Навчальна дисципліна

БАЗИ ДАНИХ

Лектор - к.т.н., доцент

Баклан Ігор Всеволодович

Site: baklaniv.at.ua

E-mail: iaa@ukr.net

2016-2017

Лекція №4.

Правила цілісності та нормалізація

In order to design high quality databases, you need to be cognizant of the fundamental integrity and normalization rules. We will discuss these rules in this chapter. Sub-topics to be discussed include:

- Fundamental Integrity Rules
- Foreign Key Concept
- Rationale for Normalization
- Functional Dependence and Non-loss Decomposition
- First Normal Form
- Second Normal Form
- Third Normal Form
- Boyce/Codd Normal Form
- Fourth Normal Form
- Fifth Normal Form
- Other Normal Forms
- Summary and Concluding Remarks

4.1 Fundamental Integrity Rules

Two fundamental integrity rules that the database designer must be cognizant of are the *entity integrity rule* and the *referential integrity rule*.

Entity Integrity Rule: The entity integrity rule states that no component of the primary key in a base relation is allowed to accept nulls. Put another way, in a relational model, we never record information about something that we cannot identify. Three points are worth noting here:

- The rule applies to base relations.
- The rule applies to primary key, not alternate keys.
- The primary key must be wholly non-null.

Referential Integrity Rule: The referential integrity rule states that the database must not contain unmatched foreign key values. By unmatched foreign key value, we mean a non-null foreign key value for which there is no match in the referenced (target) relation. Put another way, if B references A then A must exist. The following points should be noted:

- The rule requires that foreign keys must match primary keys, not alternate keys.
- Foreign key and referential integrity are defined in terms of each other. It is not possible to explain one without mentioning the other.

4.2 Foreign Key Concept

The concept of a foreign key was introduced in the previous chapter. Let us revisit this concept, by introducing a more formal definition:

Attribute FK of base relation R2 is a foreign key if and only if (denoted iff from this point) it satisfies the following conditions:

a. Each value of FK is either wholly null or wholly non-null.
b. There exists a base relation, R1 with primary key PK such that each non-null value of FK is identical to the value of PK in some tuple of R1.

We will use the notation R1 → R2 to mean, the relation R1 references the relation R2. In this case, R1 is the referencing (primary) relation and R2 is the referenced relation. Since R1 is the referencing relation, it contains a foreign key. We will also use the notion R1{A, B, C, ...} to mean, the relation R1 contains attributes A, B, C, and so on. Where specific examples are given, the relation-name will be highlighted or placed in upper case; attribute-names of specific examples will not be highlighted when stated with the related relation; however, they will be highlighted when reference is made to them from the body of the text.

Based on the definition of a foreign key, the following consequential points should be noted:

1. The foreign key and the referenced primary key must be defined on the same domain. However, the attribute-names can be different (some implementations of SQL may require that they be identical).
2. The foreign key need not be a component of the primary key for the host (holding) relation (in which case nulls may be accepted, but only with the understanding that they will be subsequently updated).
3. If relations Rn, R(n-1), R(n-2) .... R1 are such that Rn → R(n-1) → R(n-2) → .... R2 → R1 then the chain Rn to R1 forms a *referential path*.
4. A relation can be both referenced and referencing. Consider the referential path R3 → R2 → R1. In this case, R2 is both a referenced and a referencing relation.

5. We can have a self-referencing relation. Consider for example, the relation **Employee**{Emp#, EmpName .... MgrEmployee#} with primary key [Emp#]where attribute **MgrEmp#** is a foreign key defined on the relation **Employee**. In this case we have a self-referencing relation.

6. More generally, a referential cycle exists when there is a referential path from Rn to itself: Rn → R(n-1) → .... R1 → Rn

7. Foreign-to-primary-key matches are said to be the "glue" that holds the database together. As you will see, relations are joined based on foreign keys.

Deletion of Referenced Tuples

Now that we have established the importance of foreign keys, we need to address a question: How will we treat deletion of referenced tuples? Three alternatives exist.

- *Restrict* deletion to tuples that are not referenced.
- *Cascade* deletion to all referencing tuples in referencing relations.
- Allow the deletion but *nullify* all referencing foreign keys in referencing relations.

The third approach is particularly irresponsible, as it could quite quickly put the integrity of the database in question, by introducing *orphan records*. Traditionally, DBMS suites implement the restriction strategy (and for good reasons). The cascading strategy has been surfacing in contemporary systems, as an optional feature. It must be used with much care, as it is potentially dangerous when used without discretion.

4.3 Rationale for Normalization

Normalization is the process of ensuring that the database (conceptual schema) is defined in such a manner as to ensure efficiency and ease of data access. Normalization ensures the following:

- Data integrity
- Control of redundancy
- Logical data independence
- Avoidance of modification anomalies

The following problems can be experienced from having un-normalized files in a system:

- Data redundancy that leads to the modification anomalies
- Modification anomalies which include:
  - ✓ Insertion anomaly: Data cannot be inserted when it is desirable; one has to wait on some future data, due to organization of the data structure
  - ✓ Deletion anomaly: Deletion of some undesirable aspect(s) of data necessarily means deletion of some other desirable aspect(s) of data
  - ✓ Update anomaly: Update of some aspect(s) of data necessarily means update of other aspect(s) of data
- Inefficient file manipulation; lack of ease of data access
- Inhibition to the achievement of logical data independence
- Compromise on the integrity of data
- Pressure on programming effort to make up for the poor design

Figure 4-1 indicates the six most commonly used *normal forms*. The hierarchy is such that a relation in a given normal form is automatically in all normal forms prior to it. Thus a relation in the second normal form (2NF) is automatically in the first normal form (1NF); a relation in the third normal form (3NF) is in 2NF and so on. Edgar Frank Codd defined the first three normal forms in the early 1970s; the Boyce-Codd normal form (BCNF) was subsequently deduced from his work. The fourth and fifth normal forms (4NF and 5NF) were subsequently defined by Ronald Fagin in the late 1970s.

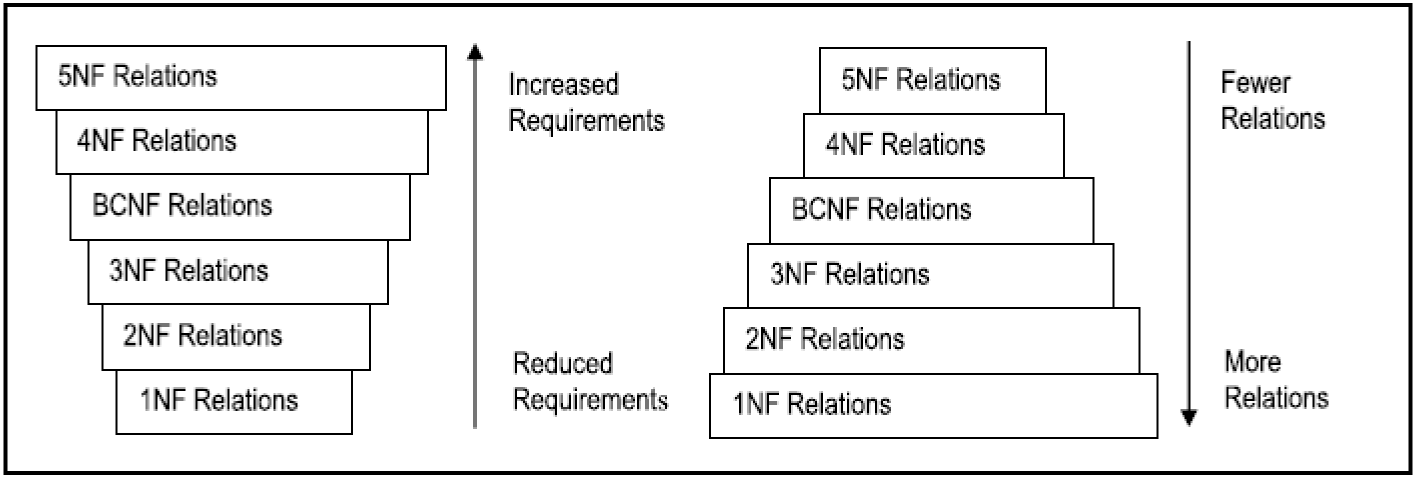


***Figure 4-1.** Normal Forms*

The *normalization procedure* involves decomposing relations into other relations of repeatedly higher normal forms. The process is reversible. Moreover, normalization must be conducted in a manner that ensures that there is no loss of information.

4.4 Functional Dependence and Non-loss Decomposition

Before discussion of the normal forms, we need to define and clarify two fundamental concepts: *functional dependence* and *non-loss decomposition*.

4.4.1 Functional Dependence

Given a relation, R{A, B, C, …}, then attribute B is *functionally dependent* on attribute A, written A → B (read as "A determines B") if and only if (denoted *iff* from this point onwards) each value of A in R has precisely one B-value in R at any point in time. Attributes A and B may or may not be composite.

An alternate way to describe functional dependence (FD) is as follows: Given a value of attribute A, one can deduce a value for attribute B since any two tuples which agree on A must necessarily agree on B.

Example 1:

In relation **Employee** {Emp#, S-Name, F-Name, Address, …}, the following FD holds:
Emp# → S-Name, F-Name, Address

From definition of primary key, all attributes of a relation are functionally dependent on the primary key. This is precisely what is required; in fact, an attribute (or group of attributes) qualifies as a candidate key iff all other attributes of the entity are dependent on it.

We need to further refine our concept of FD by introducing another term — *full functional dependence*: Attribute B is said to be *fully functionally dependent* on attribute A if it is functionally dependent on A and not functionally dependent on any proper subset of A.

As a spinoff from the definition of functional dependence, please note the following:

1. FD constraints have similarities with referential constraints, except that here, reference is internal to the relation.
2. FDs help us to determine primary keys.
3. Each FD defines a *determinant* in a relation: the attribute(s) on the right are dependent on the attribute(s) on the left; the attribute(s) on the left constitute(s) a determinant.

4.4.2 Non-loss Decomposition

Suppose we have a relation **R0** as follows: **R0**{Suppl#, SuplName, Item#, ItemName, Quantity, SuplStatus, Location}

Functional dependencies of **R0** are illustrated in Figure 4-2; they may also be listed as follows:

- [Suppl# , Item#] → {Quantity, SuplName, SuplStatus, Location, ItemName}
- Suppl# → {SuplName, SuplStatus, Location}
- Item# → ItemName

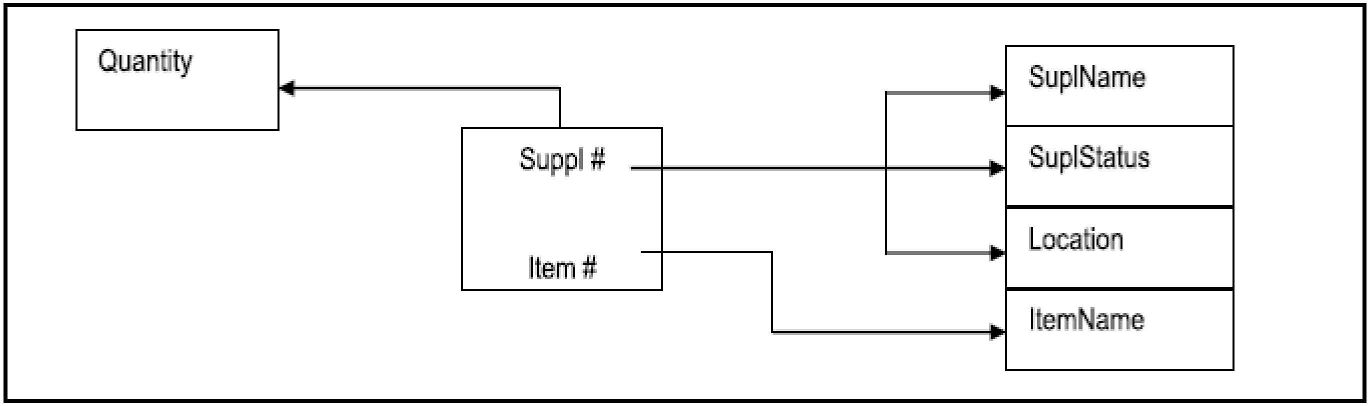


Figure 4-2. *FD Diagram for Relation R0*

Storing **R0** this way causes duplication. The reason is that **R0** is not sufficiently normalized. As an alternate, we could have the following:

R1{Supl#, SuplNam, Location, SuplStatus}

R2{Item#, ItemName}

R3{Supl#, Item#, Quantity}

R1, **R2**, and **R3** constitute (an example of) a *non-loss decomposition* (NLD) of **R0**.

Here is a formal definition of an NLD:

> If R is a relation and P1, P2, ... Pn are projections on R such that
> P1 JOIN P2 JOIN ..... JOIN Pn = R,
> then P1, P2, ... Pn constitutes a non-loss decomposition of R.

Notice that we have glided into two new terms, *projection* and *join*. These will be formerly treated later in the course. Suffice it now to say that a projection on a relation is an extraction (into a new relation) of some attributes of the relation; a join requires at least two relations and may be construed as the opposite of a projection: If you can project R into P1 and P2, then you may be able to join P1 and P2 to yield R.

Given this definition, we need to address the following questions:

1. How do we find non-loss decompositions?
2. When should we replace a relation by a non-loss decomposition?
3. What are the advantages?

Heath's theorem (of [Heath, 1971]) addresses questions (1) and (2). The answer to the third question is stated in section 4.3 above. Heath's theorem is stated below:

If we have a relation R{A, B, C, ...} and if A→B and B→ C, then projections P1{A, B} and P2{B, C} constitute a non-loss decomposition of R.

Example 2: Proof of Heath's Theorem:

We wish to show that R0 {A, B, C} = P1 {A, B} JOIN P2 {B, C}.

Let P1 {A, B} and P2 {B, C} be projections of R0 {A, B, C}
Assume further that A,B,C are single attributes.

Suppose that (a, b, c) is a tuple in R0.
Then (a, b) is in P1 and (b, c) is in P2.
So (a, b, c) is in P1 JOIN P2 .... (1)

Suppose that (a, b, c) is in P1 JOIN P2.
Then (a, b, c1) is in R0 for some value c1
and (a1, b, c) is in R0 for some value a1.
But B → C therefore b → c so that c1 must be c.
Therefore (a, b, c) is in R0 .... (2)

We have shown any tuple (a,b,c) that is in R0 is also in P1 JOIN P2, and that any tuple (a,b,c) that is in P1 JOIN P2 is also in R0. Therefore R0 = P1 JOIN P2.

Corollary of Heath's Theorem

An important corollary from Heath's theorem is as follows:

> If P1, P2, ... Pn is a non-loss decomposition of R and relations P1, P2, ... Pn all share a candidate key, then there is no reduction in data duplication.

Example 3: The following example illustrates the importance of the above-mentioned corollary:

> Suppose that a relation **Student**{SID, SName, Grade, Dept} is decomposed into **S1**{SID, SName} and **S2**{SID, Grade, Dept}.
>
> Assume further that **SID** is the primary key (or at least a candidate key) of **Student**. Note that **SID** also occurs in S1 and S2. It should be obvious that there is no point in proceeding with this decomposition as it simply compounds the duplication problem (**SID** would now be stored in two relations rather than one, to no avail).
>
> Also, decomposition of **Student** into S3{SID, SName} and S4{Grade, Dept} makes no sense.

Conclusion

Based on Heath's theorem and its corollary, we can assert with confidence, the following advice:

- Decompose only when there is a non-loss decomposition such that the resulting relations do not share a candidate key.
- Do not decompose if each resulting relation does not have a primary key.

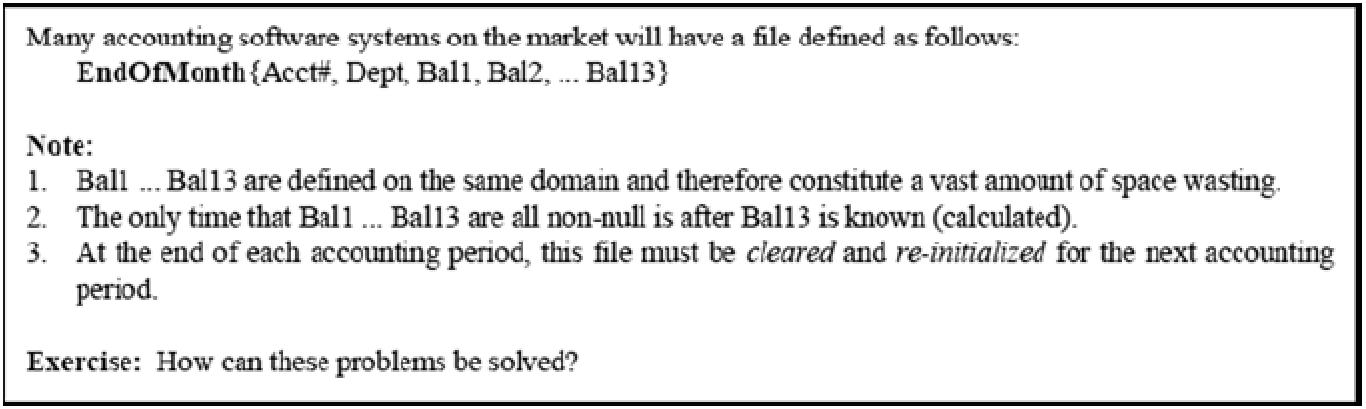
4.5 The First Normal Form

A relation R is in the first normal form (1NF) iff it is a flat file i.e. it has no repeating groups, no duplicate records, no null values in the primary key.

Put another way, a relation is in 1NF iff all its underlying simple domains (hence attributes) are atomic, i.e. for every tuple in the relation, each attribute can have only one type of value.

By definition, all relations are in 1NF. This is by no means coincidental, but by design: we defined a relation to consist of atomic attributes, and subject to the entity integrity constraint and the referential integrity constraint. However, as you will soon see, having relations in 1NF only is often not good enough.

Example 4:



Many accounting software systems on the market will have a file defined as follows:

EndOfMonth{Acct#, Dept, Bal1, Bal2, ... Bal13}

Note:

1. Bal1 ... Bal13 are defined on the same domain and therefore constitute a vast amount of space wasting.
2. The only time that Bal1 ... Bal13 are all non-null is after Bal13 is known (calculated).
3. At the end of each accounting period, this file must be *cleared* and *re-initialized* for the next accounting period.

Exercise: How can these problems be solved?

Problems with Relations in 1NF Only

Relation **R0** of the previous section is in 1NF only. However it is undesirable to store it as is due to a number of problems. In the interest of clarity, the relation is restated here:

R0{Supl#, SuplName, Item#, ItemName, Quantity, SuplStatus, Location}

Functional dependencies of **R0** as illustrated in Figure 4-2 are as follows:

- FD1: [Suppl#, Item#] → {Quantity, SuplName, SuplStatus, Location, ItemName}
- FD2: Suppl# → {SuplName, SuplStatus, Location}
- FD3: Item# → ItemName

The following data anomalies exist with **R0** (and most relations in 1NF only):

- **Replication of data:** Every time we record a supplier - item pair, we also have to record supplier name and item name.
- **Insertion anomaly**: We cannot insert a new item until it is supplied; neither can we insert a new supplier until that supplier supplies some item.
- **Deletion anomaly**: We cannot delete an item or a supplier without destroying an entire shipment, as well as information about a supplier's location.
- **Update anomaly**: If we desire to update a supplier's location or item name, we have to update several records, in fact, an entire shipment, due to the duplication problem.

Insertion, deletion update anomalies constitute *modification anomalies,* caused by duplication of data due to improper database design.

4.6 The Second Normal Form

A relation is in the second normal form (2NF) iff it is in 1NF and every *non-key attribute* is fully functionally dependent on the primary key.

By non-key attribute, we mean that the attribute is not part of the primary key. Relation R0 (of the previous section), though in 1NF, is not in 2NF, due to FD2 and FD3. Using Heath's theorem, we may decompose relation R0 as follows (note that the abbreviation PK is used to denote the primary key):

R1{Supl#, Sname, Location, SuplStatus} PK[Suppl#]

R2{Item#, Itemname} PK[Item#]

R3{Supl#, Item#, Qty} PK[Supl#, Item#]

We then check to ensure that the resulting relations are in 2NF (and they are).

So based on the definition of 2NF, and on the authority of Heath's theorem, we would replace **R0** with **R1**, **R2**, and **R3**. Please note the consequences of our treatment of **R0** so far:

1. The problems with relations in 1NF only have been addressed.
2. By decomposing, we have introduced foreign keys in relation R3.
3. JOINing is the opposite of PROJecting. We can rebuild relation R0 by simply JOINing R3 with R1 and R3 with R2, on the respective foreign keys.
4. From the definition of 2NF, two observations should be obvious: Firstly, if you have a relation with a single attribute as the primary key, it is automatically in 2NF. Secondly, if you have a relation with *n* attributes and *n-1* of them form the primary key, the relation is also in 2NF.

Problems with Relations in 2NF Only

In this example, relations **R2** and **R3** are in 2NF (in fact they are in 3NF*)*, but *we* still have potential problems with **R1**: What if we have a situation where there may be several suppliers from a given location? Or what if we want to keep track of locations of interest? In either case, we would have modification anomalies as described below:

- Insertion anomaly: We cannot record information about a location until we have at least one supplier from that location.
- Deletion anomaly: We cannot delete a particular location without also deleting supplier(s) from that location.
- Update anomaly: If we wish to update information on a location, we have to update all supplier records from that location.

These problems can be addressed if we take the necessary steps to bring R1 into the third normal form (3NF). But first, we must define what 3NF is.

4.7 The Third Normal Form

> A relation is in the third normal form (3NF) iff it is in 2NF and no non-key attribute is fully functionally dependent on other non-key attribute(s).
>
> Put another way, a relation is in 3NF iff non-key attributes are *mutually independent* and fully functionally dependent on the primary key. (Two or more attributes are mutually independent if none of them is functionally dependent on any combination of the other.)
>
> Put another way, a relation is in 3NF iff it is in 2NF and every non-key attribute is *non-transitively* dependent on the primary key. (Non-transitivity implies mutual independence.)

Transitive dependence refers to dependence among non-key attributes. In particular, if A → B and B → C, then C is transitively dependent on A (i.e. A→ C transitively).

In the previous section, relation R1 is problematic because it is not in 3NF. If it is desirable to store additional information about the locations as indicated in the previous section, then we must be smart enough to discern that location is to be treated as an entity with attributes such as location code, location name (and perhaps others). Using Heath's theorem, we may therefore decompose R1 as follows:

R4{Supl#, Sname, LocationCode} PK[Supl#]

R5{LocationCode, LocationName} PK[LocationCode]

We now check to ensure that the relations are in 3NF (and they are). Again, please take careful notice of the consequences of our actions to this point:

1. The problems with relations in 2NF only have been addressed.
2. Again, by decomposing, we have introduced a foreign key in relation **R4**.
3. We can rebuild relation **R1** by simply JOINing **R4** with **R5** on the foreign key.
4. From the definition of 3NF, it should be obvious that if you have a relation with one candidate key and *n* mutually independent non-key attributes, or only one non-key attribute, it is in 3NF.

Problems with Relations in 3NF Only

Relations **R2**, **R3**, **R4**, and **R5** above are all in 3NF. However, it has been found that 3NF-only relations suffer from certain inadequacies. It is well known that 3NF does not deal satisfactorily with cases where the following circumstances hold:

- There are multiple composite candidate keys in a relation.
- The candidate keys overlap (i.e. have at least one attribute in common).

For these situations, the Boyce-Codd normal form (BCNF) provides the perfect solution. As you shall soon see, the BCNF is really a refinement of 3NF. In fact, where the above-mentioned conditions do not hold, BCNF reduces to 3NF.

4.8 The Boyce-Codd Normal Form

Simply, Boyce-Codd normal form (BCNF) requirement states:

A relation is in BCNF iff every *determinant* in the relation is a candidate key.

A determinant is an attribute (or group of attributes) on which some other attribute(s) is (are) fully functionally dependent. Examination of **R2**, **R3**, **R4**, and **R5** above will quickly reveal that they are in BCNF (hence 3NF). We therefore need to find a different example that illustrates the importance of BCNF.

Consider the situation where it is desirous to keep track of animals in various zoos, and the assigned keepers for these animals. Let us tentatively construct the relation **R6** as shown below:

R6{Zoo, Animal, Keeper}

Assume further that that a keeper works at one and only one zoo. We can therefore identify the following FDs:

- [Zoo, Animal] → Keeper
- Keeper → Zoo

Given the above, we conclude that [Zoo, Animal] is the primary key. Observe that **R6** is in 3NF but not in BCNF, since **Keeper** is not a candidate key but is clearly a determinant. Using Heath's theorem, we may decompose **R6** as follows:

R7{Animal, Keeper} PK[Animal]

R8{Keeper, Zoo} PK[Keeper]

As on previous occasions, let us examine the consequences of our action:

1. By achieving BCNF, we benefit from further reduction in data duplication, and modification anomalies.
2. A further advantage is that we can now store *dangling records*. In our example, a keeper can be assigned to a zoo even before he/she is assigned an animal.
3. One possible drawback with BCNF is that more relations have to be accessed (joined) in order to obtain useful information. Again referring to the example, **R7** must be joined with **R8** in order to derive Zoo-Animal pairs.

Observe: The principle of BCNF is very simple but profound. By being guided by it, you can actually bypass obtaining 1NF, 2NF and 3NF relations, and move directly into a set of BCNF relations. Adopting this approach will significantly simplify the analysis process. Moreover, in most practical situations, you will not be required to normalize beyond BCNF. This approach will be further clarified in the next chapter.

4.9 The Fourth Normal Form

The fourth normal form (4NF) relates to the situation where mutually independent, but related attributes form a relation and the inefficient arrangement causes duplication and hence modification anomalies. Consider the database file, **CTT-Schedule**, representing course-teacher-text combinations in an educational institution. Assume the following:

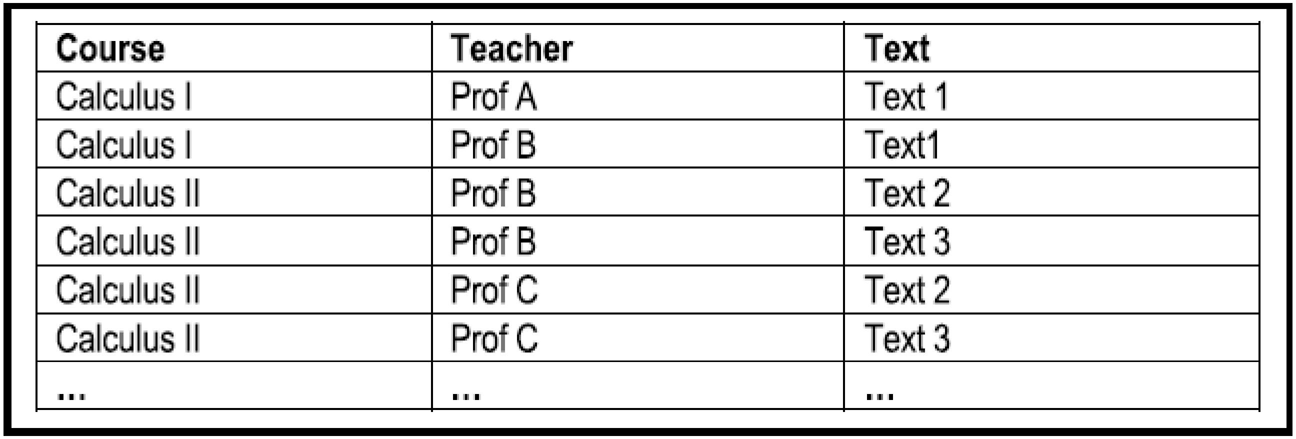
a. A course can be taught by several teachers.

b. A course can require any number of texts.

c. Teachers and texts are independent of each other i.e. the same texts are used irrespective of who teaches the course.

d. A teacher can teach several courses.

Figure 4-3 provides some sample data for the purpose of illustration.



Course	Teacher	Text
Calculus I	Prof A	Text 1
Calculus I	Prof B	Text1
Calculus II	Prof B	Text 2
Calculus II	Prof B	Text 3
Calculus II	Prof C	Text 2
Calculus II	Prof C	Text 3
...	...	...

Figure 4-3. *CTT-Schedule File*

Note that the theory so far, does not provide a method of treating such a situation, except flattening the structure (by making each attribute part of the primary key) as shown below:

R9{Course, Teacher, Text} PK[Course, Teacher, Text]

Since **R9** is keyed on all its attributes, it is in BCNF. Yet, two potential problems are data redundancy and modification anomalies (the former leading to the latter). In our example, in order to record that Calculus II is taught by both Professor B and Professor C, four records are required. In fact, if a course is taught by p professors and requires n texts, the number of records required to represent this situation is p*n. This is extraordinary, and could prove to be very demanding on storage space.

Relation **R9**, though in BCNF, is not in 4NF, because it has a peculiar dependency, called a *multi-valued dependency* (MVD). In order to state the 4NF, we must first define MVD.

4.9.1 Multi-valued Dependency

A multi-valued dependency (MVD) is defined as follows:

> Given a relation R(A, B, C), the MVD A -» B (read "A multi-determines B") holds iff every B-value matching a given (A-value, C-value) pair in R depends only on the A-value and is independent of the C-value.
>
> Further, given R(A B C), A -» B holds iff A -» C also holds. MVDs always go together in pairs like this. We may therefore write A -» B/C.

Please note the following points arising from the definition of an MVD:

1. For MVD, at least three attributes must exist.
2. FDs are MVDs but MVDs are not necessarily FDs.
3. A -» B reads "A multi-determines B" or "B is multi-dependent on A."

Let us get back to **R9: Course -» Text/Teacher**. Note that **Course** is the pivot of the MVD. **Course -» Teacher** since **Teacher** depends on Course, independent of Text. **Course -» Text** since **Text** depends on **Course**, independent of **Teacher**.

4.9.2 Fagin's Theorem

Fagin's theorem (named after Ronald Fagin who proposed it) may be stated as follows:

> Relation R{A, B, C} can be non-loss decomposed into projections R1{A, B} and R2{A, C} iff the MVDs A -» B/C both hold.

Note that like Heath's theorem, which prescribes how to treat FDs, Fagin's theorem states exactly how to treat MVDs. With this background, we can proceed to defining the 4NF:

> A relation is in 4NF iff whenever there exists an MVD, say A -» B, then all attributes of R are also functionally dependent on A.
>
> Put another way, R{A, B, C…} is in 4NF iff every MVD satisfied by R is implied by the candidate key of R.
>
> Put another way, R{A, B, C…} is in 4NF iff the only dependencies are of the form [candidate key] → [other non-key attribute(s)].
>
> Put another way, R{A, B, C ...}is in 4NF iff it is in BCNF and there are no MVD's (that are not FDs).

In the current example, **R9** is not in 4NF. This is so because although it is in BCNF, an MVD exists. Using Fagin's theorem, we may decompose it as follows:

R10{Course, Text} PK[Course, Text]

R11{Course, Teacher} PK[Course, Teacher]

Note: Fagin's theorem prescribes a method of decomposing a relation containing an MVD that is slightlydifferent from the decomposition of an FD as prescribed by Heath's theorem: Figure 4-4 clarifies this.

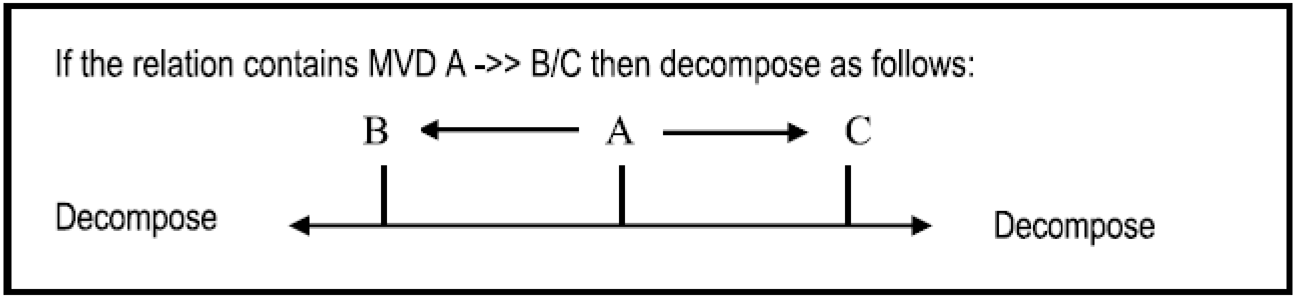

If the relation contains MVD A ->> B/C then decompose as follows:
B
A
C
Decompose
Decompose

Figure 4-4. *Treating MVDs*

4.10 The Fifth Normal Form

So far we have been treating relations that are decomposable into two other relations. In fact, there are relations which cannot be so decomposed, but can be decomposed into n other relations where n > 2. They are said to be *n-decomposable* relations (n > 2). The fifth normal form (5NF) is also commonly referred to as the *projection-join normal form* (PJNF) because it relates to these (*n* > 2) projections (of a relation not in 5NF) into decompositions that can be rejoined to yield the original relation.

Recall the **SupplierSchedule** relationship (linking suppliers, inventory items and projects) mentioned in section 3.5; it is represented here as outlined below:

> **SupplierSchedule**{Suppl#, Item#, Proj#} PK[Suppl#, Item#, Proj#]

The relation represents a M:M relationship involving **Suppliers**, **Items**, and **Projects**. Observe the following features about the relation:

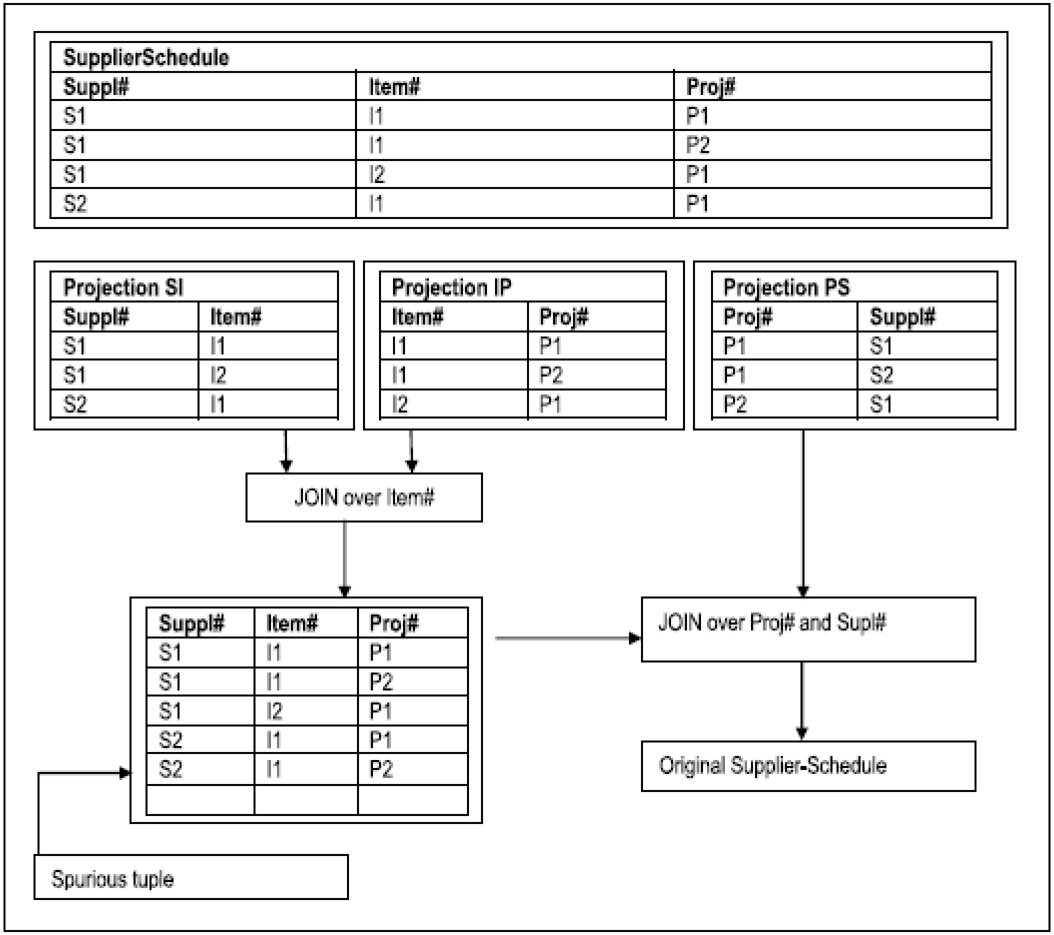
1. **SupplierSchedule** is keyed on all attributes and therefore by definition, is in BCNF. By inspection, it is also in 4NF.

2. It is not possible to decompose this relation into two other relations.

3. If there are *S* suppliers, N items and J projects, then theoretically, there may be up to S*N*J records. Not all of these may be valid.

4. If we consider S suppliers, each supplying N items to J projects, then it does not take much imagination to see that a fair amount of duplication will take place, despite the fact that the relation is in 4NF.

Let us examine a possible decomposition of **SupplierSchedule** as shown in Figure 4-5. If we employ the first two decompositions only, this will not result in a situation that will guarantee us the original **SupplierSchedule**. In fact, if we were to join these two decompositions (**SI** and **IP**), we would obtain a false representation of the original relation. The third projection (**PS**) is absolutely necessary, if we are to have any guarantee of obtaining the original relation after joining the projections.



SupplierSchedule		
Suppl#	Item#	Proj#
S1	I1	P1
S1	I1	P2
S1	I2	P1
S2	I1	P1

Projection SI	
Suppl#	Item#
S1	I1
S1	I2
S2	I1

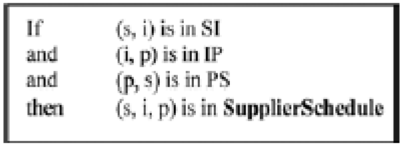
Projection IP	
Item#	Proj#
I1	P1
I1	P2
I2	P1

Projection PS	
Proj#	Suppl#
P1	S1
P1	S2
P2	S1

Suppl#	Item#	Proj#
S1	I1	P1
S1	I1	P2
S1	I2	P1
S2	I1	P1
S2	I1	P2

Figure 4-5. Illustrating Possible Decompositions of Supplier-Schedule

Note: The first join produces **SupplierSchedule** plus additional *spurious* tuples. The effect of the second join is to eliminate the spurious tuples. To put it into perspective, **SupplierSchedule** is subject to a (time independent)
3-decomposable (3D) constraint, namely:



If	(s, i) is in SI
and	(i, p) is in IP
and	(p, s) is in PS
then	(s, i, p) is in **SupplierSchedule**

This is an example of a *join dependency (JD) constraint*.

4.10.1 Definition of Join Dependency

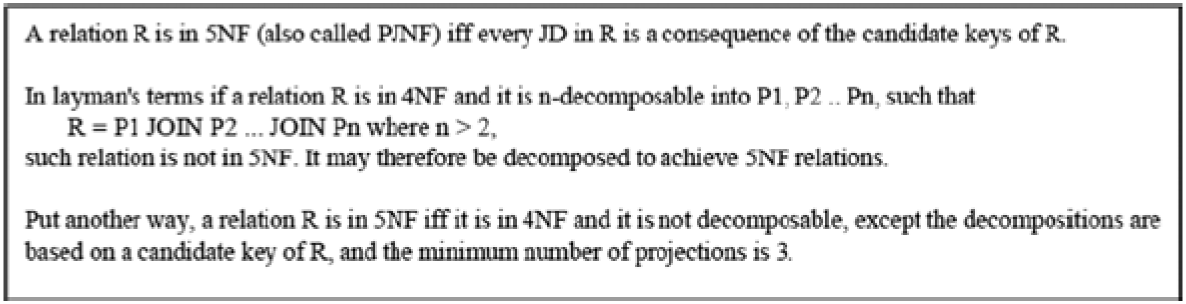
A join dependency (JD) constraint may be defined as follows:

> Relation R satisfies the JD P1, P2, ... Pn iff R = P1 JOIN P2 JOIN ... JOIN Pn
> where the attributes of P1 ... Pn are subsets of the attributes of R.

Relations that are in 4NF, but not in 5NF (such as **SupplierSchedule**) suffer from duplication, which in turn leads to modification anomalies. These problems are directly related to the presence of the JD constraint(s) in such relations. Fagin's theorem for 5NF relations provides the solution.

4.10.2 Fagin's Theorem

Fagin's theorem for the fifth normal form (5NF) states:



A relation R is in 5NF (also called P/NF) iff every JD in R is a consequence of the candidate keys of R.

In layman's terms if a relation R is in 4NF and it is n-decomposable into P1, P2 .. Pn, such that
R = P1 JOIN P2 ... JOIN Pn where n > 2,
such relation is not in 5NF. It may therefore be decomposed to achieve 5NF relations.

Put another way, a relation R is in 5NF iff it is in 4NF and it is not decomposable, except the decompositions are based on a candidate key of R, and the minimum number of projections is 3.

Now examine relation **SupplierSchedule. SupplierSchedule** is not in 5NF because it has a JD (i.e. the JD constraint) that is not a consequence of its candidate key. In other words, **SupplierSchedule** can be decomposed, but this is not implied by its candidate key [Supl#, Item#, Proj#].

Note: For most practical purposes, you only have to worry about 5NF if you are trying to implement an M:M relationship involving more than two relations. Once in 5NF, further decompositions would share candidate keys and are therefore to no avail (recall corollary of Heath's theorem). Notwithstanding this, other normal forms have been proposed, as will be discussed in the upcoming section.

4.11 Other Normal Forms

The field of Database Systems is potentially a contemptuous one. Indeed, there are accounts of former friends or colleagues becoming foes over database quibbles (in Figure 4-7 of section 4.12, the current author relates a personal experience he had as a young software engineer on a project of national importance). Various individuals have proposed several database theorems and methodologies, but they have not all gained universal acceptance as have the normal forms of the previous sections. Two additional normal forms that have been, and will no doubt continue to be the subject of debate are the *domain-key normal form* (DKNF) and the *sixth normal form* (6NF). Without picking sides of the debate on these two normal forms, this section will summarize each.

4.11.1 The Domain-Key Normal Form

The domain-key normal form (DKNF) was proposed by Ronald Fagin in 1981. Unlike the other normal forms which all relate to FDs, MVDs and JDs, this normal form is defined in terms of domains and keys (hence its name). In his paper, Fagin showed that a relation DKNF has no modification anomalies, and that a relation without modification anomalies must be in DKNF. He therefore argued that a relation in DKNF needed no further normalization (at least, not for the purpose of reducing modification anomalies). The definition of DKNF is as follows:

> A relation is in DKNF if every constraint on the relation is a logical consequence of the definition of its keys and domains.

This definition contains three important terms that need clarification:

- A constraint is used to mean any rule relating to static values of attributes. Constraints therefore include integrity rules, editing rules, foreign keys, intra-relation references, FDs and MVDs, but exclude time-dependent constraints, cardinality constraints and constraints relating to changes in data values.
- A key is a unique identifier of a row (as defined in Chapter 3).
- A domain is a pool of legal attribute values (as defined in Chapter 3).

The implication of the DKNF is clear: If we have a relation that contains constraint(s) that is (are) not a logical consequence of its (candidate) key and domains, then that

relation is not in DKNF, and should therefore be further normalized. The DKNF as proposed by Fagin, therefore represents an ideal situation to strive for.

Unfortunately, a number of problems arise from consideration of DKNF:

- Any constraint that restricts the cardinality of a relation (i.e. the number of tuples in the relation) will render it in violation of DKNF. (It was perhaps for this reason that Fagin excluded from his definition of constraints, time-dependent constraints or constraints relating to data values.) However, there are many relations for which such constraints are required.
- There is no known algorithm for converting a relation to DKNF. The conversion is intuitive and for this reason described as artistic rather than scientific.
- Not all relations can be reduced to DKNF (relations with cardinality constraints fall in this category).
- It is not precisely clear as to when a relation can be reduced to DKNF.

For these reasons, the DKNF has been compared by Date (see [Date, 2006]) to a "straw man… of some considerable theoretical interest but not yet of much practical ditto."

4.11.2 The Sixth Normal Form

A sixth normal form (6NF) has been proposed by C. J. Date in [Date, 2003], after several years of exploitation, expounding, and research in the field of database systems. It relates to so-called *temporal databases*. Date wrote a whole book on the subject; a summary of the essence is presented in this sub-section. Date defines a temporal database as a database that contains historical data as well as, or instead of current data. Temporal databases are often read-only databases, or update-once databases, but they could be used otherwise. In this sense, a temporal database may be considered as a precursor to a data warehouse (discussed in Chapter 24).

For the purpose of illustration, assume that we are in a college or university setting and desire to store the relation **Course** as defined below:

Course {CourseNo, CourseName, CourseCred}

Suppose further that we desire to show different courses at the time they existed in the database. To make our analysis more realistic, let us also make the following additional assumptions:

- The primary key is **CourseNo**; for any given course, the attribute **CourseNo** cannot be changed.
- For any given course, the attribute **CourseName** may be changed any point in time.
- For any given course, the attribute **CourseCred** may be changed any point in time.

We may be tempted to introduce a timestamp on each tuple, and therefore modify the definition of **Course** as follows:

Course {CourseNo, CourseName, CourseCred, EffectiveDate}

Figure 4-6 provides some sample data for the **Course** relation. By introducing the attribute **EffectiveDate**, we have actually introduced a new set of concerns as summarized below:

1. If we assume that the FD CourseNo → {CourseName, CourseCred, EffectiveDate} is (still) in vogue, then **Course** is in 5NF. However, in this case, if the **CourseName** or **CourseCred** of a given **Course** tuple changes at a given effective date, there is no way of showing what it was before, unless we create a new course and assign a new **CourseNo**. In either case, this clearly, is undesirable.

2. Suppose we assume the FDs CourseNo → CourseName and [CourseNo, EffectiveDate] → CourseCred.Then, the relation is not in 2NF, and therefore needs to be decomposed into two decompositions:

   CourseDef {CourseNo, CourseName} and

   CourseTimeStamp {CourseNo, EffectiveDate, CourseCred}

   Both of these relations would now be in 5NF. However, if we now desire to change the **CourseName** of a course for a given effective date, we cannot represent this in the current schema.

3. We could introduce a surrogate (say **CourseRef**) into relation **Course**, and key on the surrogate, while ignoring the FDs stated in (1) and (2) above. In this case, **Course** would be in violation of 3NF, and if we attempt to decompose, we would revert to the situation in case (2) above.

CourseNo	CourseName	CourseCred	EffectiveDate
CS120	Introduction to Computer Science	3	1990
CS120	Introduction to Computer Science	4	2005
CS140	Computer Programming I	3	1990
CS140	Computer Programming I	4	2005
CS145	Computer Programming II	3	1990
CS145	Computer Programming II	4	2005
CS130	Pascal Programming	3	1990
...			

***Figure 4-6.** Sample Data for the Course Relation*

The reason for these problems can be explained as follows: The relation **Course** as described, defines the following predicate:

- Each course is to be accounted for (we say **Course** is under contract).
- Each course has a **CourseName** which is under contract.
- Each course has a **CourseCred** which is under contract.

The predicate involves three distinct propositions. We are attempting to use the timestamp attribute (**EffectiveDate**) to represent more than one proposition about the attribute values. This, according to Date, is undesirable and violates the sixth normal form.

We now state Date's theorem for the sixth normal form (6NF):

A relation R is in 6NF iff it satisfies no non-trivial JDs at all. (A JD is trivial iff at least one of its projections is over all of the attributes of the relation.)

Put another way, a relation R is in 6NF iff the only JDs that it satisfies are trivial ones.

Note: 6NF as defined, essentially refines 5NF. It is therefore obvious from the definition that a relation in 6NF is necessarily in 5NF also.

Let us now revisit the **Course** relation: With the introduction of the timestamp attribute (**EffectiveDate**), and given the requirements of the relation, there is a non-trivial JD that leads to the following projections:

CourseInTime {CourseNo, EffectiveDate} PK [CourseNo, EffectiveDate]

CourseNameInTime {CourseNo, CourseName, EffectiveDate} PK [CourseNo, EffectiveDate]

CourseCredInTime {CourseNo, CourseCred, EffectiveDate} PK [CourseNo, EffectiveDate]

Observe that the projection **CourseInTime** is strictly speaking, redundant, since it can be obtained by a projection from either **CourseNameInTime** or **CourseCredInTime**. However, in the interest of clarity and completeness, it has been included.

This work by C. J. Date represents a significant contribution to the field of database systems, and will no doubt be a topical point of discussion in the future.

4.12 Summary and Concluding Remarks

This concludes one of the most important topics in your database systems course. Take some time to go over the concepts. Figure 4-7 should help you to remember the salient points.

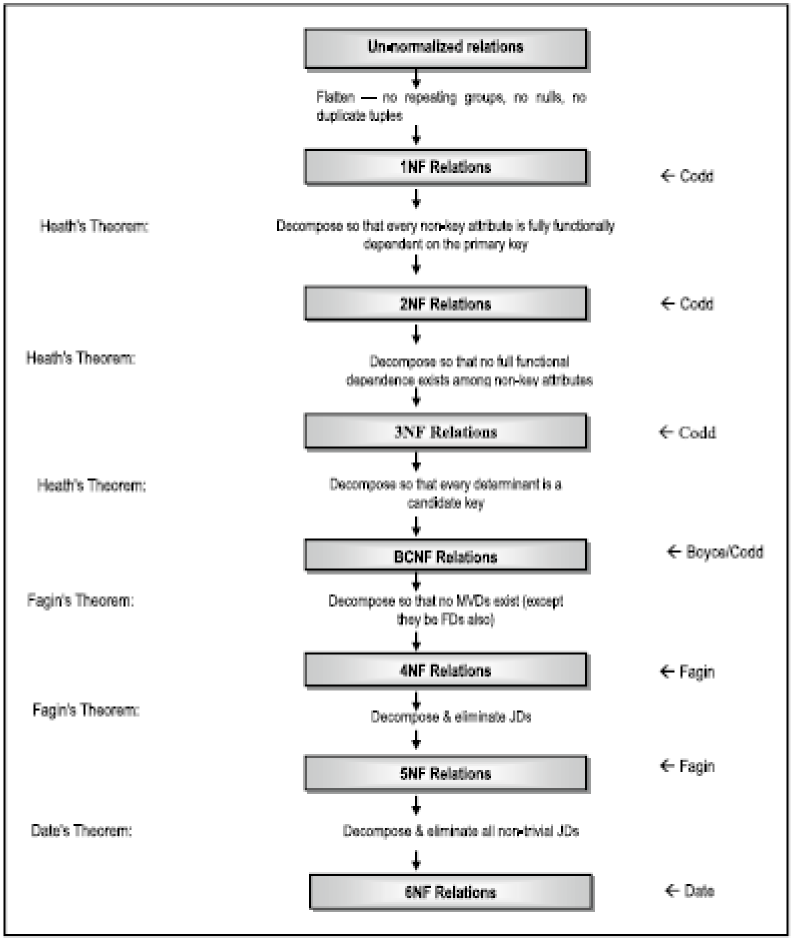
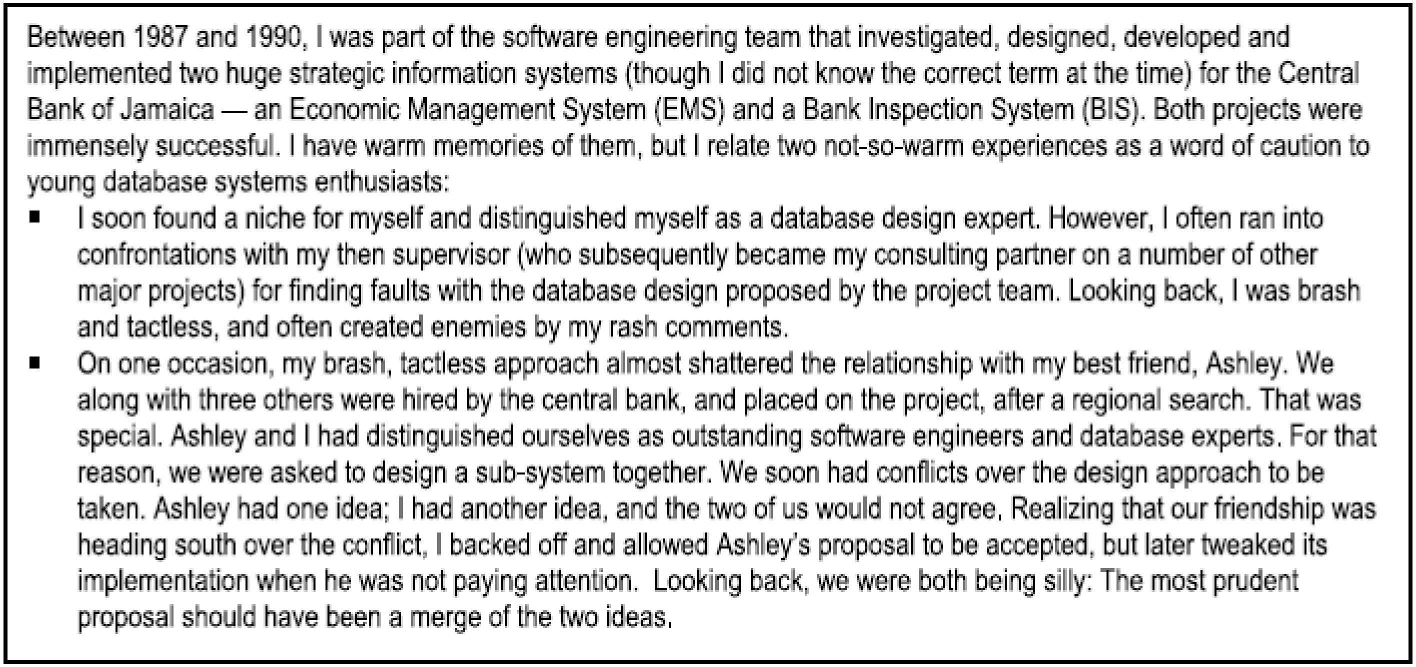

Un-normalized relations
Flatten — no repeating groups, no nulls, no duplicate tuples
1NF Relations
← Codd
Heath's Theorem:
Decompose so that every non-key attribute is fully functionally dependent on the primary key
2NF Relations
← Codd
Heath's Theorem:
Decompose so that no full functional dependence exists among non-key attributes
3NF Relations
← Codd
Heath's Theorem:
Decompose so that every determinant is a candidate key
BCNF Relations
← Boyce/Codd
Fagin's Theorem:
Decompose so that no MVDs exist (except they be FDs also)
4NF Relations
← Fagin
Fagin's Theorem:
Decompose & eliminate JDs
5NF Relations
← Fagin
Date's Theorem:
Decompose & eliminate all non-trivial JDs
6NF Relations
← Date

Figure 4-7. *Summary of Normalization*

Traditionally, it has been widely accepted that for most databases, attainment of 3NF is acceptable. This course recommends a minimum attainment of BCNF. Recall that as stated earlier (in section 4.8), BCNF is really a refinement of 3NF, and the normalization process can bypass 2NF and 3NF, and go straight to BCNF. In rare circumstances, it may be required to proceed to 5NF or 6NF, which is the ultimate.

Normalization is a technique that must be mastered by database designers. It improves with practice and experience, and ultimately becomes almost intuitive. As your mastery of normalization improves, you will find that there is a corresponding improvement in your ability to design and/or propose database systems for various software systems. However, be aware that the converse is also true: failure to master fundamental principles of database design will significantly impair one's ability to design quality software systems. Notwithstanding this, be careful not to be antagonistic about your views, as informed as they may be. In this regard, Figure 4-8 summarizes a practical experience of the author.



Between 1987 and 1990, I was part of the software engineering team that investigated, designed, developed and implemented two huge strategic information systems (though I did not know the correct term at the time) for the Central Bank of Jamaica — an Economic Management System (EMS) and a Bank Inspection System (BIS). Both projects were immensely successful. I have warm memories of them, but I relate two not-so-warm experiences as a word of caution to young database systems enthusiasts:

- I soon found a niche for myself and distinguished myself as a database design expert. However, I often ran into confrontations with my then supervisor (who subsequently became my consulting partner on a number of other major projects) for finding faults with the database design proposed by the project team. Looking back, I was brash and tactless, and often created enemies by my rash comments.
- On one occasion, my brash, tactless approach almost shattered the relationship with my best friend, Ashley. We along with three others were hired by the central bank, and placed on the project, after a regional search. That was special. Ashley and I had distinguished ourselves as outstanding software engineers and database experts. For that reason, we were asked to design a sub-system together. We soon had conflicts over the design approach to be taken. Ashley had one idea; I had another idea, and the two of us would not agree, Realizing that our friendship was heading south over the conflict, I backed off and allowed Ashley's proposal to be accepted, but later tweaked its implementation when he was not paying attention. Looking back, we were both being silly: The most prudent proposal should have been a merge of the two ideas.

Figure 4-8. *Database Quibbles almost got me in Trouble*