Database Processing

Fundamentals, Design, and Implementation

FOURTEENTH EDITION

David M. Kroenke • David J. Auer

ALWAYS LEARNING

PEARSON

OTHER MIS TITLES OF INTEREST

Introductory MIS

Managing Information Technology, 7/e
Brown, DeHayes, Hoffer, Martin & Perkins ©2012

Experiencing MIS, 6/e
Kroenke & Boyle ©2016

Using MIS, 8/e
Kroenke & Boyle ©2016

MIS Essentials, 4/e
Kroenke ©2015

Management Information Systems, 14/e
Laudon & Laudon ©2016

Essentials of Management Information Systems, 11/e
Laudon & Laudon ©2015

IT Strategy, 3/e
McKeen & Smith ©2015

Processes, Systems, and Information: An Introduction to MIS, 2/e
McKinney & Kroenke ©2015

Information Systems Today, 7/e
Valacich & Schneider ©2016

Introduction to Information Systems, 2/e
Wallace ©2015

Database

Hands-on Database, 2/e
Conger ©2014

Modern Database Management, 12/e
Hoffer, Ramesh & Topi ©2016

Database Systems: Introduction to Databases and Data Warehouses
Jukic, Vrbsky & Nestorov ©2014

Essentials of Database Management
Hoffer, Topi & Ramesh ©2014

Database Concepts, 7/e
Kroenke & Auer ©2015

Database Processing, 14/e
Kroenke & Auer ©2016

Systems Analysis and Design

Modern Systems Analysis and Design, 7/e
Hoffer, George & Valacich ©2014

Systems Analysis and Design, 9/e
Kendall & Kendall ©2014

Essentials of Systems Analysis and Design, 6/e
Valacich, George & Hoffer ©2015

Decision Support Systems

Business Intelligence, 3/e
Sharda, Delen & Turban ©2014

Decision Support and Business Intelligence Systems, 10/e
Sharda, Delen & Turban ©2014

Data Communications & Networking

Applied Networking Labs, 2/e
Boyle ©2014

Digital Business Networks
Dooley ©2014

Business Data Networks and Security, 10/e
Panko & Panko ©2015

Electronic Commerce

E-Commerce: Business, Technology, Society, 11/e
Laudon & Traver ©2015

Enterprise Resource Planning

Enterprise Systems for Management, 2/e
Motiwalla & Thompson ©2012

Project Management

Project Management: Process, Technology and Practice
Vaidyanathan ©2013

Normalization Example 1: The "Straight-to-BCNF" Method

Now let's rework this example using the "Straight-to-BCNF" method. SKU and SKU_Description determine all of the columns in the table, so they are candidate keys. Buyer is a determinant, but it does not determine all of the other columns, and hence it is not a candidate key. Therefore, SKU_DATA has a determinant that is not a candidate key and is therefore not in BCNF. It will have modification anomalies.

To remove such anomalies, in step 3A in Figure 3-19, we move the columns of functional dependency whose determinant is not a candidate key into a new table. In this case, we place Buyer and Department into a new table:

BUYER (Buyer, Department)

Next, in step 3B in Figure 3-19, we make the determinant of the functional dependency the primary key of the new table. In this case, Buyer becomes the primary key:

BUYER (Buyer, Department)

Next, following step 3C in Figure 3-19, we leave a copy of the determinant as a foreign key in the original relation. Thus, SKU_DATA becomes SKU_DATA_2:

SKU_DATA_2 (SKU, SKU_Description, *Buyer*)

The resulting tables are thus:

SKU_DATA_2 (SKU, SKU_Description, *Buyer*)
BUYER (Buyer, Department)

where SKU_DATA_2.Buyer is a foreign key to the BUYER table.

Both of these tables are now in BCNF and will have no anomalies due to functional dependencies. For the data in these tables to be consistent, however, we also need to define the referential integrity constraint in step 3D in Figure 3-19:

SKU_DATA_2.Buyer must exist in BUYER.Buyer

This statement means that every value in the Buyer column of SKU_DATA_2 must also exist as a value in the Buyer column of BUYER. Sample data for the resulting tables is the same as shown in Figure 3-21.

Note that both the "Step-by-Step" method and the "Straight-to-BCNF" method produced exactly the same results. Use the method you prefer; the results will be the same. To keep this chapter reasonably short, we will use only the "Straight-to-BCNF" method for the rest of the normalization examples.

Normalization Example 2

Now consider the EQUIPMENT_REPAIR relation in Figure 3-10. The structure of the table is:

EQUIPMENT_REPAIR (ItemNumber, EquipmentType, AcquisitionCost, RepairNumber, RepairDate, RepairCost)

Examining the data in Figure 3-10, the functional dependencies are:

ItemNumber → (EquipmentType, AcquisitionCost)
RepairNumber → (ItemNumber, EquipmentType, AcquisitionCost, RepairDate, RepairCost)

Both ItemNumber and RepairNumber are determinants, but only RepairNumber is a candidate key. Accordingly, EQUIPMENT_REPAIR is not in BCNF and is subject to

modification anomalies. Following the procedure in Figure 3-19, we place the columns of the problematic functional dependency into a separate table, as follows:

EQUIPMENT_ITEM (ItemNumber, EquipmentType, AcquisitionCost)

and remove all but ItemNumber from EQUIPMENT_REPAIR (and rearrange the columns so that the primary key RepairNumber is the first column in the relation) to create:

REPAIR (RepairNumber, *ItemNumber*, RepairDate, RepairCost)

We also need to create the referential integrity constraint:

REPAIR.ItemNumber must exist in EQUIPMENT_ITEM.ItemNumber

Data for these two new relations are shown in Figure 3-22.

BY THE WAY

There is another, more intuitive way to think about normalization. Do you remember your eighth-grade English teacher? She said that every paragraph should have a single theme. If you write a paragraph that has two themes, you should break it up into two paragraphs, each with a single theme.

The problem with the EQUIPMENT_REPAIR relation is that it has two themes: one about repairs and a second about items. We eliminated modification anomalies by breaking that single table with two themes into two tables, each with a single theme. Sometimes, it is helpful to look at a table and ask, "How many themes does it have?" If it has more than one, then redefine the table so that it has a single theme.

Normalization Example 3

Consider now the Cape Codd database ORDER_ITEM relation with the structure:

ORDER_ITEM (OrderNumber, SKU, Quantity, Price, ExtendedPrice)

with functional dependencies:

(OrderNumber, SKU) → (Quantity, Price, ExtendedPrice)
(Quantity, Price) → ExtendedPrice

FIGURE 3-22
The Normalized EQUIPMENT_ITEM and REPAIR Relations

EQUIPMENT_ITEM

	ItemNumber	EquipmentType	AcquisitionCost
1	100	Drill Press	3500.00
2	200	Lathe	4750.00
3	300	Mill	27300.00

REPAIR

	RepairNumber	ItemNumber	RepairDate	RepairCost
1	2000	100	2015-05-05	375.00
2	2100	200	2015-05-07	255.00
3	2200	100	2015-06-19	178.00
4	2300	300	2015-06-19	1875.00
5	2400	100	2015-07-05	0.00
6	2500	100	2015-08-17	275.00

This table is not in BCNF because the determinant (Quantity, Price) is not a candidate key. We can follow the same normalization practice as illustrated in examples 1 and 2, but in this case, because the second functional dependency arises from the formula ExtendedPrice = (Quantity * Price), we reach a silly result.

To see why, we follow the procedure in Figure 3-19 to create tables such that every determinant is a candidate key. This means that we move the columns Quantity, Price, and ExtendedPrice to tables of their own, as follows:

EXTENDED_PRICE (Quantity, Price, ExtendedPrice)
ORDER_ITEM_2 (OrderNumber, SKU, *Quantity*, *Price*)

Notice that we left both Quantity and Price in the original relation as a composite foreign key. These two tables are in BCNF, but the values in the EXTENDED_PRICE table are ridiculous. They are just the results of multiplying Quantity by Price. The simple fact is that we do not need to create a table to store these results. Instead, any time we need to know ExtendedPrice we will just compute it. In fact, we can define this formula to the DBMS and let the DBMS compute the value of ExtendedPrice when necessary. You will see how to do this with Microsoft SQL Server 2014, Oracle's Oracle Database, and MySQL 5.6 in Chapters 10A, 10B, and 10C, respectively.

Using the formula, we can remove ExtendedPrice from the table. The resulting table is in BCNF:

ORDER_ITEM_2 (OrderNumber, SKU, Quantity, Price)

Note that Quantity and Price are no longer foreign keys. The ORDER_ITEM_2 table with sample data now appears as shown in Figure 3-23.

Normalization Example 4

Consider the following table that stores data about student activities:

STUDENT_ACTIVITY (StudentID, StudentName, Activity, ActivityFee, AmountPaid)

where StudentID is a student identifier, StudentName is student name, Activity is the name of a club or other organized student activity, ActivityFee is the cost of joining the club or participating in the activity, and AmountPaid is the amount the student has paid toward the ActivityFee. Figure 3-24 shows sample data for this table.

StudentID is a unique student identifier, so we know that:

StudentID → StudentName

However, does the functional dependency exist?

StudentID → Activity

FIGURE 3-23
The Normalized ORDER_ITEM_2 Relation

ORDER_ITEM_2

	OrderNumber	SKU	Quantity	Price
1	1000	201000	1	300.00
2	1000	202000	1	130.00
3	2000	101100	4	50.00
4	2000	101200	2	50.00
5	3000	100200	1	300.00
6	3000	101100	2	50.00
7	3000	101200	1	50.00

FIGURE 3-24
Sample Data for the STUDENT_ACTIVITY Relation

STUDENT_ACTIVITY

	StudentID	StudentName	Activity	ActivityFee	AmountPaid
1	100	Jones	Golf	65.00	65.00
2	100	Jones	Skiing	200.00	0.00
3	200	Davis	Skiing	200.00	0.00
4	200	Davis	Swimming	50.00	50.00
5	300	Garrett	Skiing	200.00	100.00
6	300	Garrett	Swimming	50.00	50.00
7	400	Jones	Golf	65.00	65.00
8	400	Jones	Swimming	50.00	50.00

It does if a student belongs to just one club or participates in just one activity, but it does not if a student belongs to more than one club or participates in more than one activity. Looking at the data, student Davis with StudentID 200 participates in both Skiing and Swimming, so StudentID does *not* determine Club. StudentID does not determine ActivityFee or AmountPaid, either.

Now consider the StudentName column. Does StudentName determine StudentID? Is, for example, the value 'Jones' always paired with the same value of StudentID? No, there are two students named 'Jones', and they have different StudentID values. StudentName does not determine any other column in this table, either.

Considering the next column, Activity, we know that many students can belong to a club. Therefore, Activity does not determine StudentID or StudentName. Does Activity determine ActivityFee? Is the value 'Skiing', for example, always paired with the same value of ActivityFee? From these data, it appears so, and using just this sample data, we can conclude that Activity determines ActivityFee.

However, this data is just a sample. Logically, it is possible for students to pay different costs, perhaps because they select different levels of activity participation. If that were the case, then we would say that

(StudentID, Activity) → ActivityFee

To find out, we need to check with the users. Here, assume that all students pay the same fee for a given activity. The last column is AmountPaid, and it does not determine anything.

So far, we have two functional dependencies:

StudentID → StudentName
Activity → ActivityFee

Are there other functional dependencies with composite determinants? No single column determines AmountPaid, so consider possible composite determinants for it. AmountPaid is dependent on both the student and the club the student has joined. Therefore, it is determined by the combination of the determinants StudentID and Activity. Thus, we can say

(StudentID, Activity) → AmountPaid

So far we have three determinants: StudentID, Activity, and (StudentID, Activity). Are any of these candidate keys? Do any of these determinants identify a unique row? From the data, it appears that (StudentID, Activity) identifies a unique row and is a candidate key. Again, in real situations, we would need to check this assumption out with the users.

STUDENT_ACTIVITY_PAYMENT is not in BCNF because columns StudentID and Activity are both determinants but neither is a candidate key. StudentID and Activity are only part of the candidate key (StudentID, Activity).

BY THE WAY Both StudentID and Activity are *part of* the candidate key (StudentID, Activity). This, however, is not good enough. A determinant must have *all* of the same columns to be the same as a candidate key.

To normalize this table, we need to construct tables so that every determinant is a candidate key. We can do this by creating a separate table for each functional dependency as we did before. The result is:

STUDENT (StudentID, StudentName)
ACTIVITY (Activity, ActivityFee)
PAYMENT (*StudentID*, *Activity*, AmountPaid)

with referential integrity constraints:

PAYMENT.StudentID must exist in STUDENT.StudentID

and

PAYMENT.Activity must exist in ACTIVITY.Activity

These tables are in BCNF and will have no anomalies from functional dependencies. The sample data for the normalized tables are shown in Figure 3-25.

Normalization Example 5

Now consider a normalization process that requires two iterations of step 3 in the procedure in Figure 3-19. To do this, we will extend the SKU_DATA relation by adding the budget code of each department. We call the revised relation SKU_DATA_3 and define it as follows:

SKU_DATA_3 (SKU, SKU_Description, Department, DeptBudgetCode, Buyer)

Sample data for this relation are shown in Figure 3-26. SKU_DATA_3 has the following functional dependencies:

SKU → (SKU_Description, Department, DeptBudgetCode, Buyer)
SKU_Description → (SKU, Department, DeptBudgetCode, Buyer)
Buyer → (Department, DeptBudgetCode)
Department → DeptBudgetCode
DeptBudgetCode → Department

FIGURE 3-25
The Normalized STUDENT, ACTIVITY, and PAYMENT Relations

STUDENT

	StudentID	StudentName
1	100	Jones
2	200	Davis
3	300	Garrett
4	400	Jones

ACTIVITY

	Activity	ActivityFee
1	Golf	65.00
2	Skiing	200.00
3	Swimming	50.00

PAYMENT

	StudentID	Activity	ActivityFee
1	100	Golf	65.00
2	100	Skiing	200.00
3	200	Skiing	200.00
4	200	Swimming	50.00
5	300	Skiing	200.00
6	300	Swimming	50.00
7	400	Golf	65.00
8	400	Swimming	50.00

FIGURE 3-26
Sample Data for the SKU_DATA_3 Relation

SKU_DATA_3

	SKU	SKU_Description	Department	DeptBudgetCode	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	BC-100	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	BC-100	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	BC-100	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	BC-100	Nancy Meyers
5	201000	Half-dome Tent	Camping	BC-200	Cindy Lo
6	202000	Half-dome Tent Vestibule	Camping	BC-200	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	BC-300	Jerry Martin
8	302000	Locking Carabiner, Oval	Climbing	BC-300	Jerry Martin

Of the five determinants, both SKU and SKU_Description are candidate keys, but Buyer, Department, and DeptBudgetCode are not candidate keys. Therefore, this relation is not in BCNF.

To normalize this table, we must transform this table into two or more tables that are in BCNF. In this case, there are two problematic functional dependencies. According to the note at the end of the procedure in Figure 3-19, we take the functional dependency whose determinant is not a candidate key and has the largest number of columns first. In this case, we take the columns of

Buyer → (Department, DeptBudgetCode)

and place them in a table of their own.

Next, we make the determinant the primary key of the new table, remove all columns except Buyer from SKU_DATA_3, and make Buyer a foreign key of the new version of SKU_DATA_3, which we will name SKU_DATA_4. We can also now assign SKU as the primary key of SKU_DATA_4. The results are:

BUYER (Buyer, Department, DeptBudgetCode)
SKU_DATA_4 (SKU, SKU_Description, *Buyer*)

We also create the referential integrity constraint:

SKU_DATA_4.Buyer must exist in BUYER.Buyer

The functional dependencies from SKU_DATA_4 are:

SKU → (SKU_Description, Buyer)
SKU_Description → (SKU, Buyer)

Because every determinant of SKU_DATA_4 is also a candidate key, the relationship is now in BCNF. Looking at the functional dependencies from BUYER we find:

Buyer → (Department, DeptBudgetCode)
Department → DeptBudgetCode
DeptBudgetCode → Department

BUYER is *not* in BCNF because neither of the determinants Department and DeptBudgetCode are candidate keys. In this case, we must move (Department, DeptBudgetCode) into a table of its own. Following the procedure in Figure 3-19 and breaking BUYER into two tables (DEPARTMENT and BUYER_2) gives us a set of three tables:

DEPARTMENT (Department, DeptBudgetCode)
BUYER_2 (Buyer, *Department*)
SKU_DATA_4 (SKU, SKU_Description, *Buyer*)