CustomerID (FK)
EmployeeID (FK)
SalesStaffInformation
EmployeeID
SalesPerson
SalesOfficeID (FK)
SalesOffice
SalesOfficeID
SalesOffice
OfficeNumber

- To better visualize this, here are the Customer and PostalCode tables with data.



Customer		
<u>CustomerID</u>	**CustomerName**	**CustomerPostalCode**
C1000	Ford	48123
C1010	GM	48213
C1020	Dell	78720
C1030	HP	94303
C1040	Apple	95014
C1050	Boeing	60601

- Now each column in the customer table is dependent on the primary key. Also, the columns don’t rely on one another for values. Their only dependency is on the primary key.



PostalCode

PostalCode	**City**
48123	Dearborn
48213	Detroit
60601	Chicago
78720	Austin
94303	Palo Alto
95014	Cupertino

- The same holds true for the PostalCode table.
- At this point our data model fulfills the requirements for the third normal form. For most practical purposes this is usually sufficient; however, there are cases where even further data model refinements can take place

- If you are curious to know about these advanced normalization forms, I would encourage you to read about BCNF (Boyce-Codd Normal Form) and more!

Can Normalization Get out of Hand?

- Can database normalization be taken too far? You bet! There are times when it isn't worth the time and effort to fully normalize a database. In our example you could argue to keep the database in second normal form, that the CustomerCity to CustomerPostalCode dependency isn't a deal breaker.
- I think you should normalize if you feel that introducing update or insert anomalies can severely impact the accuracy or performance of your database application. If not, then determine whether you can rely on the user to recognize and update the fields together.
- There are times when you'll intentionally denormalize data. If you need to present summarized or complied data to a user, and that data is very time consuming or resource intensive to create, it may make sense to maintain this data separately.