RESEARCH REPORT

OCCASIONAL PAPER

Improving Data Quality in Relational Databases: Overcoming Functional Entanglements

Tennyson X. Chen, Martin D. Meyer, Nanthini Ganapathi, Shuangquan (Sean) Liu, and Jonathan M. Cirella

May 2011

RTI Press

RTI INTERNATIONAL

About the Authors

Tennyson X. Chen, MS, is a senior system architect and database manager in RTI International's Research Computing Division. His main focus is the National Survey of Drug Use and Health (NSDUH) project.

Martin D. Meyer, PhD, is also a senior system architect and database manager in RTI International's Research Computing Division. He is the NSDUH data processing manager.

Nanthini Ganapathi, MS, is a senior web developer and a NSDUH database designer at RTI International.

Shuangquan (Sean) Liu, PhD, is a senior software developer and a NSDUH database designer at RTI International.

Jonathan M. Cirella, BS, is a software developer and NSDUH database practitioner at RTI International.

RTI Press publication OP-0004-1105

Suggested Citation

Chen, T. X., Meyer, M. D., Ganapathi, N., Liu, S., and Cirella, J. M. 2011. Improving Data Quality in Relational Databases: Overcoming Functional Entanglements RTI Press publication No. OP-0004-1105. Research Triangle Park, NC: RTI Press. Retrieved [date] from http://www.rti.org/rtipress.

This publication is part of the RTI Research Report series. Occasional Papers are scholarly essays on policy, methods, or other topics relevant to RTI areas of research or technical focus.

RTI International
3040 Cornwallis Road
PO Box 12194
Research Triangle Park, NC
27709-2194 USA

Tel: +1.919.541.6000
Fax: +1.919.541.5985
E-mail: rtipress@rti.org
Web site: www.rti.org

doi:10.3768/rtipress.2011.op.0004.1105

www.rti.org/rtipress

Improving Data Quality in Relational Databases: Overcoming Functional Entanglements

Tennyson X. Chen, Martin D. Meyer,
Nanthini Ganapathi, Shuangquan (Sean) Liu, and
Jonathan M. Cirella

Abstract

The traditional vertical decomposition methods in relational database normalization fail to prevent common data anomalies. Although a database may be highly normalized, the quality of the data stored in this database may still deteriorate because of potential data anomalies. In this paper, we first discuss why practitioners need to further improve their databases after they apply the traditional normalization methods, because of the existence of functional entanglement, a phenomenon we defined. We outline two methods for identifying functional entanglements in a normalized database as the first step toward data quality improvement. We then analyze several practical methods for preventing common data anomalies by eliminating and restricting the effects of functional entanglements. The goal of this paper is to reveal shortcomings of the traditional database normalization methods with respect to the prevention of common data anomalies, and offer practitioners useful techniques for improving data quality.

Contents

RTI INTERNATIONAL

Introduction

Today's computer systems can store massive amounts of data of all types. To make searching enormous databases more efficient, programmers rely on a collection of tables, often called "relations." A database built around such tables is called a "relational database," and such databases are built to handle all types of data. For example, a survey system may create a searchable collection of tables that records respondent address, phone number, sex, age, survey questions, survey answers, and other information. A company database may have records representing employee name, office number, telephone number, e-mail, supervisor name, supervisor telephone number, and supervisor email, among other information. Relational databases provide a fast and efficient way to store and retrieve electronic data in modern computer systems.

Database designers create and work with relational databases on a regular basis. However, these practitioners can face numerous problems when building a database using tables. One common problem is data redundancy, which occurs when data are duplicated in a database table, or relation. These duplicated data can cause anomalies that affect data quality and provide users with incorrect information. Therefore, practitioners must follow certain rules while designing and normalizing their databases.

The traditional method of preventing data redundancy and the resulting anomalies is called "database normalization." In the normalization process, practitioners examine functional dependency, multi-valued dependency, project-join dependency, and several other data dependencies to decompose a relation into multiple relations. The end result of normalization is a set of relations that meet the requirements of different levels of the normal form. The higher the level of the normal form we reach in a database, the lower the possibility that data anomalies can occur.

The normalization process is well defined in the literature and is commonly understood by database designers. Our experience and recent research[1,2] indicate that basic data anomalies can still exist in high-level normal forms even if the database meets all traditional normalization requirements. Consequently, in addition to following the traditional normalization methods, practitioners should strive to identify functional entanglements in a database and restrict their effects. By doing so, practitioners can significantly improve data quality.

In this paper, we continue to build on these research findings. We first discuss methods of identifying functional entanglements in a normalized database. Then we analyze several practical approaches for restricting the potential effects of these functional entanglements. The discussion provides practitioners with tools they can use to improve database design and implementation, extending what is typically done during the traditional normalization process.

Through the remainder of this manuscript, we use an *italic font* to represent the names of relations and fields in all our discussions and examples.

Data Dependencies and Relation Decomposition

Table 1 illustrates data redundancy in the *Supervisors* relation. Notice that the employees in the first two rows have the same supervisor with the same telephone number.

Table 1. *Supervisors:* display of data redundancy and anomaly

EmployeeName	SupervisorName	SupervisorPhone
Cooper	**Davis**	**888-1111**
Smith	**Davis**	**888-1111**
Barton	**Davis**	**888-1212**
Armstrong	Taylor	888-2222

In this relation, identical information on (*SupervisorName, SupervisorPhone*) can appear multiple times. This is an example of data redundancy. If one of the telephone numbers in the first two rows is accidentally changed, or if a new employee with the same supervisor (Davis) but a different phone number is inserted, the relation will provide users with inconsistent information on the telephone number for this supervisor.

In the third record of this relation, the employee has the same supervisor as the first two records, but for some reason the telephone number is different. This is a typical example of a data anomaly that can occur when a record is inserted or updated. Here, data redundancy directly affects data quality for users by providing conflicting information. Removing data redundancies in a relational database is an important technique for improving data quality.

Traditionally, relational database design relies on a method called "vertical decomposition" to normalize relations and eliminate redundancies. To illustrate the basic mechanism of this method, we can decompose the *Supervisors* relation into two smaller relations, *SupervisorsNew* and *SupervisorPhones*, as shown in Table 2 and Table 3, respectively.

Table 2. *SupervisorsNew:* result of decomposition

EmployeeName	SupervisorName
Cooper	**Davis**
Smith	**Davis**
Barton	**Davis**
Armstrong	**Taylor**

Table 3. *SupervisorPhones:* result of decomposition

SupervisorName	SupervisorPhone
Davis	888-1111
Taylor	888-2222

After this decomposition, the system can retrieve the supervisor telephone number for a particular employee by first joining the two normalized relations through the common field *SupervisorName*. The decomposition removes the data redundancy, does not remove information presented in the original relation, and eliminates the original potential data anomaly problem. Thus, decomposition is a useful tool in improving data quality.

The above decomposition is based on the concept of functional dependency, which states that the value of one field (X) always determines the value of another field (Y) of the same row in a relation. This is illustrated in the relationship between the fields *SupervisorName* and *SupervisorPhone* in Table 3, with a functional dependency (denoted as $X \rightarrow Y$) existing between these two fields.

Functional dependency is the most common cause of data redundancy. Other less common data dependencies that can cause data redundancy are multi-valued dependency and project-join dependency. Researchers have also discussed other, rarer types of data dependency, such as "template dependency,"[3] "subset dependency,"[4] and "join dependency."[5] The solutions for the problems caused by these data dependencies all rely on vertical decomposition methods.

However, our main discussion focuses on the problems caused by **functional entanglement**, which is in direct contrast to functional dependency. The problems caused by functional entanglement discussed in this paper cannot be solved by vertical decomposition. Thus, other than functional dependency, we do not address the other types of data dependencies that are mentioned in the preceding paragraph.

Based on different types of data dependencies, vertical decomposition methods can normalize a relational model into different levels of normal form, such as Boyce-Codd normal form (BCNF), 4th normal form (4NF), and 5th normal form (5NF). These normal forms are considered high level and are the desired goals of a database normalization process. The normalization process that relies on vertical decomposition to remove data redundancies is well illustrated in the literature by Date,[6] Kroenke,[7] and Ullman,[8] among many others.

Data Redundancy in High-Level Normal Forms

Although vertical decomposition is the most commonly accepted method, its inability to completely eliminate some very common data redundancies is often overlooked. Practitioners should consider steps beyond vertical decomposition to prevent data anomalies.[1,2] To illustrate this need, we present the *Employees* relation described in Table 4.

Table 4. ***Employees:*** **A normalized relation that contains data redundancy**

EmployeeID	Title	Sex	Last Name
101	**Mr**	**M**	Cooper
102	**Mr**	**M**	Smith
103	Dr	F	Armstrong
104	Dr	M	Davis
105	Ms	F	Taylor

In this relation, data redundancy exists between fields *Title* and *Sex* in the first two rows. The value set (Mr, M) in (Title, Sex) appears twice in Table 4, and it would likely occur many times in the entire relation. This kind of data redundancy can cause data anomalies when records are inserted or updated. However, because the value "Dr" in *Title* of rows 3 and 4 is associated with two different values in *Sex*, a functional dependency relationship does not exist between *Title* and *Sex*. Thus, database designers cannot use decomposition methods to normalize this relation despite the common data redundancy we have observed. Therefore, if we assume the field *EmployeeID* can uniquely identify a row in this relation, we can simply designate this field as the primary key for the *Employees* relation. This relation is in 5NF, based on the normal form definition and despite the data redundancy and potential data anomalies.

The domain/key normal form (DKNF) does not permit this kind of data redundancy. A relation is in DKNF if and only if every constraint on the relation is a logical consequence of the definition of keys and domains.[9] A relation "key" is a field or a set of fields whose values can uniquely identify a row in the relation. Because we assume that *EmployeeID* can uniquely identify a row in the *Employees* relation, we can set this field as the relation key. The "domain" of a field is the set of all possible values that the field can take in the relation. For example, it is desirable to set the domain of *Sex* in the *Employees* relation as {M, F}. DKNF requires that all database design specifications be defined in terms of keys and domains. Because the interrelationship between *Title* and *Sex* in this example is not a constraint that can be defined by domains or keys, the *Employees* relation is not in DKNF. However, researchers have noted[10-12] that no direct algorithms exist by which to reach DKNF, and implementing DKNF is impractical for several reasons.

To better understand the data dependency that occurs in Table 4, Chen et al.[1] proposed a concept called "functional independency" that practitioners can apply to prevent data anomalies in database designs. Functional independency is based on the observation that two fields that are not functionally dependent on each other may sometimes still relate to each other: that is, they may not in fact be functionally independent from each other. Functional independency is defined as follows:

Given a relation R, field X of R is functionally independent on field Y of R (noted as X><Y) if and only if, for any instance $x_i \in$ Domain(X) and any instance $y_j \in$ Domain(Y), instance (x_i, y_j) is always valid for set of fields (X, Y) in R.[1]

We noted earlier that the fields *Title* and *Sex* in the *Employees* relation are not functionally dependent on each other. Neither are they functionally independent, because some values in these two fields cannot appear together in a given row. For example, there should never be a record in this relation with value (Ms, M) in (Title, Sex). We define this kind of data dependency, which is between the states of functional dependency and functional independency, as "functional entanglement" throughout this paper.

Functional entanglements, like the interrelationship between *Title* and *Sex* in the *Employees* relation, cannot be automatically removed by the traditional vertical decomposition methods. This characteristic separates functional entanglement from all other data dependencies mentioned in this paper so far.

As we have seen from the *Employees* relation (Table 4), if two fields are not functionally dependent on each other but are interrelated through functional entanglement, data anomalies may occur and data quality can suffer. Database practitioners should be alert to the problems caused by functional entanglements in a normalized database so that they can identify them and restrict their effects.

In the following section, we explore the causes of common functional entanglement and analyze potential practical solutions to improve data quality without the need to comply with the strict and sometimes impossible DKNF criteria.

Method

Throughout this paper, we use a fictitious electronic human resources (eHR) system as an example to illustrate different data anomaly problems and examine potential solutions. In the eHR model, each employee is identified with a unique *EmployeeID*, as we showed in the *Employees* relation in Table 4.

In the following section, we examine several examples to investigate how different types of functional entanglement can exist in high-level normal forms. We follow that discussion with a section describing how practitioners can better deal with weaknesses resulting from functional entanglements.

Because fields in a database model can relate to each other in multiple ways, and because functional entanglements can appear in many different forms, trying to identify all the functional entanglements can be difficult. This is one reason that DKNF is so difficult to achieve. Our objective in this paper is not to provide an optimal or perfect database design methodology. Rather, we aim to help practitioners identify the weaknesses in real-world database models so they can take constructive and important steps to improve data quality.

Identifying Functional Entanglements

In this section, we discuss two methods for identifying some of the most common functional entanglements. The first method is based on detection of "subdomain dependencies."

Detecting Subdomain Dependencies

The cause of the data redundancy problem in the *Employees* relation (Table 4) is a functional dependency relationship between a domain subset of *Title* ({Mr, Ms}) and *Sex* ({M, F}). This phenomenon is called a subdomain dependency and defined formally as follows:

Given a relation R, field Y of R is functionally dependent on field X of R in subdomain (noted as $X \rightarrow^{s} Y$) if and only if,

1. **$X \rightarrow Y$ does not hold, and**
2. **There exists at least one instance $x_i \in$ Domain(X) so that x_i is associated with one and only one Y-value in R.[1]**

In other words, between fields X and Y, functional dependency appears in a subset of instances, but we cannot establish such a relationship for the entire two fields. Because subdomain dependencies have characteristics similar to those of functional dependencies, data redundancies caused by functional dependencies can similarly appear in subdomain dependencies. The partial dependencies and transitive dependencies that are supposedly removed by the low-level 2nd normal form (2NF) and 3rd normal form (3NF) can still appear through subdomain dependencies in high-level normal forms.

Using a decomposition method is feasible only when we can establish functional dependency between two *entire* fields. Owing to the nature of subdomain dependency, in which functional dependency occurs only between domain subsets of the fields but not in their entire domains, database designers cannot use decomposition methods to remove data redundancies caused by this kind of dependency.

Subdomain dependency is a specific type of functional entanglement. Detecting subdomain dependencies is the first extra step a practitioner should consider after having applied the traditional normalization methods. This process is similar to that of identifying functional dependencies. The clues that allow us to detect subdomain dependencies usually reside in the data modeling specifications.

The eHR model we are using as example needs to capture the employment status of each employee. An employee is hired on either a full-time (denoted as FT) or hourly (denoted as PT) basis. The model also needs to keep the ratio of hours to full-time employment for every employee, so that administrators can calculate payment scales and benefits. For a full-time employee, the ratio is always 1. For an hourly employee, this ratio is between 0 and 1. We designed the *Employment* relation shown in Table 5 to capture this information.

Table 5. ***Employment:*** **a normalized relation with subdomain dependency**

EmployeeID	EmploymentStatus	EmploymentRatio
101	FT	1
102	FT	1
103	PT	0.5
104	PT	0.75
105	FT	0.5

In this example, we cannot establish functional dependency between fields *EmploymentStatus* and *EmploymentRatio* because value "PT" in *EmploymentStatus* is associated with multiple values in *EmploymentRatio*. But we can identify the following subdomain dependency:

EmploymentStatus $\rightarrow^{s}$ *EmploymentRatio*

This dependency exists because if *EmploymentStatus* is "FT," then *EmploymentRatio* must be 1. With this subdomain dependency, data anomalies, such as the record with *EmployeeID* 105 in Table 5, may occur after record insertion or update, despite the fact that we can designate *EmployeeID* as the primary key and this relation is in 5NF. The clue to help identify this subdomain dependency is within the statement in the requirement specifications: "For a full-time employee, the ratio is always 1."

Subdomain dependencies are a common problem in relational databases. If a practitioner does not make an extra effort to identify and restrict their effects, data redundancies can cause significant data quality problems.

As another example of the complexities of these potential dependencies, the eHR model needs to store address information for the employees. Each address record contains street number, street name, city, state, and zip code. Some zip codes are uniquely associated with certain cities and states. Some cities have multiple zip codes, and a zip code can sometimes be used for multiple cities in the same state. One zip code can also potentially cross state lines in special areas such as military bases. Finally, the same city name can appear in different states. We designed the *EmployeeAddresses* relation shown in Table 6 to store the address information.

Because of the specified requirements of city, state, and zip code, we cannot establish functional dependencies among these three fields because some zip codes are used for more than one city or state. Although this relation is in 5NF with *EmployeeID* as the primary key, the following subdomain dependencies exist.

Zip Code $\rightarrow^{s}$ *State*, and

Zip Code $\rightarrow^{s}$ *City*.

This situation arises because of the requirement "some zip codes are uniquely associated with certain cities and states," while some zip codes can be used for multiple cities or even states. From this example we can again observe: one can spot subdomain dependency information from the design requirements that describe the characteristics of the data.

Because of these subdomain dependencies, data redundancies are widespread in the *EmployeeAddresses* relation. The same value in (*City, State, Zip Code*) appears multiple times, as in rows 1 and 2, and redundancy on (*State, Zip Code*) is visible in rows 4 and 5. These data redundancies are the prime sources

Table 6. ***EmployeeAddresses:*** **another example of subdomain dependency**

EmployeeID	Street Number	Street Name	City	State	Zip Code
101	101	1st Ave.	Cleveland	AL	35049
102	233	2nd Ave.	Cleveland	AL	35049
103	125	A Drive	Birmingham	AL	35215
104	256	B Drive	Hoover	AL	35216
105	54	C Drive	Birmingham	AL	35216
106	808	White Rd.	**Birmingham**	**AL**	**35049**

for data anomalies. The record with *EmployeeID* 106 is an example of data anomaly among city, state, and zip code. In reality, zip code 35049 is not valid for Birmingham, AL, but the current design cannot prevent this record from being introduced into the relation by record insertion or update.

Subdomain dependencies are a common phenomenon in a relational database. Practitioners should attempt to detect subdomain dependencies after applying the traditional database normalization methods.

Identifying Restricted Domains

Functional entanglements can appear in forms other than subdomain dependency. To illustrate these functional entanglements, we first introduce two terms related to field domain—"unrestricted domain" and "restricted domain." Specifically, for any given field, if all possible values in its domain can be assigned to any record in this field, we call the domain of this field an unrestricted domain; otherwise, we call it a restricted domain.

The value in a field of a given row can be restricted in one of two ways: (1) by other values in the same field of other records, or (2) by the values in another field of the same record. We can observe these two types of restriction in the example that follows.

The eHR system needs to keep track of the total number of advanced degrees each employee has obtained; it also needs to record, among those degrees, how many are related to information technology (IT). We can design a 5NF relation denoted *Degrees* (*EmployeeID, Total_Degree, IT_Degree*), with EmployeeID as the primary key. The domains of all three fields are nonnegative integers. *Total_Degree* and *IT_Degree* give the total number of advanced degrees and IT-related advanced degrees, respectively, per employee. Table 7 shows a few rows of this relation.

Table 7. *Degrees:* a relation with restricted domains

EmployeeID	Total _Degree	IT_Degree
101	1	0
102	1	1
103	**1**	**2**

In this relation, the *EmployeeID* field has a restricted domain because of the uniqueness requirement of its values. If we need to insert a new row in Table 7, the *EmployeeID* of the new record must not duplicate any of the values that have already been assigned to other records in this field. Therefore, the domain of this field is restricted by other existing values in the same field.

Total_Degree and *IT_Degree* also have restricted domains. This is because in any given row, the value of *IT_Degree* must be no greater than the value of *Total_Degree*. Once we assign a value to *Total_Degree*, we cannot assign a greater value to *IT_Degree* in the same row. Similarly, if we have assigned a value to *IT_Degree* in a given record, we cannot assign a smaller value to *Total_Degree* of the same record. Said another way, the value (1, 2), shown for *EmployeeID* 103 in Table 7, is invalid for (*Total_Degree, IT_Degree*); the person could not have more advanced IT degrees than total professional degrees. In short, not all possible values in the two domains are free to be assigned to all rows of these two fields in this relation. The values in these two fields are causing domain restrictions for each other. Because our goal is to identify functional entanglements among fields, we are interested only in the restricted domains that are caused by values in other fields of the same row.

The subdomain dependency discussed in the *Employment* relation (Table 5) and the *EmployeeAddress* relation (Table 6) is a *specific* appearance of restricted domain. In the *Employment* relation, the fields *EmploymentStatus* and *EmploymentRatio* are restricting each other's domain. However, the interrelationship between *Total_Degree* and *IT_Degree* is not one of subdomain dependency. In subdomain dependency, some values in a field can uniquely determine the values in another field. However, the value of *Total_Degree* or *IT_Degree* in a given row can determine only the range of values in the same row of the other field, not the specific value. Any time that a field's domain is restricted by the values of another field in a relation, functional entanglements and potential data anomalies exist. In the *Degrees* relation of Table 7, the value (1, 2) is a data anomaly for (*Total_Degree, IT_Degree*). No mechanisms in the current design can prevent this value from being inserted or updated into the *Degrees* relation, which is in 5NF.

Analyzing the domain of each field in a relation can help root out functional entanglements. If database designers detect any fields with restricted domains caused by the values of another field, then they need to provide an extra mechanism to prevent data anomalies. We can look at another example to illustrate how analyzing the domains of fields can help identify functional entanglements in a relation. So far, in all our examples, functional entanglements among fields appear within the same relations. Sometimes, however, similar functional entanglements can come from different relations, as illustrated by considering salary information as yet another part of the hypothetical eHR system.

The eHR system keeps salary information for all employees. Each employee belongs to a salary grade that is identified with an alpha character, and each employee has a current (annual) salary (denoted here in US$, rounded to the nearest $1,000). In addition, each salary grade is associated with a minimum salary value. The annual salary of an employee must be greater than or equal to the minimum salary of the grade that he or she is in.

We can achieve 5NF by designing two relations: *EmployeeSalary*(*EmployeeID, SalaryGrade, Salary*) as shown in Table 8, and *Grade* (*SalaryGrade, MinimumSalary*), as shown in Table 9.

Table 8. *EmployeeSalary:* functional entanglement caused by restricted domain

EmployeeID	SalaryGrade	Salary
101	A	52,000
102	B	63,000
103	**C**	**68,000**

Table 9. *Grade:* functional entanglement caused by restricted domain

SalaryGrade	MinimumSalary
A	50,000
B	60,000
C	70,000

The functional dependencies in this model are as follows:

EmployeeID → *SalaryGrade*

EmployeeID → *Salary*

SalaryGrade → *MinimumSalary*

The domains of each field are as follows:

EmployeeID	: any nonnegative integers
SalaryGrade	: any single alpha characters
MinimumSalary	: any positive decimal numbers
Salary	: any positive decimal numbers

If we further examine the domain of each field in these two relations, we find two additional requirements:

1. For any row in *EmployeeSalary*, the value of *SalaryGrade* must be one of the values in the field that bears the same name in the *Grade* relation.
2. For any row in *EmployeeSalary*, the value of *Salary* must be greater than or equal to the value in *MinimumSalary* in the row with a matching *SalaryGrade* in the *Grade* relation.

We can address the first requirement by simply using the well-known and commonly used technique of enforcing a foreign key constraint on the field *SalaryGrade* from the *Grade* relation to *EmployeeSalary*. The second requirement indicates that the domains of *SalaryGrade* and *Salary* in the *EmployeeSalary* relation are restricted. The domain of *Salary* in the *EmployeeSalary* relation is restricted by the values in *MinimumSalary* in the *Grade* relation through the common field *SalaryGrade*. Because of these restricted domains, functional entanglements exist in this relation design.

As we can see in Table 8, the third record (namely, EmployeeID 103) contains a data anomaly because the salary is lower than the minimum value of the corresponding grade. Unfortunately, we cannot prevent this record from being inserted or updated into this 5NF relation based on the current design. In short, the correlation between restricted domains and data anomalies is clear.

In conclusion, by detecting subdomain dependencies and identifying restricted domains, we can determine whether a database design contains functional entanglements. In the next section, we analyze practical approaches to eliminate or prevent data anomalies caused by functional entanglements.

Practical Approaches for Preventing Data Anomalies

Chen et al.[2] proposed three practical methods for preventing data anomalies caused by functional entanglements. We summarize them briefly below, explore their applicability, and discuss their strengths and shortcomings in dealing with different types of functional entanglement. Following that discussion, we introduce and analyze another practical method that can be applied to prevent data anomalies.

Preventing Data Anomalies by Changing Relation Design

As shown in the previous section, after having normalized a database into BCNF or even 5NF with the vertical decomposition method, database designers need to take further steps to refine their databases to prevent data anomalies. One of these steps is to analyze and modify the data model at the design level. Two main options of changing relation design are available to achieve this objective: field-level disentanglement and horizontal decomposition.

Field-Level Disentanglement

The first option, which we call "field-level disentanglement," seeks to untangle data interrelationships at the field level. We can demonstrate this approach by further analyzing the relation *Degrees*(*EmployeeID, Total_Degree, IT_Degree*) shown in Table 7, in which both *Total_Degree* and *IT_Degree* have restricted domains because they do not represent two disjoint subsets in terms of categorical classification of degrees. In essence, *IT_Degree* is a part of *Total_Degree*. Hence, there is a constraint, *Total_Degree* ≥ *IT_Degree*, for all rows. This constraint can lead to data anomalies, although the relation is already in 5NF and cannot be further decomposed.

To remove the restricted domains in this relation, we can change the relation design by using disjoint subsets. In this particular example, we can replace the field *Total_Degree* with *Non_IT_Degree*; this field would then reflect the number of professional degrees that are not IT-related for each employee. The new relation, *EmployeeDegrees*, is as follows:

EmployeeDegrees(EmployeeID, Non_IT_Degree, IT_Degree), where *Total_Degree = Non_IT_Degree + IT_Degree*.

After the redesign, the fields *Non_IT_Degree* and *IT_Degree* represent two disjoint and mutually complementary subsets. Because no restrictions exist on how the values of these two fields can be assigned in any rows, these two fields both have unrestricted domains. This new design eliminates the potential of data anomalies. In addition, because the value of the original *Total_Degree* can be derived from the sum of *Non_IT_Degree*, and *IT_Degree*,, no information is lost from the original relation model. This example demonstrates that both relations *Degrees* and *EmployeeDegrees* are in 5NF, but *EmployeeDegrees* is better than *Degrees* at preventing data anomalies. One important point about this improvement is that *EmployeeDegrees* is a product of relation redesign rather than decomposition.

This example also highlights the importance of further actions after database designers have applied the traditional decomposition methods when designing a database. Practitioners should analyze functional entanglements and continue to refine the design of relations as an integral part of normalizing a database. When a field exists that is a subset of another field in terms of categorical classification, like *Total_Degree* and *IT_Degree* in *Degrees*, or when mathematical or logical interrelationships exist among fields, this relation will usually have insertion and update anomalies that vertical decomposition methods cannot remove. If we identify restricted domains in a relation, we can first consider using the field-level disentanglement method, which redesigns the relation model to improve data quality.

The main advantage of the field-level disentanglement method is that changes are made at the design level: database designers should thus encounter little or no additional cost at the programming or production levels. Once practitioners apply this method successfully, functional entanglements and the resulting insertion and update anomalies no longer exist, even before the database is implemented. This process is efficient and straightforward; it does not require any other database system tools to enhance data quality.

Although the field-level disentanglement method is very effective in disentangling data dependencies caused by categorical classification and logical or mathematical interrelationships, this method does have a disadvantage: its usefulness in other situations is limited. Usually, this method works only when functional entanglements occur at the field level for all rows. For example, it cannot be used to solve the data anomaly problems for the relation *EmployeeAddresses* (Table 6), because the subdomain dependencies only affect a subset of rows.

Horizontal Decomposition

Our second option to eliminate restricted domains at the design level is to use horizontal decomposition. Database designers can apply this method to normalize some relations into DKNF in certain situations.[6] As the name suggests, this method decomposes a relation horizontally by splitting a relation into multiple relations with the same table structure. It targets mainly relations with restricted domains that are caused by a limited number of domain subsets. Practitioners use decomposition along the line of these domain subsets to remove the restrictions on domains.

The relation *Employment*(*EmployeeID, EmploymentStatus, EmploymentRatio*) of Table 5 illustrates this approach. According to the horizontal decomposition method, we can split *Employment* into two smaller relations with identical structure:

- *Employment_WholeRatio(EmployeeID, EmploymentStatus, EmploymentRatio)*, which holds only full-time employees with *EmploymentStatus*="FT" and *EmploymentRatio* =1, and
- *Employment_PartialRatio(EmployeeID, EmploymentStatus, EmploymentRatio)*, which holds only hourly employees with *EmploymentStatus*="PT" and $0 \leq EmploymentRatio \leq 1$.

After the decomposition, the domains of fields *EmploymentStatus* and *EmploymentRatio* in the new relations are no longer restricted, and the new model eliminates potential data anomalies. As a result, subdomain dependency *EmploymentStatus* $\rightarrow^{s}$ *EmploymentRatio* no longer exists in either relation.

Although the horizontal decomposition method is simple and straightforward, its drawbacks may outweigh its benefits. The method has not been commonly adopted since its debut. First, this method increases the complexity of a system and the cost of maintenance. When introducing a new record, the programmer must know in which relation to insert this record. Merely updating the value of a row may also require the row to be moved to another relation. Second, this method may cause the information in the database to be widely scattered. Using this method to decompose the *EmployeeSalary* relation (Table 8), for instance, may produce quite a few decomposed relations, one for each possible salary grade, which makes queries for information extremely difficult. Finally, although the horizontal decompositions appear to have normalized some relations into DKNF, they can also inadvertently introduce a new constraint that requires the decomposed relations to maintain the same or similar structure. If database designers need to change the design of one relation, they must also make the same change to multiple relations. This kind of redundancy adds costs to maintaining the database. Furthermore, this new constraint in itself violates the definition of DKNF.

We demonstrated above that by changing relation design, practitioners can prevent data anomalies in certain situations. We also noted that these methods have limitations. When changing the design is not feasible, we can consider employing some existing relational database management system (RDBMS) tools in the implementation stage.

Next, we demonstrate how to use these standard RDBMS tools to restrict or reduce the effects of functional entanglement to prevent data anomalies. These tools are not new techniques per se, and our intention is not to explain what they are and how they work. Rather, we explain how to use those tools and techniques to handle functional entanglements.

Preventing Data Anomalies by Utilizing RDBMS Tools

Implementing Lookup Relations

One approach based on using RDBMS tools involves enforcing referential integrity and using lookup relations. As the name suggests, lookup is a referencing mechanism that allows the values of certain fields of a relation to be verified by the values of the same fields in another relation.

To establish a lookup relationship, practitioners can first build a lookup relation with all possible values in certain fields. They can then enforce referential integrity on the referencing relation to the lookup relation by using a foreign key constraint. This mechanism ensures that the values of the underlying fields in the referencing relation are validated whenever a row is inserted or updated.

Referential integrity is a well-known and widely used technique in database implementation. It ensures that different relations are properly related in a database.[13] To illustrate how this method can reduce data anomalies, we return to the *EmployeeAddresses* relation (Table 6). In this relation, data anomalies arise because of subdomain dependencies among zip code, city, and state. The first step of improvement is to build an additional relation that holds all possible valid values of zip code, city and, state. Table 10 shows a few records of the relation *Zipcodes* (*Zip Code, City, State*). This relation serves a lookup purpose for the matched fields in the referencing *EmployeeAddresses* relation. In this lookup relation, because no functional dependencies arise among *Zip Code, City,* and *State*, all three fields are part of the primary key.

Table 10. *Zipcodes:* lookup relation with all possible underlining values

Zip Code	City	State
35049	Cleveland	AL
35215	Center Point	AL
35215	Birmingham	AL
35216	Hoover	AL
35216	Birmingham	AL
35220	Center Point	AL

After we build a lookup relation of this sort, we can set up a foreign key constraint so that fields *Zip Code, City,* and *State* in *EmployeeAddresses* are linked to the corresponding fields in *Zipcodes*. A generic structured query language (SQL) statement for defining referential integrity for this model is shown below:

```
ALTER TABLE [dbo].[ EmployeeAddresses] WITH CHECK
ADD CONSTRAINT [FK_EmployeeAddresses_Zipcodes]
FOREIGN KEY          ([Zip Code],
                  [City],
                  [State])
REFERENCES [dbo].[Zipcodes]
                  ([Zip Code],
                   [City],
                   [State])†
```

Once we enforce referential integrity in this way, every time a row in *EmployeeAddresses* is inserted or updated, the *City*, *State*, and *Zip Code* values are validated through the *Zipcodes* relation. The insert or update operation will succeed only when a row with matching values in all three fields in *Zipcodes* exists. By keeping clean records in *Zipcodes*, we can largely prevent invalid entries in the *EmployeeAddresses* relation.

The main advantages of defining referential integrity in a database are improved data quality and consistency. Because referential integrity is declared at the design level, custom programming is not needed. This advantage in turn eliminates unintended programming errors. The data are kept valid and intact by the database itself.

The lookup relation method has some disadvantages. First, subdomain dependencies still exist in *Zipcodes*, and data anomalies may still occur in this lookup relation. If the content in a lookup relation is not valid, then the data quality of the main relation can erode. This erosion can in turn affect the maintenance costs of keeping the correct content in the lookup relation. Second, this method requires an increased level of processing to validate data, and it increases the complexity of the data model by introducing more relations. Third, using cascading deletes and updates, which database servers offer as a built-in option to maintain data integrity, may cause

† When writing SQL statements in this paper, we are using SQL Server 2005 syntax.

inadvertent loss of data without the user's knowledge. Finally, if the lookup values require constant updates or user manipulations, this method will increase the processing burden and possibly cause potential data errors in the main relation.

By understanding the drawbacks of the lookup relation method, we can determine when to apply it. This method works best in situations that have finite, predefined, stable, and standard lookup values, such as zip code data (which can be purchased and do not require constant user manipulations). This method is also helpful for alleviating the problem explained in the *Employees* relation shown in Table 4 by predefining a set of finite values for (*Title, Sex*) in a lookup relation. If building such a lookup relation is impossible or not appropriate, such as in the *Employment* relation of Table 5 and the *EmployeeSalary* relation of Table 8, using this method is not suitable.

Utilizing Check Constraints

In the examples of the *Employment* and *EmployeeSalary* relations above, we demonstrated how to detect functional entanglements by identifying subdomain dependencies and restricted domains. In these two examples, creating lookup relations to prevent data anomalies was impossible because of the infinite number of possible lookup values. However, in similar situations, the "check constraints" method is effective in preventing data anomaly problems.

Check constraints are fast, row-level integrity checks offered by most database servers as a standard, built-in RDBMS tool. Traditionally, RDBMS check constraints are designed to limit the domain scope of a field. Using check constraints can improve domain integrity by limiting a field to a set of allowable values.[14] For example, when implementing the *Employees* relation in Table 4, the data type of field *Sex* can be defined only as a one-character string in the database. Without a check constraint, we can assign any character to this field, which could obviously introduce a considerable number of data problems. If database designers want this field to take only a value of {F, M} and nothing else, a check constraint can be defined as follows:

```
ALTER TABLE [dbo].[Employees] WITH CHECK
ADD CONSTRAINT [CK_Employees_Sex]
CHECK ([Sex] = 'F' or [Sex] = 'M')
```

This same technique can help prevent data anomalies caused by functional entanglements. For example, we can build a check constraint as follows for the *Employment* relation (Table 5):

```
ALTER TABLE [dbo].[Employment] WITH CHECK
ADD CONSTRAINT [CK_StatusRatio]
CHECK (([EmploymentStatus] = 'FT' and
        [EmploymentRatio] = 1) or
       ([EmploymentStatus] = 'PT' and
        [EmploymentRatio] >= 0 and
        [EmploymentRatio <= 1))
```

Once the check constraint is implemented, any attempts to insert or update a row in the *Employment* relation with an invalid combination in *(EmploymentStatus, EmploymentRatio)* will fail. Similarly, a check constraint can be devised to prevent data anomalies in the *EmployeeSalary* relation (Table 8) as follows:

```
ALTER TABLE [dbo].[ EmployeeSalary] WITH CHECK
ADD CONSTRAINT [CK_SalaryGrade]
CHECK (([SalaryGrade] = 'A' and [Salary] >= 50000) or
       ([SalaryGrade] = 'B' and [Salary] >= 60000) or
       ([SalaryGrade] = 'C' and [Salary] >= 70000))
```

When this check constraint is implemented, any attempts to insert or update a row in *EmployeeSalary* with an invalid salary grade, or with an out-of-range salary value, will cause the RDBMS to generate an error and the operation will fail.

The main advantages of the check constraint method are its simplicity and its ability to handle situations that the other methods discussed cannot. This method is considerably faster than using lookup relations and other RDBMS tools.[15] In some instances, the check constraint method can also simplify the relation model. In the example involving the *EmployeeSalary* relation, the *Grade* relation may no longer be needed if the business model does not use it for purposes other than limiting the salary values in the *EmployeeSalary* relation.

The main disadvantage of the check constraint method is that it is difficult to maintain. If frequent changes are needed to the information related to the minimum value of each salary grade in the *EmployeeSalary* relation, or adding or deleting salary grade has to be done on a regular basis, then database administrators will constantly need to change the SQL code in the check constraint definition. The additional custom programming required to implement and maintain check constraints can itself be a source of errors. In addition, with the check constraint method, some useful information is hidden in SQL code, and the information is not available through database queries.

Also, in most database servers, a check constraint is limited to checking information within one relation. Querying information in another relation in a check constraint is impossible.[15] For example, with respect to the salary and grade example, it would be ideal if the check constraint could query into the *Grade* relation to retrieve the value of *MinimumSalary* for a given *SalaryGrade*. This would mean keeping both relations (*EmployeeSalary* and *Grade*) while making the constraint simpler and easier to implement and maintain. It would also mean making the minimum salary information available via database queries. Because this querying mechanism is not possible via a check constraint method, the usefulness of this method is limited.

The check constraint method is effective for enforcing general data validation rules or simple business rules. The checks are usually very easy to implement if the functional entanglement is based on simple logic, such as in the *Employment* relation of Table 5. However, when no clear logic for the rules among fields exists in a relation, such as the *EmployeeAddresses* relation (Table 6), this method cannot apply.

When all the above methods fail to eliminate data anomalies, we recommend that practitioners consider database triggers, as we describe next.

Implementing Database Triggers

A database trigger can be defined as a special type of stored procedure that is executed automatically based on the occurrence of a database event.[14] A trigger ensures that specific events of data insertion, update, or deletion cause the database to automatically execute the programming code written in the corresponding trigger. The stored procedure in a database trigger can perform virtually any operation in a database. Therefore, using database triggers can be a powerful tool in preventing data anomalies. Unlike other RDBMS tools mentioned earlier, the functionality of a database trigger goes beyond enforcing data rules. Therefore, practitioners should exercise caution when designing and implementing them.

A database trigger can perform all data validations that are enforceable with check constraints. An example of how to implement a trigger for the *Employment* relation of Table 5 is as follows:

```
ALTER TRIGGER [dbo].[EmployeeID] ON [dbo].
                        [Employment] AFTER INSERT AS
BEGIN
   SET NOCOUNT ON;
   DECLARE @EmploymentCount int

   SELECT @EmploymentCount = count(*)
   FROM [dbo].Inserted
   WHERE (([EmploymentStatus] = 'FT' and
            [EmploymentRatio] = 1) or
           ([EmploymentStatus] = 'PT' and
            [EmploymentRatio] >= 0 and
            [EmploymentRatio <= 1))

   IF @EmploymentCount <= 0
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR ('Invalid Employment Status or
      out of range Employment Ratio', 10, 1)
   END
END
```

This is a typical insert trigger. The database server executes the above SQL code every time a row is inserted into the *Employment* relation. The content of the inserted row is temporarily stored in a one-row relation named "Inserted" that is available for data validation by the trigger. When this trigger is implemented, any attempts to insert a row in the *Employment* relation with invalid combinations in (*EmploymentStatus, EmploymentRatio*) will fail. A similar mechanism can be implemented separately for other data events such as row update or deletion.

One advantage of database triggers over check constraints is their ability to perform relation queries. With respect to the *EmployeeSalary* relation, for instance, we noted that the ideal approach would be to keep both relations *EmployeeSalary* and *Grade* and build a relationship between them to validate data.

This kind of relationship is not possible with check constraints, but it is with a database trigger, which can be done as follows:

```
ALTER TRIGGER [dbo].[SalaryRequirement] ON [dbo].
                         [EmployeeSalary] AFTER INSERT AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @SalaryCount int

    SELECT @SalaryCount = count(*)
    FROM [dbo].Grade G, Inserted I
    WHERE (I.SalaryGrade = G.SalaryGrade) and
              (I.Salary >= G.MinimumSalary)

    If @SalaryCount <= 0
    BEGIN
        ROLLBACK TRANSACTION
        RAISERROR ('Invalid Salary Grade
             or out of range Salary value', 10, 1)
    END
END
```

When this trigger is implemented, any attempt to insert a row into *EmployeeSalary* will cause the RDBMS to query and compare the salary information with the *MinimumSalary* value in the *Grade* relation based on the value of *SalaryGrade*. If the row that is supposed to be inserted has an invalid salary grade, or has an out-of-range salary value, the RDBMS will generate an error and the insertion will fail. This type of data validation can be a useful tool for maintaining data quality.

The main benefits of using database triggers are flexibility and power. The stored procedure triggered by a data operation of a relation can provoke any SQL code. This allows practitioners to build numerous features, such as a relation query, as discussed earlier, into their databases. Moreover, a trigger can enforce data validation rules at the row level as well as the field level. For example, if a condition is imposed on the *Grade* relation (Table 9) such that the total number of different grades should be 20 or fewer, then this condition would require data validation that this relation can contain only up to 20 rows. No other RDBMS tools, other than an insert trigger, can enforce this requirement. Compared with other RDBMS tools in general, and check constraints in particular, a database trigger offers superior functionality in many aspects. Using triggers correctly often leads to improved data quality.

Nevertheless, implementing database triggers has potential drawbacks. First, comparing the solutions offered by check constraints and database triggers on the *Employment* relation above, we can see that programming a trigger is more complicated than programming a check constraint. Moreover, a check constraint performs data validation on both record insertion and update. Practitioners need to implement triggers for relation insertion and update separately, with very similar code. Consequently, when we need to change data validation rules, we have to alter the program in multiple places.

A second potential drawback is that, because a database trigger is so powerful, it can easily cause unintended consequences. For example, a trigger in a relation can cause a data operation in another relation with a trigger that provokes a data operation in the first relation. This can cause a recursive execution of triggers that may damage or destroy the database. Third, though powerful, database triggers have their own limitations. For example, we have not found an effective solution using a database trigger to address the data anomaly issue in the *EmployeeAddresses* relation (Table 6) without first building a lookup relation, as stated above. The business rules that dictate the relationship among city, state, and zip code are too complicated to be captured in a database trigger without some kind of lookup function.

Conclusion

In creating and maintaining relational databases, merely meeting the traditional normalization requirements is not enough to eliminate some basic data anomalies. Common data anomalies can exist in high-level normal forms because of the existence of functional entanglements. By identifying functional entanglements in a database and restricting their effects, practitioners can greatly improve data quality.

We introduced two different methods to identify functional entanglements by detecting subdomain dependencies and restricted domains. We also examined two methods of eliminating functional entanglements at the design level in a normalized database: field-level disentanglement and horizontal decomposition. Finally, we analyzed three other practical approaches for restricting the effects of functional entanglements with RDBMS tools: building lookup relations, utilizing check constraints, and implementing database triggers.

Each solution presented in this paper has its strengths and shortcomings when handling different types of problems. Based on these strengths and shortcomings, practitioners should carefully evaluate the requirements at hand and apply the most appropriate methods to deal with potential data anomalies.

This paper provides practitioners with some important principles that promote improved database design and implementation by going above and beyond the traditional normalization techniques.

References

1. Chen T, Liu S, Meyer M, Gotterbarn D. An introduction to functional independency in relational database normalization. In: John D, Kerr S. Proceedings of the 45th ACM Southeast Regional Conference (ACMSE 2007); 2007 March 23-24; Winston Salem, NC. New York: ACM; 2007. p. 221-5.
2. Chen T, Meyer M, Ganapathi N. Implementation considerations for improving data integrity in normalized relational databases. In: McGregor JD, chair. Proceedings of the 47th ACM Southeast Regional Conference (ACMSE 2009); 2009 March 19-21; Clemson, SC. New York: ACM; 2009. Article No. 3.
3. Sadri F, Ullman DJ. A complete axiomatization for a large class of dependencies in relational databases. In: Miller RE, Ginsberg S, Burkhard WA, Lipton RJ, chairs. STOC ’80. Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing; 1980 April 28-30; Los Angeles, CA. New York: ACM; 1980. p. 117-22.
4. Sagiv Y, Walecka S. Subset dependencies as an alternative to embedded multivalued dependencies. J Assoc Comput Machinery. 1982;29(1):103-17.
5. Date CJ, Darwen H, Lorentzos N. Temporal data and the relational model. 1st ed. San Francisco: Morgan Kaufmann; 2002.
6. Date CJ. An introduction to database systems. 8th ed. Reading (MA): Addison-Wesley Pub. Co; 1999.
7. Kroenke MD. Database processing: fundamentals, design, implementation. New York: Macmillan Pub. Co; 1992.
8. Ullman DJ. Principles of database systems. Rockville (MD): Computer Science Press; 1982.
9. Fagin R. A normal form for relational databases that is based on domains and keys. ACM transactions of database systems (TODS). 1981;6(3):387-415.
10. Celko J. SQL for smarties: advanced SQL programming. 3rd ed. San Francisco: Morgan Kaufmann; 2005.
11. Chao L. Database development and management. Boca Raton, FL: Auerbach Publications; 2006.
12. Thalheim B. Entity-relationship modeling: foundations of database technology. New York: Springer; 2000.
13. Date CJ. The relational database dictionary. Sebastopol (CA): O’Reilly Media, Inc.; 2006.
14. Rankins R, Bertucci P, Gallelli C, Silverstein AT. Microsoft SQL Server 2005 unleashed. Indianapolis (IN): Sams Publishing; 2007.
15. Nielsen P. SQL Server 2005 bible. New Delhi: Wiley-India; 2006.

Acknowledgments

We would like to thank Craig R. Hollingsworth of RTI's Research Computing Division for making substantial editorial contributions to this paper. We also owe a debt of gratitude to an RTI Press senior editor, Kathleen N. Lohr, for her invaluable suggestions on the style and substance of this paper.

www.rti.org/rtipress **RTI Press publication OP-0004-1105**