NORMALIZATION FOR NORMAL PEOPLE UNDERSTANDING ALGORITHMS FOR GETTING TO 3NF

Larry Newcomer
The Pennsylvania State University
York Campus

ABSTRACT

Database projects typically weave database-specific activities (such as Enterprise Data Modeling, Conceptual Data Modeling, Logical Database Design, and Physical Database Design) into standard development methodologies. This enhances methodologies such as the traditional SDLC (Waterfall Model), Rapid Application Development (RAD), and the newer Object-Oriented development methodologies to accommodate data-centric projects. In all cases, an abstract model of the relevant data requirements must be developed and documented during Conceptual Data Modeling, the output of which can take several forms, such as ER diagrams (ERD), Enhanced ER diagrams, or UML Class diagrams. The Logical Database Design phase then transforms the Conceptual Data Model into a set of relational table structures which can be implemented using a commercial DBMS. Prior to physical design and implementation, however, the table structures are typically analyzed and improved through a process called normalization. This step is appropriate regardless of whether traditional (ERD, EERD) or OO (UML Class diagram) modeling has been used, thus making good normalization skills highly desirable for all database project teams. However, such skills are not always easy to come by. Many introductory texts present normalization solely through examples, an approach that does not scale well to real projects. Advanced texts present formal mathematics that is not readily assimilated by undergraduates and practicing professionals. This paper presents algorithms for converting improper relations to 3NF in a formal manner that is nevertheless accessible to undergraduates and database professionals alike. The algorithms depend only on the concepts of primary key, candidate key, and functional dependency which are defined herein. The paper begins with basic concepts and definitions, gives two equivalent algorithms for putting an improper relation into 1NF, and ends with a discussion of algorithms for 2NF and 3NF. The relationship between 3NF and BCNF is also briefly discussed. These results should prove useful in both the classroom and corporate environments. The paper assumes that the reader is already familiar with database development methodologies, data modeling, normal forms, and basic normalization concepts.

INTRODUCTION

Current database development methodologies all produce a Conceptual Data Model of relevant data requirements and business rules that is independent of any particular technology, hardware/software platform, etc [1]. Although conceptual modeling tools may differ, the activity of modeling remains constant across methodologies. For example, the traditional SDLC (Waterfall model), Information Engineering (IE), and Rapid Application Development (RAD) methodologies may employ Entity-Relationship Diagrams (ERD's) or Enhanced Entity-Relationship Diagrams (EERD's) for Conceptual Modeling [2], while Object-Oriented development methodologies may employ UML Class Diagrams for the same purpose [5].

After a Conceptual Data Model has been developed, it must ultimately be transformed into a format that is suitable for implementation with a particular database technology. At the

current time, this involves converting the Conceptual Data Model into relational table structures [3] or Object-Oriented class definitions [7] in a process usually called Logical Database Design [4].

Outputs from Logical Design in turn become the inputs to the process of Physical Design in which the Logical Design specifications are translated into the format required for implementation on a specific DBMS [8]. Prior to releasing the logical design to the physical design process, however, the table or class structures are typically put through a process called *normalization*. Normalization detects and corrects several problems whose root cause is data *redundancy*, i.e., storing the same information in more than one place in the database [6].

Problems related to data redundancy include the following:

- **Wasted storage space**: Duplicate copies of the same data values wastes space.

- **Update anomalies**: An update anomaly occurs when changing a given data value in the database requires multiple updates because that value occurs in multiple copies. For example, if Fred's telephone number occurs at five different places within the database and needs to be changed, all five copies must be updated. This wastes computer time (both processor and I/O), and can also result in data inconsistency if for some reason all five copies of the phone number are not updated identically.

- **Insertion anomalies**: An insertion anomaly occurs when it is not possible to store (i.e., add) a given piece of information without also storing some other, unrelated piece of information as well. For example, it may be desired to insert Fred's student id into the database, but the database design requires that in order to do so a course number for a course that Fred is taking must also be inserted. If Fred hasn't yet enrolled in any courses, this creates an obvious problem. Although there are workarounds for this situation (e.g., entering a "dummy" course number or a null value), the workarounds create serious problems of their own [4]. Like update anomalies, insertion anomalies are undesirable in a database design.

- **Deletion anomalies**: A deletion anomaly occurs when it is not possible to delete a given piece of information without also deleting another, unrelated piece of information that it is desirable not to delete. For example, Fred may withdraw from a course and so it is desired to delete that course from Fred's student record. If the course to be deleted is the only course Fred is taking, poor database design may force other information about Fred (e.g., Fred's address and telephone number) to be deleted along with the course information. Again there are possible workarounds for this situation (e.g., replacing the course information with "dummy" or null placeholders), but these also create serious problems of their own [4]. Deletion anomalies, like insertion and update anomalies, are undesirable in a good database design.

The normalization process first identifies undesirable redundancies (and corresponding anomalies) and then eliminates them through a technique called *decomposition*. Decomposition splits one table (or class) into two (or eventually more) tables (or classes). In order to ensure that splitting tables during the normalization process does not result in loss of information or invalidation of important business rules and constraints, relational theory requires that decompositions be *lossless-join*, *dependency-preserving* decompositions [8].

- **Lossless-Join Decomposition**: The decomposition must not result in loss of information in the database (decompositions that result in loss of information are called *loss-y*). All data in the original table must be recoverable via relational join operations on the decomposed tables. This means that any row in the original table must be recoverable by combining appropriate rows of the decomposed tables [1].

- **Dependency-Preserving Decomposition**: The decomposition must not prevent constraints that were enforced on the original relation from being enforced on the decomposed relations. This means that any constraint on the original table must be enforceable by setting appropriate constraints on the decomposed tables. Thus there is no need to perform joins on the decomposed tables to discover whether a constraint on the original table is violated [1].

In order to identify undesirable redundancies and eliminate them through lossless-join, dependency-preserving decompositions, we must be able to identify *functional dependencies* in the original and decomposed relations. Understanding functional dependencies is critical to creating good database designs [9]. In order to better explore functional dependencies we first introduce the concepts of primary and candidate keys along with a notation for capturing *relation structure*.

A *candidate key* is a minimal set of one or more columns in a table whose data values (taken together as a whole if there are multiple columns) are guaranteed never to be duplicated in the table. This means that the combined data values for all the candidate key columns in a given row must differ by at least one binary digit from all the other combined data values for the candidate key columns in every other row of the table. For example, if (FName, MI, LName, Phone) are four columns that form a candidate key for a table, then it must be *impossible* for any two rows of the table to have exactly the same data values for each of FName, MI, LName, and Phone. The concept of *minimal* means that if any column is removed from the candidate set of columns, the remaining columns no longer form a candidate key (i.e., are no longer guaranteed to be unique). A candidate key consisting of more than one column is called a *composite key*, while a single-column key is called a *simple key*.

Note that while (FName, MI, LName, Phone) may be a good example for introducing the concept of candidate key, in real life it is difficult to guarantee uniqueness. Since people share names and phone numbers, duplication of key values is actually possible. Suppose, for example, that Fred J. Pherd Jr. lives with his father Fred J. Pherd Sr. at the same address with the single phone number 555-1234. If the LName data value is entered as ‘Pherd’ for both father and son, then we would have duplicate rows in the database – and of course (FName, MI, LName, Phone) would not be a candidate key. This is why database designers will often create special fields (such as account numbers, part numbers, etc.) that have no inherent meaning in the real world but whose sole purpose is to serve as candidate keys that identify rows in a table.

The database designer will also pick one key from among the candidate keys for a table to serve as that table’s *primary key*. The primary key is the one most used to locate rows in a table. Ideally it should express what the table is “about”, should be stable (i.e., once entered for a row, its value should not change), and should consist of a minimal number of columns (ideally one).

Columns that are not themselves a candidate key and are not part of a composite candidate key are known as *non-key columns*. Note well that non-key columns are not part of any

candidate key (which includes but is not limited to the primary key).

We now introduce a simple notation for capturing the name, attributes, and primary key for a table:

table-name (primary-key-col$_1$, primary-key-col$_2$, column$_3$, column$_4$, …, column$_n$)

In this straightforward notation the table name is written first followed by a parenthesized list of attributes. The primary key attribute (or attributes if it is a composite key) are all underlined. This notation for relation structure provides an essential tool for exploring functional dependencies and carrying out normalization via lossless-join, dependency-preserving decompositions. The following table structure will be used in later examples. It is an *unnormalized* or *improper* relation badly in need of improvement.

Animals (AnimalID, Breed, AnimalType, FoodType, VetID, VetPhone, OwnerID, OwnerName)

FUNCTIONAL DEPENDENCIES

Let A_1, A_2, …, A_n and B be attributes of a relation (columns of a table). We say that the set of attributes (A_1, A_2, …, A_n) *functionally determines* (or simply *determines*) B, or that B is *functionally dependent* on (A_1, A_2, …, A_n), and we write

(A_1, A_2, …, A_n) → B

if for any two rows in which the corresponding data values for (A_1, A_2, …, A_n) all match (i.e., are equal), then the data values for B in the same two rows also match (i.e., are equal) [6]. The Left Hand Side (LHS) of a functional dependency, for example (A_1, A_2, …, A_n) in the definition above, is called a *determinant*.

Another way to define functional dependency is that (A_1, A_2, …, A_n) → B if and only if for each set of data values that occurs for (A_1, A_2, …, A_n) in the table there is at most one corresponding data value for B [4]. Consider the following portion of an Enrollments table:

Table 1 Enrollments

StudentID	StuName	StuEmail	StuMajor	StuDorm
111	Fred	F111	Math	East
222	Sue	S222	InfoSci	North
333	Mary	M333	English	North
444	Fred	F444	Math	East

Assume that StudentID is the primary key and that StuEmail is a candidate key (this implies that both fields are guaranteed to always be unique). Does StudentID functionally determine any other columns? The answer is in fact that StudentID determines *all* the other columns, i.e.,

StudentID→StuName
StudentID→StuEmail
StudentID→StuMajor
StudentID → StuDorm

We can abbreviate this set of dependencies as

StudentID → StuName, StuEmail, StuMajor, StuDorm

Why do these dependencies hold? When a column (or group of columns) is unique, it automatically determines every other field in the table. This is because the uniqueness of the Left Hand Side (the determinant or LHS) means that a given data value (or set of data values if the determinant consists of multiple columns) can only occur in one row of the table. Since a given data value (or values) can only occur in one row, it can only match up with at most one value for the RHS (Right Hand Side of the dependency). This satisfies the second definition of functional dependency given above. Thus candidate keys (including primary keys) automatically determine all other columns in a table.

Since StuEmail is also unique (a candidate key), we have the following additional dependencies:

StuEmail → StudentID, StuName,StuMajor, StuDorm

Note that the Right Hand Side (RHS) of a dependency does *not* have to be unique in order to have a valid dependency, as in StudentID → StuDorm or StudentID → StuMajor. Both the StuDorm and StuMajor columns have duplicate entries, yet they correctly appear on the RHS of the indicated dependencies.

So far, the LHS (determinants) of our example dependencies have been unique. This is not required by the definition of functional dependency, however. Consider the following portion of the Animals table whose structure was shown earlier:

Table 2 Animals

AnimalID	Breed	AnimalType	FoodType
111	Dachshund	Dog	Dry
222	Persian	Cat	Tuna
333	Brittany Spaniel	Dog	Dry
444	Dachshund	Dog	Dry
555	Siamese	Cat	Tuna
666	Canary	Bird	Seed

The only column that is unique is the primary key AnimalID. Nevertheless, there are three dependencies involving the other (non-unique) columns. Specifically,

Breed → AnimalType, FoodType
AnimalType → FoodType

Notice that Breed is not unique. The data value "Dachshund" appears in the Breed column in both rows 1 and 4. What matters with respect to the dependency Breed → AnimalType is that the same two rows of the AnimalType column must also contain duplicate data values (in the example above the value "Dog" appears in both rows 1 and 4 of the AnimalType column). To have a functional dependency, whenever two or more rows of the Breed column contain data values that equal one another, then the data values in the corresponding rows of the AnimalType column must also all equal one another.

Now consider the dependency AnimalType → FoodType. Rows 1, 3, and 4 of the AnimalType column contain the value "Dog"; the corresponding rows of the FoodType column all contain the value "Dry". Likewise rows 2 and 5 of the AnimalType column contain the value "Cat" while the corresponding rows of the FoodType column all contain the value "Tuna". The value "Dog" in AnimalType always matches up with "Dry" in FoodType, while the value "Cat" in AnimalType always matches up with "Tuna" in FoodType. If every possible value of AnimalType can be proven to match up with no more than one value in FoodType, then by definition AnimalType → FoodType. Although the data in Table 2 Animals would seem to support the claim that FoodType → AnimalType, this is not the case. Suppose for example that we add a row with AnimalType = 'Rodent' and FoodType = 'Seed'. There would then be two rows with FoodType = 'Seed' but one would have AnimalType = 'Rodent' and the other would have AnimalType = 'Bird", thus violating the definition of a functional dependency.

It is critical to note that a functional dependency A → B cannot be *proven* to hold by observing existing data values in columns A and B of the relevant table. This does not work because even though a dependency may seem to exist in the data today, tomorrow someone may add new data values to the columns that would invalidate the dependency. To demonstrate a functional dependency, one must be able to give arguments proving that whenever two or more rows in the columns making up the LHS A have the same value, then the corresponding rows in column B will also be equal to one another – every single time, guaranteed. Thus in Table 2 Animals if a given type of animal is always fed the same type of food, then AnimalType → FoodType. Note again that if it is possible for different types of animals to be fed the same

type of food, then FoodType does *not* determine AnimalType.

We *can* use existing data to show that a functional dependency does *not* hold by finding a *counter-example* in the data. For example, the claim that AnimalType → Breed can be disproved by observing that in rows 1 and 3 of the existing table AnimalType has the value "Dog", but that in rows 1 and 3 Breed has not one, but *two* different values ('Dachshund' in row 1 and 'Brittany Spaniel' in row 3), thus failing to meet the definition of functional dependency.

The following table lists all the functional dependencies for Tables 1 and 2 above. We assume that students in the same StuMajor are always assigned to the same StuDorm (such a constraint derived from the way things work in the "real world" is called a *business rule*). Similarly, we assume the business rule that animals of the same type are always fed the same type of food.

Table 3 List of Functional Dependencies

Functional Dependencies for Table 1 Students	Functional Dependencies for Table 2 Animals
StudentID → StuName, StuEmail, StuMajor, StuDorm	AnimalID → Breed, AnimalType, FoodType
	Breed → AnimalType, FoodType
StuEmail → StudentID, StuName, StuMajor, StuDorm	AnimalType → FoodType
StuMajor → StuDorm	

NORMAL FORM DEFINITIONS

The normalization process is guided by a set of definitions for *normal forms*. Each normal form identifies specific design flaws and provides an algorithm (or detailed set of instructions) for correcting the design. First Normal Form (1NF) deals with issues of compound fields and repeating groups, while Second Normal Form (2NF) deals with issues involving composite candidate keys. Third Normal Form (3NF) deals with certain kinds of relationships between non-key columns, and Boyce-Codd Normal Form (BCNF) requires that all functional dependencies have determinants that are unique (i.e., have no duplicate values in the LHS column or columns). Each "higher" normal form is a refinement of its preceding normal form, so to be in 2NF a table must also be in 1NF; to be in 3NF a table must also be in 2NF (and so in 1NF); to be in BCNF a table must also be in 3NF (and so in 2NF and 1NF). In short, each higher normal form is a proper subset of its predecessor(s). We begin by presenting the definitions for each normal form.

FIRST NORMAL FORM (1NF)

A relation is in First Normal Form if all its domains are atomic. A *domain* is the set of allowable data values that may appear in a column. A domain is *atomic* if each data value in the domain can *not* be broken down into parts such that any of the parts will ever be meaningful or useful. Put another way, a domain is atomic if nobody will ever need to work with a proper subset of the characters making up any of the data values in the domain. Since this concept depends on the potential human need to work with just a part of a data value, it is obviously determined by business requirements rather than by pure theory.

In practice there are two types of design flaws that result in non-atomic domains. A *compound* (or *composite*) attribute (or field or column) consists of more than one kind of usable information stored in a single column. The classic examples of composite fields are illustrated in the following table (for simplicity shown with only one row):

Table 4 Composite Fields

Name	Address
Fred J Pherd	123 Sesame St, BirdCity, FL 12345

It is easy to fix composite fields by breaking them up into separate columns *before any data is loaded into the table* (it may not be so easy to break up composite fields *after* data has been loaded!). The table below corrects the Name and Address composite fields. Note that the Street column could have been broken up into the three columns StreetNumber (123), StreetName (Sesame), and StreetType (St). It is only safe to combine these three fields if it can be successfully argued that no one will ever want to work with them separately (or conversely that they will always be treated as one combined whole).

Table 5 Corrected Composite Fields

FName	MI	LName	Street	City	State	Zip
Fred	J	Pherd	123 Sesame St	BirdCity	FL	12345

The second type of non-atomic domain involves *repeating groups*. A repeating group is more than one instance of the same kind of information appearing in a single column. Consider the CurrentCourses column in the table below:

Table 6 Repeating Group

StudentID	CurrentCourses
111	IST 300, IST 400, Math 200, English 100

There are four course names in the CurrentCourses column. The four data values are all the same *kind* of information (course names), but the issue is that there is more than one of them. Unfortunately, it is not as simple to correct repeating groups as it is to correct composite fields. The next section presents a procedure for converting *improper relations* (with repeating groups) into 1NF (where repeating groups are eliminated).

SECOND NORMAL FORM (2NF)

Normal forms above 1NF (up to BCNF) deal with issues involving functional dependencies. Remember that a relation must be in 1NF before it can be in 2NF.

Second Normal Form deals specifically with issues that only arise when a table has one or more composite candidate keys. If a table has no composite candidate keys and it is in 1NF, then it is automatically in 2NF. For a relation in 1NF to be in 2NF when it has composite candidate keys, it must have no functional dependencies in which a non-key column depends on a proper subset of a candidate key, i.e., non-key columns must depend on the *entire* candidate key, not just a part of it. Consider the following relation structure:

Orders (CustID, ItemID, ItemDescription, Quantity, CustZip)

The combination of CustID and ItemID form the only candidate key, and therefore are chosen as the (composite) primary key. No single column in Orders is guaranteed to be unique. Customers may order multiple items, and so the same CustID may appear in multiple rows. Items may be ordered by multiple customers, and so the same ItemID may appear in multiple rows. The description, quantity, and zip code fields may also contain duplicate data values.

Now consider the functional dependencies. Since (CustID, ItemID) together are unique, they determine every other column. Since a given customer will always have the same zip code, CustID → CustZip, and since a given item will always have the same description, ItemID → ItemDescription. A complete list of functional dependencies follows:

(CustID, ItemID) → ItemDescription, Quantity, CustZip
CustID → CustZip
ItemID → ItemDescription

Since CustID by itself is a proper subset of the composite candidate key (CustID, ItemID), the dependency CustID → CustZip violates the definition of 2NF. Likewise, since ItemID by itself is a proper subset of the composite candidate key (CustID, ItemID), the dependency ItemID → ItemDescription also violates 2NF.

These problems can be corrected with lossless-join, dependency-preserving decompositions as discussed below.

To enhance understanding of 2NF we lastly consider some anomalies associated with the sample relation (not in 2NF):

Insertion Anomaly: It is impossible to add a customer's zip code unless the customer has ordered an item

Deletion Anomaly: If we drop a customer who has placed the only order for an item, we lose the item description along with the customer information

Update Anomaly: If a customer orders five items, the customer's zip code will appear in five rows of the table. If the customer moves to a different zip code, all five copies will have to be updated

THIRD NORMAL FORM (3NF)

To be in 3NF relations must first be in 2NF. In addition, 3NF tables must not have any non-key columns that are dependent on other non-key column(s). An equivalent definition is that relations in 3NF must not have any *transitive dependencies* in which A → B and B → C, from which it follows that A → C [4]. Yet another way to define 3NF is to require that if the general dependency "LHS → R" holds, then one of the following must be true [6]:

- R is a column that appears in the group of columns making up the LHS [making this a *trivial* dependency since for any columns X, Y, and R if X → R, then (X, Y) → R]
- The LHS is a superkey (a group of one or more columns that uniquely identifies each row of the table; however, unlike a candidate key, a superkey may include "extra" columns that do not contribute to the uniqueness)
- R is part of a composite candidate key for the table in question

For an example of a table in 2NF but not in 3NF, consider the structure for Table 2 Animals:

Animals (<u>AnimalID</u>, Breed, AnimalType, FoodType)

AnimalID is the only candidate key (all other columns and combinations of columns have duplicate values in the sample data) and therefore is the primary key. Since all domains appear to be atomic and since the only candidate key is single-column, the table is in 1NF and also in 2NF. A partial list of functional dependencies from Table 3 List of Functional Dependencies is repeated below:

Breed →	AnimalType
Breed →	FoodType
AnimalType →	FoodType

Since Breed, Foodtype, and AnimalType are all non-key columns (i.e., not candidate keys or part of candidate keys), all three dependencies violate the first definition of 3NF. Note too that since the primary key AnimalID determines Breed, and since Breed → FoodType, we have a transitive dependency that also violates the second definition of 3NF. Finally, consider the dependency AnimalType → FoodType. The RHS (FoodType) is not part of the LHS (AnimalType); the LHS is not a superkey (since AnimalType is not unique); and the RHS is not part of any candidate key for the table. Thus none of the conditions in the more formal third definition of 3NF hold, and the relation also violates the third definition of 3NF. Examples of anomalies resulting from the structural problems associated with not being in 3NF are shown below. These problems can be corrected by lossless-join, dependency-preserving decompositions as discussed later.

Insertion Anomaly: A new food cannot be entered into the table unless there is an animal that is fed the new FoodType.

Deletion Anomaly: If you delete the only animal that eats a given FoodType,

you must delete the information about the food along with the information about the animal.

Update Anomaly: If a given AnimalType's food changes, the change must be made to multiple rows of the table, wasting computer resources and creating the possibility of inconsistent data should all copies of FoodType fail to be updated.

BOYCE-CODD NORMAL FORM (BCNF)

Boyce-Codd Normal Form addresses certain rare problems with data redundancy that occur in relations that are already in 3NF. Fortunately, these problems only arise when both of the following conditions hold (Watson, 1999):

- The relation has multiple, composite candidate keys and
-
- At least two of the composite candidate keys overlap, i.e., have at least one column in common

Since it is rare for the above conditions to arise in real-life designs, relations that are in 3NF are usually also in BCNF. In fact it is challenging to invent convincing examples of relations in 3NF that are *not* also in BCNF (the reader is encouraged to pause here and attempt the challenge).

The definition of BCNF is more straightforward than any of the normal forms considered so far: A relation is in BCNF if all its determinants are unique. Although this definition is simple, there is unfortunately no guarantee that dependency-preserving decompositions can be found to convert a non-BCNF relation to BCNF [8]. Hence the designer may have to choose between staying in 3NF (with the possibility, albeit rare, of certain anomalies) or going to BCNF to eliminate the anomalies at the price of possibly giving up dependency preservation. Since most real-world designs in 3NF do not exhibit the two conditions enumerated above (and are therefore in BCNF as well as 3NF), this paper focuses on getting relations to 3NF.

ALGORITHMS FOR GETTING TO 1NF

In order to put an improper relation into 1NF one must eliminate all compound (composite) attributes and all repeating groups. We consider these as two separate cases. Note that in all algorithms COPY means to place a copy of a column in a new table while leaving the original column intact, while MOVE means to remove a column from its original table and place it in a new table.

Case 1: Eliminate compound (composite) attributes

Definition: Multiple data values of different kinds in one column

PROCEDURE:

Split attributes (columns) until all attributes are atomic (will never need to be divided into portions forming meaningful or useful information). Remember that whether or not a data value can be divided into something meaningful or useful is a semantic issue of business rules, policies, etc., so close cooperation with users is essential to make good decisions regarding this issue.

EXAMPLE:

See Table 4 Composite Fields and Table 5 Corrected Composite Fields above for an example of how to eliminate compound attributes from a table.

Case 2: Eliminate repeating groups (There are two equivalent solutions given)

Definition: Multiple data values of the same kind in one column

1NF PROCEDURE-1:

List all columns with repeating groups (step 1)

Form the listed columns into separate sets of columns that describe the same entity (each column in a given set should "be about" the same entity) (step 2)

For each set of columns formed above:

If there exists a column or group of columns in the set of columns that uniquely identifies each row in the set of columns

Choose a minimal group of unique columns to act as PK for the set

Else

Create a new column in the original table that alone or in combination with other columns in the set can function as the PK for the set of columns being processed

End If (step 3)

Create a new table for the set of columns and assign an appropriate name to it (step 4)

COPY the original table's primary key (PK) to the new table (step 5)

MOVE the set of columns chosen in step 3 as the PK for the set from the original table to the new table, distributing any repeating groups into separate rows (step 6)

MOVE the rest of the columns in the set to the new table, distributing any repeating groups into separate rows (step 7)

Remove duplicate rows and rename modified tables as needed to reflect the new table structures (step 8)

Declare the PK of the new table to be the group of one or more columns chosen in step 3 and moved in step 6 plus the original table's PK copied in step 5 (step 9)

End For

Example of 1NF PROCEDURE-1:

Consider the following relation structure

Animals (AnimalID, Breed, Type, Food, VetID(s), VetPhone(s), OwnerID(s), OwnerName(s))

and the corresponding sample data

(step 1)

VetID, VetPhone, OwnerID, and OwnerName are the only attributes with repeating groups

(step 2)

We can organize the repeating group columns into two sets of attributes – one about Vets (VetID and VetPhone) and the other about owners (OwnerID and OwnerName)

(step 3 – 9 first iteration)

Choose VetID as the PK for the set {VetID, VetPhone} (step 3). Create a new table named PetVets (step 4), copy the original table's PK (AnimalID) to it (step 5), move VetID (chosen in step 3) to it (step 6), then remove VetPhone from the original table and place it in the new table (step 7). Remember to place each atomic value from a repeating group into a separate row and to make sure that the correct AnimalID values "stay with" each value for VetID and VetPhone, as in:

Table 7 Repeating Groups

AnimalID	Breed	Type	Food	VetID	VetPhone	OwnerID	OwnerName
111	Dachshund	Dog	Dry	AAA, BBB	555-1111, 555-2222	O11, O22	Fred, Sue
222	Siamese	Cat	Tuna	AAA, CCC	555-1111, 555-3333	O11	Fred

AnimalID	VetID	VetPhone
111	AAA	555-1111
111	BBB	555-2222
222	AAA	555-1111
222	CCC	555-3333

In this example there are no duplicate rows to remove, and it seems appropriate to retain the name PetVets for the new table (step 8). Make VetID (from step 6) plus AnimalID (from step 5) the composite PK of the new PetVets table, which now has the structure PetVets (AnimalID, VetID, VetPhone). Note that the original table now has the structure Animals (AnimalID, Breed, Type, Food, OwnerID, OwnerName). The finished PetVets table is shown below:

Table 7 PetVets Table from 1NF Procedure-1

AnimalID	VetID	VetPhone
111	AAA	555-1111
111	BBB	555-2222
222	AAA	555-1111
222	CCC	555-3333

(step 3 – 9 second iteration)

Choose OwnerID as the PK for the set {OwnerID, OwnerName} (step 3). Create a new table named PetOwners (step 4), copy the original table's PK AnimalID to it (step 5), move OwnerID to it (step 6), then remove OwnerName from the original table and place it in the new table (step 7). Remember to place each atomic value from a repeating group into a separate row, as in:

AnimalID	OwnerID	OwnerName
111	O11	Fred
111	O22	Sue
222	O11	Fred

If there are any duplicate rows in the new table, remove them and change table names as appropriate (step 8). In this example, no changes were made for step 8. Next, make OwnerID together with AnimalID the PK of the new PetOwners table (step 9). The new table now has the structure PetOwners (AnimalID, OwnerID, OwnerName) while the original table has the structure Animals (AnimalID, Breed, Type, Food). The finished PetOwners table is shown below:

Table 8 PetOwners Table from 1NF Procecure-1

AnimalID	OwnerID	OwnerName
111	O11	Fred
111	O22	Sue
222	O11	Fred

The complete set of relation structures and tables with sample data are shown below:

Animals (AnimalID, Breed, Type, Food)
PetVets (AnimalID, VetID, VetPhone)
PetOwners (AnimalID, OwnerID, OwnerName)

Table 9 Animals 1NF

AnimalID	Breed	Type	Food
111	Dachshund	Dog	Dry
222	Siamese	Cat	Tuna

Table 10 PetVets 1NF

AnimalID	VetID	VetPhone
111	AAA	555-1111
111	BBB	555-2222
222	AAA	555-1111
222	CCC	555-3333

Table 11 PetOwners 1NF

AnimalID	OwnerID	OwnerName
111	O11	Fred
111	O22	Sue
222	O11	Fred

Table 12 Repeating Groups Eliminated with Empty Cells

AnimalID	Breed	Type	Food	VetID	VetPhone	OwnerID	OwnerName
111	Dachshund	Dog	Dry	AAA	555-1111	O11	Fred
				BBB	555-2222	O22	Sue
222	Siamese	Cat	Tuna	AAA	555-1111	O11	Fred
				CCC	555-3333		

Table 13 Repeating Groups Eliminated & Empty Cells Filled In

AnimalID	Breed	Type	Food	VetID	VetPhone	OwnerID	OwnerName
111	Dachshund	Dog	Dry	AAA	555-1111	O11	Fred
111	Dachshund	Dog	Dry	BBB	555-2222	O22	Sue
222	Siamese	Cat	Tuna	AAA	555-1111	O11	Fred
222	Siamese	Cat	Tuna	CCC	555-3333	O11	Fred

1NF PROCEDURE-2:

List all columns with repeating groups (step 1)
Form the listed columns into separate sets of columns that describe the same entity (each column in a given set should "be about" the same entity) (step 2)

For each set of columns formed above:

Locate (or create by adding an additional column or columns) a column or group of columns that uniquely identifies each row in the set of columns (step 3)
Make the column or group of columns above part of the PK for the original table (step 4)
Remove repeating groups by spreading the data values in each repeating group into separate rows, copying data values from the original atomic columns into the otherwise "empty" cells of the new table rows formed in the process (step 5)
End For

Example of 1NF PROCEDURE-2:

We will use the same table as in the example of PROCEDURE-1. The first two steps are the same in both procedures so they are not repeated here.

(step 3)
For the first iteration choose VetID; for the second iteration choose OwnerID.
(steps 4-5)
The primary key of the original table is extended to include VetID and OwnerID as shown in the tables 13 and 14. In the first version of the table we remove repeating groups into separate rows, and in the final version of the table we fill in the "empty" data values with matching values from the original table:

Although the two procedures above produce quite different results (PROCEDURE-1 produced three tables and PROCEDURE-2 produces just one), after applying the conversion algorithms for 2NF and 3NF we shall see that the end result is the same in both cases! One of the strengths of the normalization process is that, regardless of the starting point, it zeros in on a "good" design for the database in question.

ALGORITHM FOR GETTING TO 2NF

The first step in putting a table in 2NF is to list all the functional dependencies and then identify those in the list that keep the table from being in 2NF, i.e., identify those dependencies where the LHS is just part of a composite candidate key. The decomposition in the 2NF procedure shown below will then eliminate the unwanted dependencies, putting the relation in 2NF.

Definition: Part of a composite key determines a non-key column

2NF PROCEDURE:

While there are tables not in 2NF:
For each original table O not in 2NF:
List all FD's for O that violate 2NF (LHS is just part of composite candidate key; RHS is non-key column) (step 1)
For each distinct LHS in the list:
MOVE all the RHS columns that correspond to the LHS being processed to a new table N (step 2)
COPY the LHS column(s) from the original table O to the new table N (step 3)
Make the copy of the LHS from O the primary key of the new table N (step 4)
Make the original LHS in O a foreign key in O referencing the copy of the LHS in N (step 5)
Assign an appropriate name to the new table N (step 6)
IF the original table O had data loaded Then
Remove any duplicate rows from O and N
End If (step 7)
End For
End For
End While

2NF Example-1:

We first consider the relation structures produced in the example for 1NF Procedure-1:

Animals (AnimalID, Breed, Type, Food)
PetVets (AnimalID, VetID, VetPhone)
PetOwners (AnimalID, OwnerID, OwnerName)

(step 1)

The Animals table is trivially in 2NF because the only candidate key is single-column (AnimalID). PetVets and PetOwners both have only one candidate key each, but since in both cases the candidate key is composite, we must check the functional dependencies for PetVets and PetOwners:

(AnimalID, VetID) → VetPhone
VetID → VetPhone (PROBLEM DEPENDENCY!)
(AnimalID, OwnerID) → OwnerName
OwnerID → OwnerName (PROBLEM DEPENDENCY!)

Since VetID and OwnerID are LHS's that are only part of their respective composite candidate keys, neither PetVets nor PetOwners is currently in 2NF.

(step 2) [Note: for this and following steps we combine iterations into one discussion]

Create a new table for each problem dependency and move the RHS columns from each problem dependency to its corresponding new table, giving:

Animals (AnimalID, Breed, Type, Food)
PetVets (AnimalID, VetID)
PetOwners (AnimalID, OwnerID)
NewTable (VetPhone)
NewTable (OwnerName)

(steps 3 - 6)

Copy the LHS column (or columns) of each problem dependency to its respective new table where it becomes the primary key (steps 3 - 4). Make the copy of the LHS still in the original table a foreign key referencing the new table, so that PetVets.VetID references Vets.VetID and PetOwners.OwnerID references Owners.OwnerID (step 5). Finally, assign the name Vets to the vet information table and the name Owners to the owner information table (step 6) to yield:

Animals (AnimalID, Breed, Type, Food)
PetVets (AnimalID, VetID)
PetOwners (AnimalID, OwnerID)
Vets (VetID, VetPhone)
Owners (OwnerID, OwnerName)

The results for the PetVets and Vets tables are shown below. Note the duplicate row 3 in Table 15 Vets.

Table 14 PetVets

AnimalID	VetID
111	AAA
111	BBB
222	AAA
222	CCC

Table 15 Vets

VetID	VetPhone
AAA	555-1111
BBB	555-2222
AAA	555-1111
CCC	555-3333

(step 7)

If data is already loaded in the tables when normalization is carried out, it can result in duplicate rows which are removed in this final step. Data values for the PetVets and Vets tables after duplicate rows are removed are shown below. A parallel situation would exist for PetOwners and Owners.

Table 16 Completed PetVets Table

AnimalID	VetID
111	AAA
111	BBB
222	AAA
222	CCC

Table 17 Completed Vets Table with Duplicate Rows Removed

VetID	VetPhone
AAA	555-1111
BBB	555-2222
CCC	555-3333

2NF Example-2

Consider the relation structure for Table 13 Repeating Groups Eliminated & Empty Cells Filled In:

Animals (AnimalID, Breed, Type, Food, VetID, VetPhone, OwnerID, OwnerName)

(step 1)

The only candidate key is the composite key (AnimalID, VetID, OwnerID). However, each of the three key columns is by itself a determinant. The list of "problem" dependencies created in step 1 thus consists of:

AnimalID → Breed, Type, Food
VetID → VetPhone
OwnerID → OwnerName

(step 2)

Create a new table for each distinct LHS and move the corresponding RHS's to their respective new tables yielding the structures:

Animals (AnimalID, VetID, OwnerID)
NewTable (Breed, Type, Food)
NewTable (VetPhone)
NewTable (OwnerName)

(steps 3 – 6)

Now copy the LHS's from Animals to their respective new tables, where they will become the primary key (steps 3 – 4), declare the original LHS's as foreign keys referencing their respective new tables (step 5), and assign appropriate names to the new tables (and/or rename existing tables if desired):

Animals (AnimalID, VetID, OwnerID)
AnimalCare (AnimalID, Breed, Type, Food)
Vets (VetID, VetPhone)
Owners (OwnerID, OwnerName)

(step 7)

Since Animals has no non-key columns, it is in 2NF. Since the three new tables have no composite candidate keys, they are also in 2NF. Note that it may again be necessary to remove duplicate rows from the new design if the decompositions are performed on tables that already contain data. See Table 17 Completed Vets Table above for an example of this situation. Obviously it is better to carry out normalization prior to loading data into the database – since this eliminates the need to remove any duplicate rows. The Owners table just prior to removing duplicate rows is shown below. The last two rows (in italics) are duplicates and would be removed in step 7:

OwnerID	OwnerName
O11	Fred
O22	Sue
O11	Fred
O11	Fred

Note that although all relations are now in 2NF, the relation structures produced in 2NF Example-2 are different from the relation structures produced in 2NF Example-1. Example-2 models the relationships among Animals, Vets, and Owners as a single 3-way relationship, while Example-1 models the same relationships as a pair of 2-way relationships. While both designs are in 2NF, the database designer should choose the design that best models the reality (business requirements) of the organization for which the database is being developed (this issue is beyond the scope of this paper). The following table 19-2 contrasts the two designs.

ALGORITHM FOR GETTING TO 3NF

Careful checking will reveal that the tables for both 2NF Example-1 and 2NF Example-2 are not only in 2NF, but, with one exception, are in 3NF and BCNF as well. The only table not in 3NF appears as Animals (AnimalID, Breed, Type, Food) in Example-1 and AnimalCare (AnimalID, Breed, Type, Food) in Example-2 (the table name "Animals" will be used in this section). This table is not in 3NF because of the dependencies

Table 19 2-Way versus 3-Way Relationships

Example 1 Two 2-Way Relationships	Example 2 One 3-Way Relationship
Animals (AnimalID, Breed, Type, Food) PetVets (AnimalID, VetID) PetOwners (AnimalID, OwnerID) Vets (VetID, VetPhone) Owners (OwnerID, OwnerName)	Animals (AnimalID, VetID, OwnerID) AnimalCare (AnimalID, Breed, Type, Food) Vets (VetID, VetPhone) Owners (OwnerID, OwnerName)

Breed → Type
Breed → Food
Type → Food

in all of which a non-key column determines another non-key column.

Definition: A non-key column determines another non-key column

3NF PROCEDURE:

A slight variation of the 2NF Procedure can be used for 3NF as well. Use the 2NF Procedure but replace the 2NF version of step 1 with the following:

> List all FD's for O that violate 3NF (LHS is a non-key column and RHS is another non-key column). Order the list so that if a column appears both as an LHS and an RHS, all the FD's with the column on the LHS appear in the list before any FD's with the column on the RHS. Note that if a column appears on both the LHS and RHS of the same dependency it can be safely removed from the LHS since (A, B) → B if and only if A → B. Finally if there are transitive dependencies of the form A → B, B → C, and A → C in the list, remove the dependencies of the form A → C (step 1)

3NF Example-1:

(step 1)
The list of FD's for Animals (AnimalID, Breed, Type, Food) that violate 3NF is:

Breed → Type
Breed → Food
Type → Food

Since Type appears on both an LHS and an RHS, order the list so that the dependencies with Type on the LHS appear before those where Type appears on the RHS:

Type → Food
Breed → Type
Breed → Food

Now note that Breed → Type, Type → Food, and Breed → Food form a set of transitive dependencies from which we should remove the dependency Breed → Food, leaving the final list:

Type → Food
Breed → Type

(steps 2 - 7 iteration 1)
Create a new table and move the RHS of the first dependency from Animals to the new table (step 2), then copy the LHS of the first dependency to the new table and declare it the primary key (steps 3 - 4). Make Type an FK in Animals referencing the new table (step 5) then give an appropriate name to the new table (step 6) and if data is loaded remove any duplicate rows (step 7). The table structures that result are:

Animals (AnimalID, Breed, Type)
Feeds (Type, Food)

(steps 2 - 6 iteration 2)
Create a new table and move the RHS of the second dependency (Breed → Type) to it (step 2), then copy the LHS of the second dependency to it and declare it the PK (steps 3-4). Make Breed in Animals an FK referencing Breed in the new table (step 5), then give an appropriate name to the new table (step 6) and remove any duplicate rows (step 7), yielding the table structures:

Animals (AnimalID, Breed)
Feeds (Type, Food)
Breeds (Breed, Type)

All three relations are now in 3NF as desired.

SUMMARY

The normalization process is often summarized with a witty version of a courtroom oath:

The key, the whole key, and nothing but the key (so help me Codd)

The phrase "the key" refers to the fact that every candidate key in a table automatically (since by definition candidate keys are unique) determines every other column in the table. Thus for any given candidate key, "the key" determines all the other columns.

The phrase "the whole key" ensures that the relation is in 2NF, i.e., that no proper subset of a candidate key determines a non-key column. Thus for any given candidate key, it is only "the whole key" that determines non-key columns.

The phrase "nothing but the key" ensures that the relation is in 3NF, i.e., that non-key columns are functionally dependent only upon (nothing but) candidate keys. Thus there are no non-key columns that are functionally dependent upon other non-key columns.

Finally, the phrase "so help me Codd" refers to E. F. Codd, whose seminal paper published in 1970 earned him the reputation as the father of the relational database model [4].

REFERENCES

1. Connolly, T., Begg, C., Strachan, A., Database Systems: A Practical Approach to Design, Implementation, and Management, 2nd ed., Harlow, England, 1999.

2. Hoffer, J., Prescott, M., McFadden, F., Modern Database Management, 6th ed., Upper Saddle River, NJ, Prentice Hall, 2002.

3. Kroenke, D., Database Processing: Fundamentals, Design, and Implementation, 7th ed., Upper Saddle River, NJ, Prentice Hall, 2000.

4. Mannino, M., Database Design, Application Development, and Administration, 2nd ed., Boston, MA, McGraw-Hill Irwin, 2004.

5. Post, G., Database Management Systems: Designing and Building Business Applications, 2nd ed., Boston, MA, McGraw-Hill Irwin, 2002.

6. Ramakrishnan, R., Gehrke, J., Database Management Systems, 3rd ed., Boston, MA, McGraw-Hill, 2003.

7. Riccardi, G., Principles of Database Systems with Internet and Java Applications, Boston, MA, Addison Wesley, 2001.

8. Silberschatz, A., Korth, H., Sudarshan, S., Database System Concepts, 4th ed., Boston, MA, McGraw-Hill, 2002/

9. Ullman, J., Widom, J., A First Course in Database Systems, Upper Saddle River, NJ, Prentice Hall, 1997/

10. Watson, R., Data Management: Databases and Organizations, 2nd ed., New York, NY, John Wiley & Sons, Inc., 1999.

BIOGRAPHICAL INFORMATION

Larry R. Newcomer is an Associate Professor of Information Sciences and Technology (IST) at The Pennsylvania State University York Campus. He teaches IST courses in the areas of database, networking, software development, and systems development and integration. He has written four textbooks including *SELECT…SQL*, *The Relational Database Language* published by Macmillian. He has published papers in the areas of networking, database technology, systems development, and curriculum development, and has served as an industry consultant in the areas of networking, database, and software development.