# 4. Normalisation

## 4.1 Introduction

Suppose we are now given the task of designing and creating a database. How do we produce a good design? What relations should we have in the database? What attributes should these relations have? Good database design needless to say, is important. Careless design can lead to uncontrolled data redundancies that will lead to problems with data anomalies.

In this chapter we will examine a process known as *Normalisation*—a rigorous design tool that is based on the mathematical theory of relations which will result in very practical operational implementations. A properly normalised set of relations actually simplifies the retrieval and maintenance processes and the effort spent in ensuring good structures is certainly a worthwhile investment. Furthermore, if database relations were simply seen as file structures of some vague file system, then the power and flexibility of RDBMS cannot be exploited to the full.

For us to appreciate good design, let us begin by examining some bad ones.

### *4.1.1 A Bad Design*

E.Codd has identified certain structural features in a relation which create retrieval and update problems. Suppose we start off with a relation with a structure and details like:

| Customer details | | | | Transaction details | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **C#** | **Cname** | **Ccity** | **..** | **P1#** | **Date1** | **Qnt1** | **P2#** | **Date2** | | **P9#** | **Date9** | |
| **1** | **Codd** | **London** | **..** | **1** | **21.01** | **20** | **2** | **23.01** | | | | |
| **2** | **Martin** | **Paris** | **..** | **1** | **26.10** | **25** | | | | | | |
| **3** | **Deen** | **London** | **..** | **2** | **29.01** | **20** | | | | | | |

**Figure 4-1**: Simple Structure

This is a simple and straightforward design. It consists of one relation where we have a single tuple for every customer and under that customer we keep all his transaction records about parts, up to a possible maximum of 9 transactions. For every new transaction, we need not repeat the customer details (of name, city and telephone), we simply add on a transaction detail.

However, we note the following disadvantages:

- The relation is wide and clumsy
- We have set a limit of 9 (or whatever reasonable value) transactions per customer. What if a customer has more than 9 transactions?
- For customers with less than 9 transactions, it appears that we have to store null values in the remaining spaces. What a waste of space!
- The transactions appear to be kept in ascending order of P#s. What if we have to delete, for customer Codd, the part numbered 1—should we move the part numbered 2 up (or rather, left)? If we did, what if we decide later to re-insert part 2? The additions and deletions can cause awkward data shuffling.
- Let us try to construct a query to "Find which customer(s) bought P# 2" ? The query would have to access every customer tuple and for each tuple, examine every of its transaction looking for

  (P1# = 2) OR (P2# = 2) OR (P3# = 2) … OR (P9# = 2)

  A comparatively simple query seems to require a clumsy retrieval formulation!

### *4.1.2 Another Bad Design*

Alternatively, why don't we re-structure our relation such that we do not restrict the number of transactions per customer. We can do this with the following structure:

| Transaction | | | | | | |
|---|---|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

This way, a customer can have just any number of Part transactions without worrying about any upper limit or wasted space through null values (as it was with the previous structure). Constructing a query to "Find which customer(s) bought P# 2" is not as cumbersome as before as one can now simply state: P# = 2.

But again, this structure is not without its faults:

- It seems a waste of storage to keep repeated values of Cname, Ccity and Cphone.
- If C# 1 were to change his telephone number, we would have to ensure that we update ALL occurrences of C# 1's Cphone values. This means updating tuple 1, tuple

2 and all other tuples where there is an occurrence of C# 1. Otherwise, our database would be left in an inconsistent state.

- Suppose we now have a new customer with C# 4. However, there is no part transaction yet with the customer as he has not ordered anything yet. We may find that we cannot insert this new information because we do not have a P# which serves as part of the 'primary key' of a tuple. (A primary key cannot have null values).
- Suppose the third transaction has been canceled, i.e. we no longer need information about 20 of P# 1 being ordered on 26 Jan. We thus delete the third tuple. We are then left with the following relation:

| Transaction | | | | | | |
|---|---|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

But then, suppose we need information about the customer "Martin", say the city he is located in. Unfortunately as information about Martin was held in only that tuple and having the entire tuple deleted because of its P# transaction, meant also that we have lost all information about Martin from the relation.

As illustrated in the above instances, we note that badly designed, unnormalised relations waste storage space. Worse, they give rise to the following storage irregularities:

1. *Update anomaly*

Data inconsistency or loss of data integrity can arise from data redundancy/repetition and partial update.

2. *Insertion anomaly*

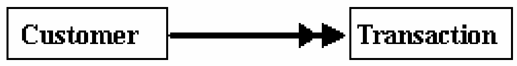
Data cannot be added because some other data is absent.

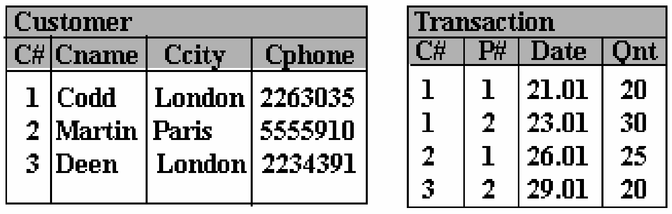
3. *Deletion anomaly*

Data maybe unintentionally lost through the deletion of other data.

### *4.1.3 The Need for Normalisation*

Intuitively, it would seem that these undesirable features can be removed by breaking a relation into other relations with desirable structures. We shall attempt by splitting the above Transaction relation into the following two relations, Customer and Transaction, which can be viewed as entities with a one to many relationship.





| Customer | | | |
|---|---|---|---|
| C# | Cname | Ccity | Cphone |
| 1 | Codd | London | 2263035 |
| 2 | Martin | Paris | 5555910 |
| 3 | Deen | London | 2234391 |

| Transaction | | | |
|---|---|---|---|
| C# | P# | Date | Qnt |
| 1 | 1 | 21.01 | 20 |
| 1 | 2 | 23.01 | 30 |
| 2 | 1 | 26.01 | 25 |
| 3 | 2 | 29.01 | 20 |

**Figure 4-2**: **1:M** data relationships

Let us see if this new design will alleviate the above storage anomalies:

1. *Update anomaly*
If C# 1 were to change his telephone number, as there is only one occurrence of the tuple in the Customer relation, we need to update only that one tuple as there are no redundant/duplicate tuples.

2. *Addition anomaly*
Adding a new customer with C# 4 can be easily done in the Customer relation of which C# serves as the primary key. With no P# yet, a tuple in Transaction need not be created.

3. *Deletion anomaly*
Canceling the third transaction about 20 of P# 1 being ordered on 26 Jan would now mean deleting only the third tuple of the new Transaction relation above. This leaves information about Martin still intact in the new Customer relation.

This process of reducing a relation into simpler structures is the process of *Normalisation*. Normalisation may be defined as a step by step reversible process of transforming an unnormalised relation into relations with progressively simpler structures. Since the process is reversible, no information is lost in the transformation.

Normalisation removes (or more accurately, minimises) the undesirable properties by working through a series of stages called *Normal Forms*. Originally, Codd defined three types of undesirable properties:

1. Data aggregates
2. Partial key dependency
3. Indirect key dependency

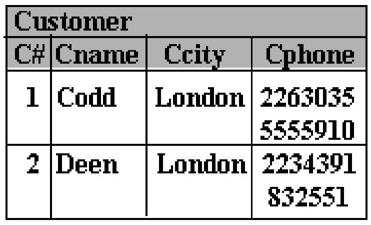
and the three stages of normalisation that remove the associated problems are known, respectively, as the:

- First Normal Form (1NF)
- Second Normal Form (2NF), and
- Third Normal Form (3NF)

We shall now show a more formal process on how we can decompose relations into multiple relations by using the Normal Form rules for structuring.

## 4.2 First Normal Form (1NF)

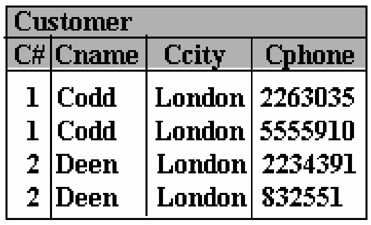
The purpose of the First Normal Form (1NF) is to simplify the structure of a relation by ensuring that it does *not* contain data aggregates or *repeating groups*. By this we mean that no attribute value can have a set of values. In the example below, any one customer has a group of several telephone entries:



| Customer | | | |
|---|---|---|---|
| C# | Cname | Ccity | Cphone |
| 1 | Codd | London | 2263035<br>5555910 |
| 2 | Deen | London | 2234391<br>832551 |

**Figure 4-3**. Presence of Repeating Groups

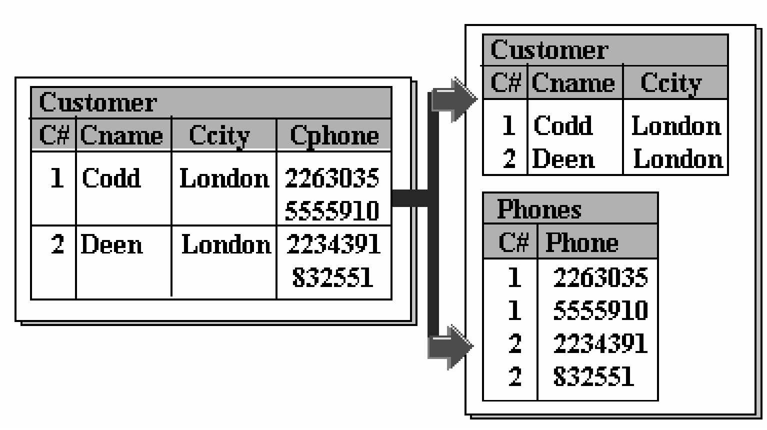
This is thus not in 1NF. It must be "flattened". This can be achieved by ensuring that every tuple defines a single entity by containing only *atomic* values. One can either re-organise into one relation as in:



| Customer | | | |
|---|---|---|---|
| C# | Cname | Ccity | Cphone |
| 1 | Codd | London | 2263035 |
| 1 | Codd | London | 5555910 |
| 2 | Deen | London | 2234391 |
| 2 | Deen | London | 832551 |

**Figure 4-4**: Atomic values in tuples

or split into multiple relations as in:



Customer

| C# | Cname | Ccity | Cphone |
|---|---|---|---|
| 1 | Codd | London | 2263035<br>5555910 |
| 2 | Deen | London | 2234391<br>832551 |

Customer

| C# | Cname | Ccity |
|---|---|---|
| 1 | Codd | London |
| 2 | Deen | London |

Phones

| C# | Phone |
|---|---|
| 1 | 2263035 |
| 1 | 5555910 |
| 2 | 2234391 |
| 2 | 832551 |

**Figure 4-4**: Reduction to 1NF

Note that earlier we defined 1NF as one of the characteristics of a relation (Lesson 2). Thus we consider that *every* relation is at least in the *first normal form* (thus the Figure 4-3 is not even a relation). The Transaction relation of Figure 4-2 is however a 1NF relation.

We may thus generalise by saying that "A relation is in the 1NF if the values in the relation are atomic for every single attribute of the relation".

Before we can look into the next two normal forms, 2NF and 3NF, we need to first explain the notion of 'functional dependency' as these two forms are constrained by functional dependencies.

## 4.3 Functional Dependencies

### *4.3.1 Determinant*

The value of an attribute can uniquely determine the value in another attribute. For example, in every tuple of the Transaction relation in Figure 4-2:

- C# uniquely determines Cname
- C# also uniquely determines Ccity as well as Cphone

Given C# 1, we will know that its Cname is 'Codd' and no other. On the other hand, we cannot say that given Ccity 'London', we will know that its Cname is 'Codd' because

Ccity 'London' will also give Cname of 'Deen'. Thus Ccity cannot uniquely determine Cname (in the same way that C# can).

Additionally, we see that:

- (C#, P#, Date) uniquely determines Qnt

We can now introduce the definition of a "determinant" as being an attribute (or a set of non-redundant) attributes which can act as a unique identifier of another attribute (or another set of attributes) of a given relation.

We may thus say that:

- C# is a unique key for Cname, Ccity and Cphone.
- (C#, P#, Date) is a unique key for Qnt.

These keys are non-redundant keys as no member of the composite attribute can be left out of the set. Hence, C# is a determinant of Cname, Ccity, and Cphone. (C#, P#, Date) is a determinant of Qnt.

A determinant is written as:

$$A \rightarrow B$$

and can be read as "***A*** determines ***B***" (or ***A*** is a determinant of ***B***). If any two tuples in the relation ***R*** have the same value for the ***A*** attribute, then they must also have the same value for their ***B*** attribute.

Applying this to the Transaction relation above, we may then say:

**C# → Cname**

**C# → Ccity**

**C# → Cphone**

**(C#, P#, Date) → Qnt**

The value of the attribute on the left-hand side of the arrow is the determinant because its value uniquely determines the value of the attribute on the right.

Note also that:

**(Ccity, Cphone) → Cname**

**(Ccity, Cphone) → C#**

The converse notation

$$\boldsymbol{A} \xrightarrow{\text{X}} \boldsymbol{B}$$

can be read as ***A*** "does not determine" ***B***.

Taking again the Transaction relation, we may say therefore that Ccity cannot uniquely determine Cname

$$\textbf{Ccity} \quad \xrightarrow{\text{X}} \quad \textbf{Cname}$$

because there exists a number of customers living in the same city.

Likewise:

$$\textbf{Cname} \xrightarrow{\text{X}} \quad \textbf{(Ccity, Cphone)}$$
$$\textbf{Cname} \xrightarrow{\text{X}} \quad \textbf{C\#}$$

as there may exist customers with the same name.

### *4.3.2 Functional Dependence*

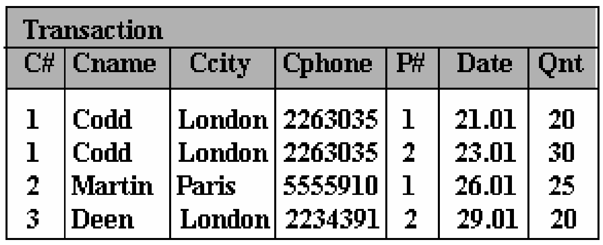
The role of determinants is also expressed as "functional dependencies" whereby we can say:

"If an attribute ***A*** is a determinant of an attribute ***B***, then ***B*** is said to be functionally dependent on ***A***"

and likewise

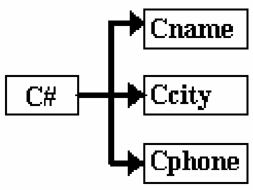
"Given a relation ***R***, attribute ***B*** of ***R*** is functionally dependent on attribute ***A*** if and only if each ***A***-value in ***R*** has associated with it one ***B***-value in ***R*** at any one time".

"C# is a determinant of Cname, Ccity and Cphone" is thus also "Cname, Ccity and Cphone are functionally dependent on C#. Given a particular value of Cname value, there exists precisely one corresponding value for each of Cname, Ccity and Cphone. This is more clearly seen via the following functional dependency diagram:



**Transaction**

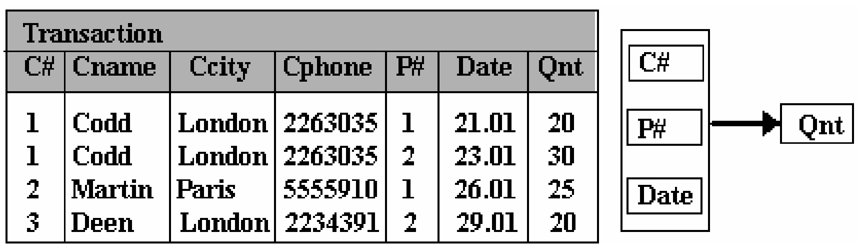
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
|---|---|---|---|---|---|---|
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |



$$\begin{tikzcd} & \boxed{\text{Cname}} \\ \boxed{\text{C\#}} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\text{Ccity}} \\ & \boxed{\text{Cphone}} \end{tikzcd}$$

**Figure 4-5:** Functional dependencies in the Transaction relation

Similarly, "(C#, P#, Date) is a determinant of Qnt" is thus also "Qnt is functionally dependent on the set of attributes (C#, P#, Date)". The set of attributes is also known as a composite attribute.



| Transaction | | | | | | |
|---|---|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

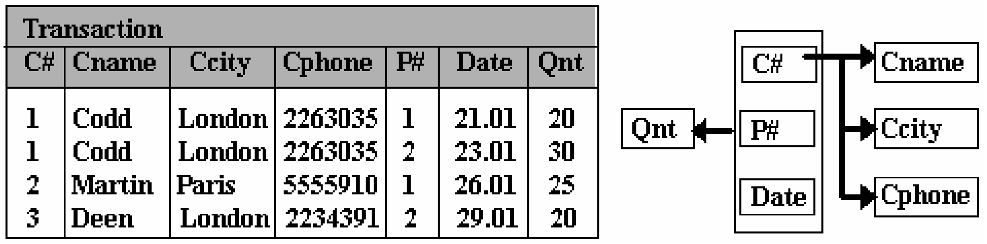
**Figure 4-6:** Functional dependency on a composite attribute

### *4.3.3 Full Functional Dependence*

"If an attribute (or a set of attributes) ***A*** is a determinant of an attribute (or a set of attributes) ***B***, then ***B*** is said to be fully functionally dependent on ***A***"

and likewise

"Given a relation ***R***, attribute ***B*** of ***R*** is *fully functionally* dependent on attribute ***A*** of ***R*** if it is functionally dependent on ***A*** and not functionally dependent on any subset of ***A*** (***A*** must be composite)".



| Transaction | | | | | | |
|---|---|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

**Figure 4-7:** Functional dependencies in the Transaction relation

For the Transaction relation, we may now say that:

- Cname is fully functionally dependent on C#
- Ccity is fully functionally dependent on C#
- Cphone is fully functionally dependent on C#
- Qnt is fully functionally dependent on (C#, P#, Date)
- Cname is not fully functionally dependent on (C#, P#, Date), it is only *partially* dependent on it (and similarly for Ccity and Cphone).

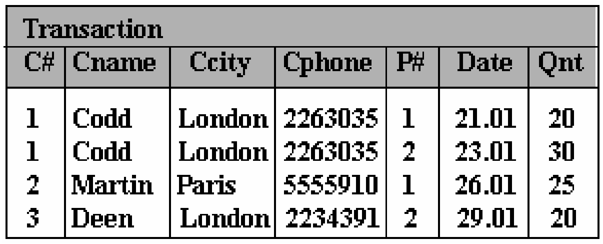
Having understood about determinants and functional dependencies, we are now in a position to explain the rules of the second and third normal forms.

## 4.4 Second Normal Form (2NF)

Consider again the Transaction relation which was in 1NF. Recall the operations we tried to do in Section 4.1.2 above and the problems encountered:

*1. Update*

What happens if Codd's telephone number changes and we update only the first tuple (but not the second)?

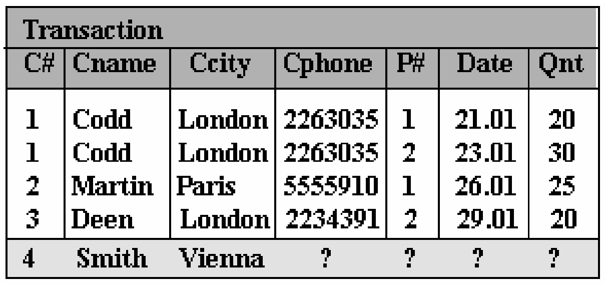


Transaction

| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
|---|---|---|---|---|---|---|
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

*2. Insertion*

If we wish to introduce a new customer, we cannot do so unless an appropriate transaction is effected.



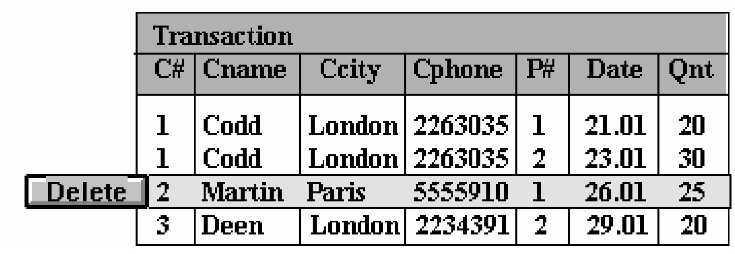
Transaction

| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
|---|---|---|---|---|---|---|
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |
| 4 | Smith | Vienna | ? | ? | ? | ? |

**Entity Integrity Violation: P# is a part of primary key !**

*3. Deletion*

If the data about a transaction is deleted, the information about the customer is also deleted. If this happens to the last transaction for that customer the information about the customer will be lost.

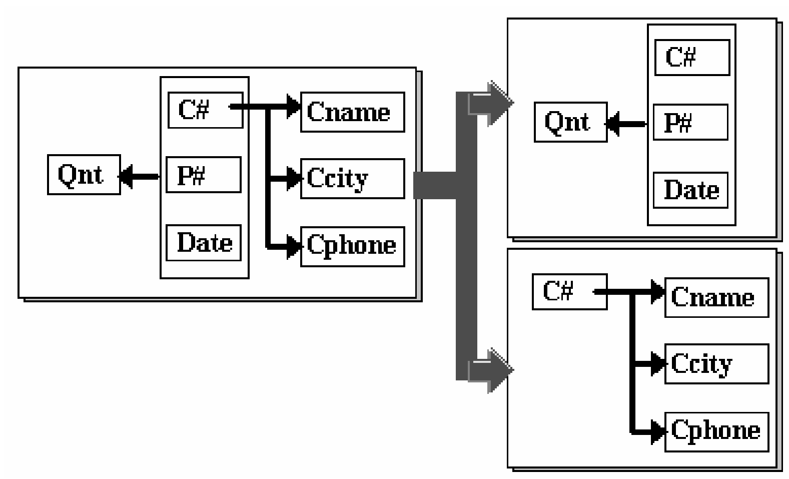


| Transaction | | | | | | |
|---|---|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

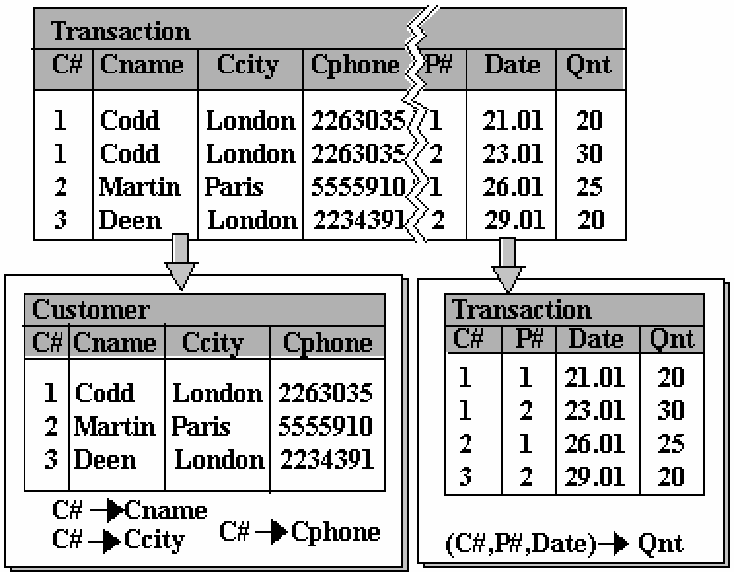
Clearly, the Transaction relation although it is normalised to 1NF still have storage anomalies. The reason for such violations to the database's integrity and consistency rules is because of the *partial dependency on the primary key*.

Recall, the functional dependencies as shown in Figure 4-x. The determinant (C#, P#, Date) is the composite key of the Transaction relation - its value will uniquely determine the value of every other non-key attribute in a tuple of the relation. Note that whilst Qnt is fully functionally dependent on all of (C#, P#, Date), Cname, Ccity and Cphone are only partially functionally dependent on the composite key (as they each depend only on the C# part of the key only but not on P# or Date).

The problems are avoided by eliminating partial key dependence in favour of full functional dependence, and we can do so by separating the dependencies as follows:



The source relation into thus split into two (or more) relations whereby each resultant relation no longer has any partial key dependencies:



| Transaction | | | | | | |
|---|---|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | P# | Date | Qnt |
| 1 | Codd | London | 2263035 | 1 | 21.01 | 20 |
| 1 | Codd | London | 2263035 | 2 | 23.01 | 30 |
| 2 | Martin | Paris | 5555910 | 1 | 26.01 | 25 |
| 3 | Deen | London | 2234391 | 2 | 29.01 | 20 |

| Customer | | | |
|---|---|---|---|
| C# | Cname | Ccity | Cphone |
| 1 | Codd | London | 2263035 |
| 2 | Martin | Paris | 5555910 |
| 3 | Deen | London | 2234391 |

C# → Cname
C# → Ccity
C# → Cphone

| Transaction | | | |
|---|---|---|---|
| C# | P# | Date | Qnt |
| 1 | 1 | 21.01 | 20 |
| 1 | 2 | 23.01 | 30 |
| 2 | 1 | 26.01 | 25 |
| 3 | 2 | 29.01 | 20 |

(C#,P#,Date) → Qnt

**Figure 4-8:** Relations in 2NF

We now have two relations, both of which are in the second normal form. These are the same relations of Figure 4-3 above, and the discussion we had earlier clearly shows that the storage anomalies caused by the 1NF relation have now been eliminated:

1. *Update anomaly*

There are no redundant/duplicate tuples in the relation, thus updates are done just at one place without nay worry for database inconsistencies.

2. *Addition anomaly*

Adding a new customer can be done in the Customer relation without concern whether there is a transaction for a part or not

3. *Deletion anomaly*

Deleting a tuple in Transaction does not cause loss of information about Customer details.

More generally, we shall summarise by stating the following:

1. Suppose, there is a relation ***R***

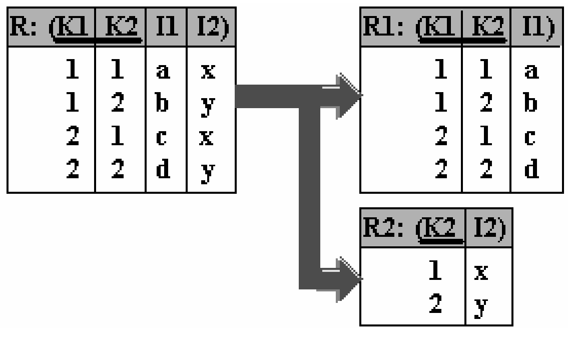
| R: (K1 | K2 | I1 | I2) |
|---|---|---|---|
| 1 | 1 | a | x |
| 1 | 2 | b | y |
| 2 | 1 | c | x |
| 2 | 2 | d | y |

where the composite attribute (K1, K2) is the Primary Key. Suppose also that there exist the following functional dependencies:

**(K1, K2)** $\rightarrow$ **I1** i.e. a full functional dependency on the composite key (K1, K2)..
**K2** $\rightarrow$ **I2** i.e. a partial functional dependency on the composite key (K1, K2).

The partial dependencies on the primary key must be eliminated. The reduction of 1NF into 2NF consists of replacing the 1NF relation by appropriate "projections" such that every non-key attribute in the relations are fully functionally dependent on the primary key of the respective relation. The steps are:

1. Create a new relation R2 from R. Because of the functional dependency **K2** $\rightarrow$ **I2,** R2 will contain K2 and I2 as attributes. The determinant, K2, becomes the key of R2.
2. Reduce the original relation R by removing from it the attribute on the right hand side of **K2** $\rightarrow$ **I2**. The reduced relation R1 thus contain all the original attributes but without I2.
3. Repeat steps 1. and 2. if more than one functional dependency prevents the relation from becoming a 2NF.
4. If a relation has the same determinant as another relation, place the dependent attributes of the relation to be non-key attributes in the other relation for which the determinant is a key.



| R: (K1 | K2 | I1 | I2) |
|---|---|---|---|
| 1 | 1 | a | x |
| 1 | 2 | b | y |
| 2 | 1 | c | x |
| 2 | 2 | d | y |

| R1: (K1 | K2 | I1) |
|---|---|---|
| 1 | 1 | a |
| 1 | 2 | b |
| 2 | 1 | c |
| 2 | 2 | d |

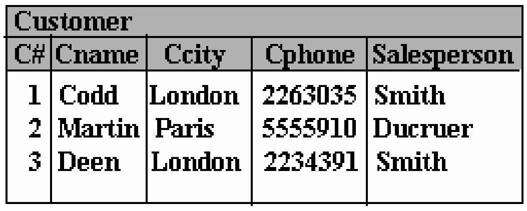
| R2: (K2 | I2) |
|---|---|
| 1 | x |
| 2 | y |

**Figure 4-9:** Reduction of 1NF into 2NF

Thus, "A relation R is in 2NF if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key".

## 4.5 Third Normal Form (3NF)

A relation in the Second Normal Form can still be unsatisfactory and show further data anomalies. Suppose we add another attribute, Salesperson, to the Customer relation who attends to the needs of the customer.



| Customer | | | | |
|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | Salesperson |
| 1 | Codd | London | 2263035 | Smith |
| 2 | Martin | Paris | 5555910 | Ducruer |
| 3 | Deen | London | 2234391 | Smith |

Its associated functional dependencies are:
**C# $\rightarrow$ Cname, Ccity, Cphone, Salesperson**

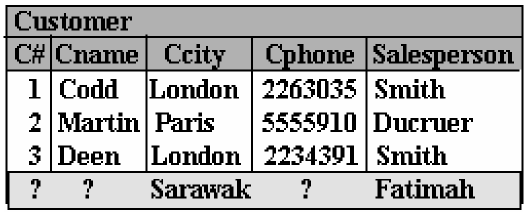
Consider again operations that we may want to do on the data

1. *Update*

Can we change the salesperson servicing customers in London? Here, we find that there are several occurrences of London customers (e.g. Codd and Deen). Thus we must ensure that we update all tuples such that 'Smith' is now replaced by the new salesperson, say 'Jones', otherwise we end up with a database inconsistency problem again.

2. *Insertion*

Can we enter data that 'Fatimah' is the salesperson for the city of 'Sarawak' although no customer exists there yet?



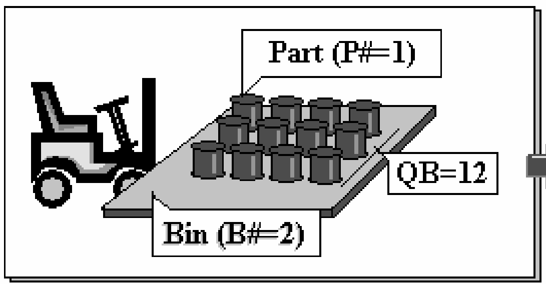
| Customer | | | | |
|---|---|---|---|---|
| C# | Cname | Ccity | Cphone | Salesperson |
| 1 | Codd | London | 2263035 | Smith |
| 2 | Martin | Paris | 5555910 | Ducruer |
| 3 | Deen | London | 2234391 | Smith |
| ? | ? | Sarawak | ? | Fatimah |

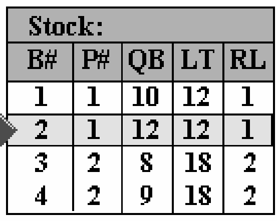
As C# is the primary key, a null value in C# cannot be allowed. Thus the tuple cannot be created.

3. *Deletion*

What happens if we delete the second tuple, i.e. we no longer need to keep information about the customer Martin, his city, telephone and salesperson ? If this tuple is removed, we will also lose all information that the salesperson Ducruet services the city of Paris as no other tuple holds this information.

As another, more complex example, consider keeping information about parts that are being kept in bins, where the following relation called Stock





Stock:

| B# | P# | QB | LT | RL |
|---|---|---|---|---|
| 1 | 1 | 10 | 12 | 1 |
| 2 | 1 | 12 | 12 | 1 |
| 3 | 2 | 8 | 18 | 2 |
| 4 | 2 | 9 | 18 | 2 |

which contains information on:

- the bin number (B#)
- the part number (P#) of the part kept inside a given bin
- the quantity of pieces of the part in a given bin (QB)
- the lead time (LT) taken to deliver a part after an order has been placed for it
- the re-order level (RL) of the part number which indicates the an order must be placed to re-order new stock of that part whenever the existing stock quantity gets too low, i.e. when QB ≤ RL

We further assume that:

- parts of a given part number may be stored in several bins
- the same bin holds only one type of part, i.e. it does not hold parts of more than one part number
- the lead time and re-order level are fixed for each part number

The only candidate key for this relation is B#, hence it must be selected as the primary key. Since the B# is a single attribute, the question of partial dependency does not arise (the relation is in Second Normal Form).

Its associated functional dependencies (which are *full* functional dependencies) are:

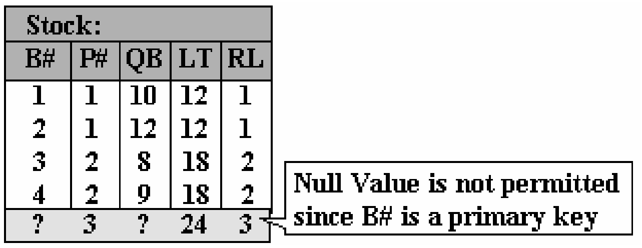
**B# $\rightarrow$ P#, QB, LT, RL**

But in this case, we also have data anomalies:

1. *Update*

Suppose the re-order level for part number 1 is updated, i.e. RL for P# 1 must be changed from 1 to 4. We must ensure that we update all tuples containing P#1, i.e. tuples 1 and 2; any partial updates will lead to an inconsistent database inconsistency.

2. *Insertion*



Stock:

| B# | P# | QB | LT | RL |
|---|---|---|---|---|
| 1 | 1 | 10 | 12 | 1 |
| 2 | 1 | 12 | 12 | 1 |
| 3 | 2 | 8 | 18 | 2 |
| 4 | 2 | 9 | 18 | 2 |
| ? | 3 | ? | 24 | 3 |

Null Value is not permitted since B# is a primary key

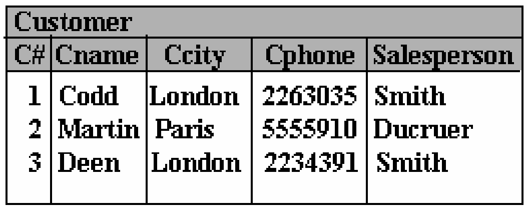
We cannot store LT and RL information for a new expected part number in the database unless we actually have a bin number allocated to it.

3. *Deletion*

If the data (tuple) about a particular bin is deleted, the information about the part is also deleted. If this happens to the last bin containing that part, the information about the concrete part (LT, RL) will also be lost.

From the two examples above, it is still evident that despite having relations in 2NF, problems can still arise and we should now try to eliminate them. It would seem we need to further normalise them, i.e. we need a third normal form to eliminate these anomalies.

Scrutinising the functional dependencies of these examples, we notice the existence of "other" dependencies:



Customer

| C# | Cname | Ccity | Cphone | Salesperson |
|---|---|---|---|---|
| 1 | Codd | London | 2263035 | Smith |
| 2 | Martin | Paris | 5555910 | Ducruer |
| 3 | Deen | London | 2234391 | Smith |

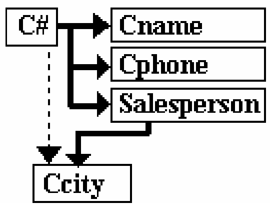


**Figure 4-10**: All functional dependencies in the Customer relation

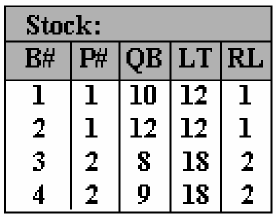
Notice for example that the dependency of the attribute Salesperson on the key C#, i.e.

**C# → Salesperson**
is only an *indirect* or *transitive* dependency, which is also indicated in the diagram as a dotted arrow ⇢ .

This is considered indirect because **C# → Ccity** and **Ccity → Salesperson**, and thus
**C# → Salesperson.**

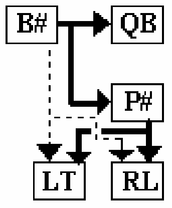
Thus for the Stock relation:
**B# → P#, QB** and **P# → LT, RL**
then **B# → P#, QB, LT, RL**

**Figure 4-11**: All functional dependencies in the Stock relation

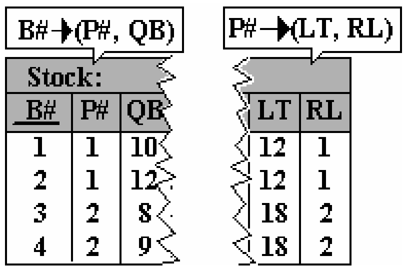


Stock:

| B# | P# | QB | LT | RL |
|---|---|---|---|---|
| 1 | 1 | 10 | 12 | 1 |
| 2 | 1 | 12 | 12 | 1 |
| 3 | 2 | 8 | 18 | 2 |
| 4 | 2 | 9 | 18 | 2 |



The Indirect Dependency obviously causes data duplication (e.g. note the two occurrences of P#1 and LT 12 in the first two tuples of Stock). which leads to the above anomalies. This can be eliminated by *removing all indirect/transitive dependencies*. We do this by splitting the source relation into two or more other relations, as illustrated in the following example:

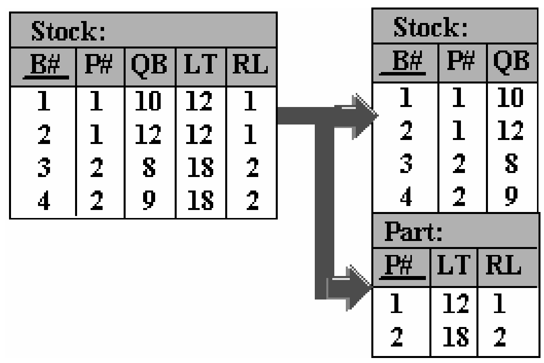


Stock:

| B# | P# | QB |
|---|---|---|
| 1 | 1 | 10 |
| 2 | 1 | 12 |
| 3 | 2 | 8 |
| 4 | 2 | 9 |

| LT | RL |
|---|---|
| 12 | 1 |
| 12 | 1 |
| 18 | 2 |
| 18 | 2 |

where we can then get



Stock:

| B# | P# | QB | LT | RL |
|---|---|---|---|---|
| 1 | 1 | 10 | 12 | 1 |
| 2 | 1 | 12 | 12 | 1 |
| 3 | 2 | 8 | 18 | 2 |
| 4 | 2 | 9 | 18 | 2 |

Stock:

| B# | P# | QB |
|---|---|---|
| 1 | 1 | 10 |
| 2 | 1 | 12 |
| 3 | 2 | 8 |
| 4 | 2 | 9 |

Part:

| P# | LT | RL |
|---|---|---|
| 1 | 12 | 1 |
| 2 | 18 | 2 |

We can say that the storage anomalies caused by the 2NF relation can now be eliminated:

1. *Update anomaly*

To update the re-order level for part number 1, we need only change one (the first) tuple in the new Part relation without concern for other duplicates that used to exist before.

2. *Addition anomaly*

We can now store LT and RL information for a new part number in the database by creating a tuple in the new Part relation, without concern whether there is a bin number allocated to it or not.

3. *Deletion anomaly*

Deleting the tuple about a particular bin will remove a tuple form the new Stock relation. Should the part that was deleted from that bin be the only bin where it could be found, however does not mean the loss of data about that part. Information on the LT and RL of the part is still in the new Part relation.

More generally, we shall summarise by stating the following:

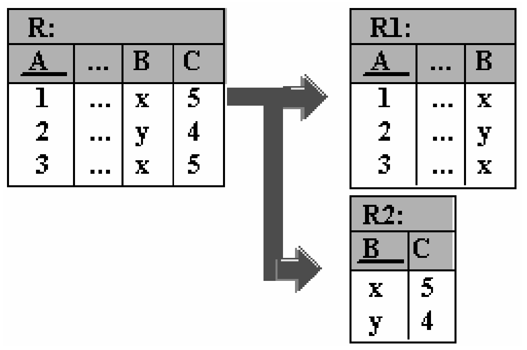
Suppose there is a relation R with attributes A, B and C. A is the determinant.
If $\boldsymbol{A} \rightarrow \boldsymbol{B}$ and $\boldsymbol{B} \rightarrow \boldsymbol{C}$
then
$\boldsymbol{A} \rightarrow \boldsymbol{C}$ is the 'Indirect Dependency'

(Of course, if $\boldsymbol{A} \rightarrow \boldsymbol{C}$ and B does not exist, then $\boldsymbol{A} \rightarrow \boldsymbol{C}$ is a 'Direct Dependency' )

The Indirect Dependencies on the primary key must be eliminated. The reduction of 2NF into 3NF consists of replacing the 2NF relation by appropriate "projections" such Indirect Key Dependencies are eliminated in favour of the Direct Key Dependencies.

The steps are:

1. Reduce the original relation R by removing from it the attribute on the right hand side of any indirect dependencies **A** $\rightarrow$ **C**. The reduced relation R1 thus contain all the original attributes but without C1.
2. Form a new relation R2 that contains all attributes that are in the dependency **B** $\rightarrow$ **C**.
3. Repeat steps 1. and 2. if more than one indirect dependency prevents the relation from becoming a 3NF.
4. Verify that every determinant in every relation is a key in that relation



R:

| A | ... | B | C |
|---|---|---|---|
| 1 | ... | x | 5 |
| 2 | ... | y | 4 |
| 3 | ... | x | 5 |

R1:

| A | ... | B |
|---|---|---|
| 1 | ... | x |
| 2 | ... | y |
| 3 | ... | x |

R2:

| B | C |
|---|---|
| x | 5 |
| y | 4 |

**Figure 4-12.** Reduction of 2NF into 3NF

Thus, "A relation R is in 3NF if it is in 2NF and every non-key attribute is fully and directly dependent on the primary key".

There is another definition of 3NF which states that "A relation is in third normal form if every data item outside the primary key is identifiable by the primary key, the whole primary key and by nothing but the primary key".

In order to avoid certain update anomalies, each relation declared in the data base schema, should be at least in the Third Normal Form. Structurally, 2NF is better than 1NF, and 3NF is better than 2NF. There are of course other higher normal forms like the Boyce-Codd Normal Form (BCNF), the Fourth Normal Form (4NF) and the Fifth Normal Form (5NF).

However, the Third Normal Form is quite sufficient for most business database design purposes, although some very specialised applications may require the higher-level normalisation.
It must be noted that although normalisation is a very important database design component, we should not always design in the highest level of normalisation, thinking that it is the best. Often at the physical implementation level, the decomposition of relations into higher normal form mean more pointer movements are required to access and the thus slower the response time of the database system. This may conflict with the end-user demand for fast performance. The designer may sometimes have to "denormalise" some portions of a database design in order to meet performance requirements at the expense of data redundancy and its associated storage anomalies.