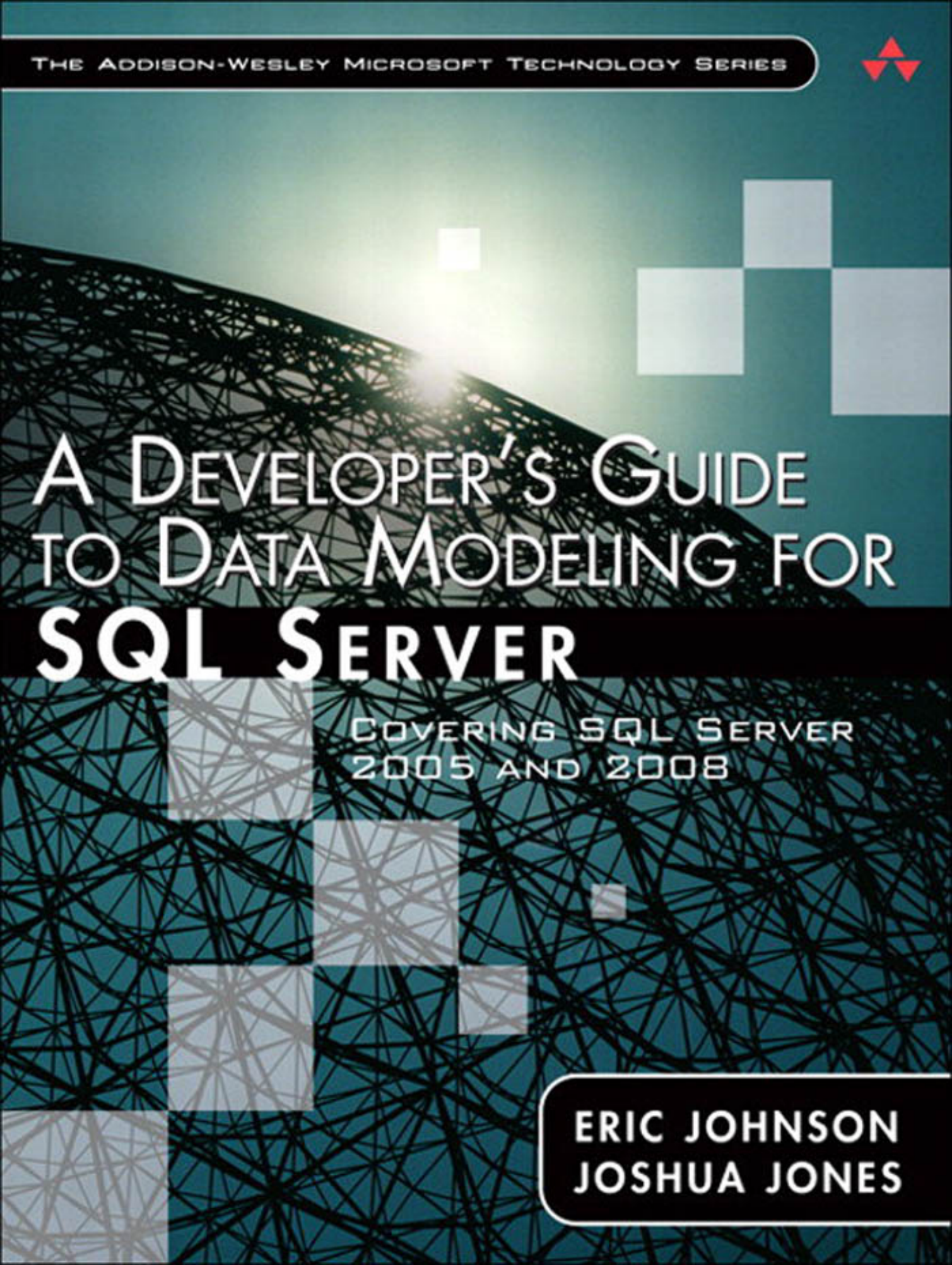
THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES
A DEVELOPER'S GUIDE TO DATA MODELING FOR SQL SERVER
COVERING SQL SERVER 2005 AND 2008
ERIC JOHNSON
JOSHUA JONES

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Johnson, Eric, 1978–
A developer's guide to data modeling for SQL server : covering SQL server 2005 and 2008 / Eric Johnson and Joshua Jones. — 1st ed.
p. cm.
Includes index.
ISBN 978-0-321-49764-2 (pbk. : alk. paper)
1. SQL server. 2. Database design. 3. Data structures (Computer science) I. Jones, Joshua, 1975- II. Title.

QA76.9.D26J65 2008
005.75'85—dc22 2008016668

ISBN-13: 978-0-321-49764-2
ISBN-10: 0-321-49764-3
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, June 2008

Preface

As database professionals, we are frequently asked to come into existing environments and "fix" existing databases. This is usually because of performance problems that application developers and users have uncovered over the lifetime of a given application. Inevitably, the expectation is that we can work some magic database voodoo and the performance problems will go away. Unfortunately, as most of you already know, the problem often lies within the design of the database. We often spend hours in meetings trying to justify the cost of redesigning an entire database in order to support the actual requirements of the application as well as the performance needs of the business. We often find ourselves tempering good design with real-world problems such as budget, resources, and business needs that simply don't allow for the time needed to completely resolve all the issues in a poorly designed database.

What happens when you find yourself in the position of having to redesign an existing database or, better yet, having to design a new database from the ground up? You know there are rules to follow, along with best practices that can help guide you to a scalable, functional design. If you follow these rules you won't leave database developers and DBAs cursing your name three years from now (well, no more than necessary). Additionally, with the advent of enterprise-level relational database management systems, it's equally important to understand the ins and outs of the database platform your design will be implemented on.

There were two reasons we decided to write this book, a reference for everyone out there who needs to design or rework a data model that will eventually sit on Microsoft SQL Server. First, even though there are dozens of great books that cover relational database design from top to bottom, and dozens of books on how to performance-tune and write T-SQL for SQL Server, there wasn't anything to help a developer or designer cover the process from beginning to end with the right mix of theory and practical experience. Second, we'd seen literally hundreds of poorly designed databases left behind by people who had neither the background in

database theory nor the experience with SQL Server to design an effective data model. Sometimes, those databases were well designed for the technology they were implemented on; then they were simply copied and pasted (for lack of a more accurate term) onto SQL Server, often with disastrous results. We thought that a book that discussed design for SQL Server would be helpful for those people redesigning an existing database to be migrated from another platform to SQL Server.

We've all read that software design, and relational database design in particular, should be platform agnostic. We do not necessarily disagree with that outlook. However, it is important to understand which RDBMS will be hosting your design, because that can affect the capabilities you can plan for and the weaknesses you may need to account for in your design. Additionally, with the introduction of SQL Server 2005, Microsoft has implemented quite a bit of technology that extends the capabilities of SQL Server beyond simple database hosting. Although we don't cover every piece of extended functionality (otherwise, you would need a crane to carry this book), we reference it where appropriate to give you the opportunity to learn how this functionality can help you.

Within the pages of this book, we hope you'll find everything you need to help you through the entire design and development process—everything from talking to users, designing use cases, and developing your data model to implementing that model and ensuring it has solid performance characteristics. When possible, we've provided examples that we hope will be useful and applicable to you in one way or another. After spending hours developing the background and requirements for our fictitious company, we have been thinking about starting our own music business. And let's face it—reading line after line of text about the various uses for a varchar data type can't always be thrilling, so we've tried to add some anecdotes, a few jokes, and even a paraphrased movie quote or two to keep it lively.

Writing this book has also been an adventure for both of us, in learning how the publishing process works, learning the finer details of writing for a mass audience, and learning that even though we are our own worst critics, it's hard to hear criticism from your friends, even if they're right; but you're always glad that they are.

CHAPTER 4

NORMALIZING A DATA MODEL

Data normalization is probably one of the most talked-about aspects of database modeling. Before building your data model, you must answer a few questions about normalization. These questions include whether or not to use the formal normalization forms, which of these forms to use, and when to denormalize.

To explain normalization, we share a little bit of history and outline the most commonly used normal forms. We don't dive very deeply into each normal form; there are plenty of other texts that describe and examine every detail of normalization. Instead, our purpose is to give you the tools necessary to identify the current state of your data, set your goals, and normalize (and denormalize) your data as needed.

What Is Normalization?

At its most basic level, **normalization** is the process of simplifying your data into its most efficient form by eliminating redundant data. Understanding the definition of the word *efficient* in relation to normalization is the key concept. **Efficiency,** in this case, refers to reducing complexity from a logical standpoint. Efficiency does not necessarily equal better performance, nor does it necessarily equate to efficient query processing. This may seem to contradict what you've heard about design, so first let's walk through the concepts in normalization, and then we'll talk about some of the performance considerations.

Normal Forms

E. F. Codd, who was the IBM researcher credited with the creation and evolution of the relational database, set forth a set of rules that define how data should be organized in a relational database. Initially, he proposed three sequential forms to classify data in a database: first normal form

(1NF), second normal form (2NF), and third normal form (3NF). After these initial normal forms were developed, research indicated that they could result in update anomalies, so three additional forms were developed to deal with these issues: fourth normal form (4NF), fifth normal form (5NF), and the Boyce-Codd normal form (BCNF). There has been research into a sixth normal form (6NF); this normal form has to do with temporal databases and is outside the scope of this book.

It's important to note that the normal forms are nested. For example, if a database meets 3NF, by definition it also meets 1NF and 2NF. Let's take a brief look at each of the normal forms and explain how to identify them.

First Normal Form (1NF)

In **first normal form,** every entity in the database has a primary key attribute (or set of attributes). Each attribute must have only one value, and not a set of values. For a database to be in 1NF it must not have any repeating groups. A **repeating group** is data in which a single instance may have multiple values for a given attribute.

For example, consider a recording studio that stores data about all its artists and their albums. Table 4.1 outlines an entity that stores some basic data about the artists signed to the recording studio.

Table 4.1 Artists and Albums: Repeating Groups of Data

Artist Name	Genre	Album Name	Album Release Date
The Awkward Stage	Rock	Home	10/01/2006
Girth	Metal	On the Sea	5/25/1997
Wasabi Peanuts	Adult Contemporary Rock	Spicy Legumes	11/12/2005
The Bobby Jenkins Band	R&B	Live! Running the Game	7/27/1985 10/30/1988
Juices of Brazil	Latin Jazz	Long Road White	1/01/2003 6/10/2005

Notice that for the first artist, there is only one album and therefore one release date. However, for the fourth and fifth artists, there are two albums and two release dates. In practice, we cannot guarantee which release date belongs to which album. Sure, it'd be easy to assume that the first release date belongs to the first album name, but how can we be sure

that album names and dates are always entered in order and not changed afterward?

There are two ways to eliminate the problem of the repeating group. First, we could add new attributes to handle the additional albums, as in Table 4.2.

Table 4.2 Artists and Albums: Eliminate the Repeating Group, but at What Cost?

Artist Name	Genre	Album Name 1	Release Date 1	Album Name 2	Release Date 2
The Awkward Stage	Rock	Home	10/01/2006	NULL	NULL
Girth	Metal	On the Sea	5/25/1997	NULL	NULL
Wasabi Peanuts	Adult Contemporary Rock	Spicy Legumes	11/12/2005	NULL	NULL
The Bobby Jenkins Band	R&B	Running the Game	7/27/1985	Live!	10/30/1988
Juices of Brazil	Latin Jazz	Long Road	1/01/2003	White	6/10/2005

We've solved the problem of the repeating group, and because no attribute contains more than one value, this table is in 1NF. However, we've introduced a much bigger problem: what if an artist has more than two albums? Do we keep adding two attributes for each album that any artist releases? In addition to the obvious problem of adding attributes to the entity, in the physical implementation we are wasting a great deal of space for each artist who has only one album. Also, querying the resultant table for album names would require searching every album name column, something that is very inefficient.

If this is the wrong way, what's the right way? Take a look at Tables 4.3 and 4.4.

Table 4.3 The Artists

ArtistName	Genre
The Awkward Stage	Rock
Girth	Metal
Wasabi Peanuts	Adult Contemporary Rock
The Bobby Jenkins Band	R&B
Juices of Brazil	Latin Jazz

Table 4.4 The Albums

AlbumName	ReleaseDate	ArtistName
White	6/10/2005	Juices of Brazil
Home	10/01/2006	The Awkward Stage
On The Sea	5/25/1997	Girth
Spicy Legumes	11/12/2005	Wasabi Peanuts
Running the Game	7/27/1985	The Bobby Jenkins Band
Live!	10/30/1988	The Bobby Jenkins Band
Long Road	1/01/2003	Juices of Brazil

We've solved the problem by adding another entity that stores album names as well the attribute that represents the relationship to the artist entity. Neither of these entities has a repeating group, each attribute in both entities holds a single value, and all of the previously mentioned query problems have been eliminated. This database is now in 1NF and ready to be deployed, right? Considering there are several other normal forms, we think you know the answer.

Second Normal Form (2NF)

Second normal form (2NF) specifies that, in addition to meeting 1NF, all non-key attributes have a functional dependency on the entire primary key. A **functional dependency** is a one-way relationship between the primary key attribute (or attributes) and all other non-key attributes in the same entity. Referring again to Table 4.3, if ArtistName is the primary key, then all other attributes in the entity must be identified by ArtistName. So we can say, "ArtistName determines ReleaseDate" for each instance in the entity. Notice that the relationship does not necessarily hold in the reverse direction; any genre may appear multiple times throughout this entity. Nonetheless, for any given artist, there is one genre. But what if an artist crosses over to another genre?

To answer that question, let's compare 1NF to 2NF. In 1NF, we have no repeating groups, and all attributes have a single value. However, in 1NF, if we have a composite primary key, it is possible that there are attributes that rely on only one of the primary key attributes, and that can lead to strange data manipulation anomalies. Take a look at Table 4.5, in

Table 4.5 Artists: 1NF Is Met, but with Problems

PK—Artist Name	PK—Genre	Signed Date	Agent	Agent Primary Phone	Agent Secondary Phone
The Awkward Stage	Rock	9/01/2005	John Doe	(777)555-1234	NULL
Girth	Metal	10/31/1997	Sally Sixpack	(777)555-6789	(777)555-0000
Wasabi Peanuts	Adult Contemporary Rock	1/01/2005	John Doe	(777)555-1234	NULL
The Bobby Jenkins Band	R&B	3/15/1985	Johnny Jenkins	(444)555-1111	NULL
The Bobby Jenkins Band	Soul	3/15/1985	Johnny Jenkins	(444)555-1111	NULL
Juices of Brazil	Latin Jazz	6/01/2001	Jane Doe	(777)555-4321	(777)555-9999
Juices of Brazil	World Beat	6/01/2001	Jane Doe	(777)555-4321	(777)555-9999

which we have solved the multiple genre problem. But we have added new attributes, and that presents a new problem.

In this case, we have two attributes in the primary key: Artist Name and Genre. If the studio decides to sell the Juices of Brazil albums in multiple genres to increase the band's exposure, we end up with multiple instances of the group in the entity, because one of the primary key attributes has a different value. Also, we've started storing the name of each band's agent. The problem here is that the Agent attribute is an attribute of the artist but not of the genre. So the Agent attribute is only partially dependent on the entity's primary key. If we need to update the Agent attribute for a band that has multiple entries, we must update multiple records or else risk having two different agent names listed for the same band. This practice is inefficient and risky from a data integrity standpoint. It is this type of problem that 2NF eliminates.

Tables 4.6 and 4.7 show one possible solution to our problem. In this case, we can break the entity into two different entities. The original entity still contains only information about our artists; the new entity contains information about agents and the bands they represent. This technique removes the partial dependency of the Agent attribute from the original entity, and it lets us store more information that is specific to the agent.

Table 4.6 Artists: 2NF Version of This Entity

PK—Artist Name	PK—Genre	SignedDate
The Awkward Stage	Rock	9/01/2005
Girth	Metal	10/31/1997
Wasabi Peanuts	Adult Contemporary Rock	1/01/2005
The Bobby Jenkins Band	R&B	3/15/1985
The Bobby Jenkins Band	Soul	3/15/1985
Juices of Brazil	Latin Jazz	6/01/2001
Juices of Brazil	World Beat	6/01/2001

Table 4.7 Agents: An Additional Entity to Solve the Problem

PK—Agent Name	Artist Name	Agent PrimaryPhone	Agent SecondaryPhone
John Doe	The Awkward Stage	555-1234	NULL
Sally Sixpack	Girth	(777)555-6789	(777)555-0000
Johnny Jenkins	The Bobby Jenkins Band	(444)555-1111	NULL
Jane Doe	Juices of Brazil	555-4321	555-9999

Third Normal Form (3NF)

Third normal form is the form that most well-designed databases meet. 3NF extends 2NF to include the elimination of transitive dependencies. **Transitive dependencies** are dependencies that arise from a non-key attribute relying on another non-key attribute that relies on the primary key. In other words, if there is an attribute that doesn't rely on the primary key but does rely on another attribute, then the first attribute has a transitive dependency. As with 2NF, to resolve this issue we might simply move the offending attribute to a new entity. Coincidentally, in solving the 2NF problem in Table 4.7, we also created a 3NF entity. In this particular case, AgentPrimaryPhone and AgentSecondaryPhone are not actually attributes of an artist; they are attributes of an agent. Storing them in the Artists entity created a transitive dependency, violating 3NF.

The differences between 2NF and 3NF are very subtle. 2NF deals with partial dependency, and 3NF with transitive dependency. Basically, a

partial dependency means that attributes in the entity don't rely entirely on the primary key. Transitive dependency means that attributes in the entity don't rely on the primary key at all, but they do rely on another non-key attribute in the table. In either case, removing the offending attribute (and related attributes, in the 3NF case) to another entity solves the problem.

One of the simplest ways to remember the basics of 3NF is the popular phrase, "The key, the whole key, and nothing but the key." Because the normal forms are nested, the phrase means that 1NF is met because there is a primary key ("the key"), 2NF is met because all attributes in the table rely on all the attributes in the primary key ("the whole key"), and 3NF is met because none of the non-key attributes in the entity relies on any other non-key attributes ("nothing but the key"). Often, people append the phrase, "So help me Codd." Whatever helps you keep it straight.

Boyce-Codd Normal Form (BCNF)

In certain situations, you may discover that an entity has more than one potential, or candidate, primary key (single or composite). **Boyce-Codd normal form** simply adds a requirement, on top of 3NF, that states that if any entity has more than one possible primary key, then the entity should be split into multiple entities to separate the primary key attributes. For the vast majority of databases, solving the problem of 3NF actually solves this problem as well, because identifying the attribute that has a transitive dependency also tends to reveal the candidate key for the new entity being created. However, strictly speaking, the original 3NF definition did not specify this requirement, so BCNF was added to the list of normal forms to ensure that this was covered.

Fourth Normal Form (4NF) and Fifth Normal Form (5NF)

You've seen that 3NF generally solves most logical problems within databases. However, there are more-complicated relationships that often benefit from 4NF and 5NF. Consider Table 4.8, which describes an alternative, expanded version of the Agents entity.

Table 4.8 Agents: More Agent Information

PK—Agent Name	PK—Agency	PK—Artist Name	Agent PrimaryPhone	Agent SecondaryPhone
John Doe	AAA Talent	The Awkward Stage	(777)555-1234	NULL
Sally Sixpack	A Star Is Born Agency	Girth	(777)555-6789	(777)555-0000
John Doe	AAA Talent	Wasabi Peanuts	(777)555-1234	NULL
Johnny Jenkins	Johnny Jenkins Talent	The Bobby Jenkins Band	(444)555-1111	NULL
Jane Doe	BBB Talent	Juices of Brazil	(777)555-4321	(777)555-9999

Specifically, this entity stores information that creates redundancy, because there is a multivalued dependency within the primary key. A **multivalued dependency** is a relationship in which a primary key attribute, because of its relationship to another primary key attribute, creates multiple tuples within an entity. In this case, John Doe represents multiple artists. The primary key requires that the Agent Name, Agency, and Artist Name uniquely define an agent; if you don't know which agency an agent works for and if an agent quits or moves to another agency, updating this table will require multiple updates to the primary key attributes.

There's a secondary problem as well: we have no way of knowing whether the phone numbers are tied to the agent or tied to the agency. As with 2NF and 3NF, the solution here is to break Agency out into its own entity. 4NF specifies that there be no multivalued dependencies in an entity. Consider Tables 4.9 and 4.10, which show a 4NF of these entities.

TABLE 4.9 Agent-Only Information

PK—Agent Name	Agent PrimaryPhone	Agent SecondaryPhone	Artist Name
John Doe	(777)555-1234	NULL	The Awkward Stage
Sally Sixpack	(777)555-6789	(777)555-0000	Girth
John Doe	(777)555-1234	NULL	Wasabi Peanuts
Johnny Jenkins	(444)555-1111	NULL	The Bobby Jenkins Band
Jane Doe	(777)555-4321	(777)555-9999	Juices of Brazil

Table 4.10 Agency Information

PK—Agency	AgencyPrimaryPhone
AAA Talent	(777)555-1234
A Star Is Born Agency	(777)555-0000
AAA Talent	(777)555-4455
Johnny Jenkins Talent	(444)555-1100
BBB Talent	(777)555-9999

Now we have a pair of entities that have relevant, unique attributes that rely on their primary keys. We've also eliminated the confusion about the phone numbers.

Often, databases that are being normalized with the target of 3NF end up in 4NF, because this multivalued dependency problem is inherently obvious when you properly identify primary keys. However, the 3NF version of these entities would have worked, although it isn't necessarily the most efficient form.

Now that we have a number of 3NF and 4NF entities, we must relate these entities to one another. The final normal form that we discuss is **fifth normal form** (5NF). 5NF specifically deals with relationships among three or more entities, often referred to as **tertiary** relationships. In 5NF, the entities that have specified relationships must be able to stand alone as individual entities without dependence on the other relationships. However, because the entities relate to one another, 5NF usually requires a physical entity that acts as a resolution entity to relate the other entities to one another. This additional entity has three or more foreign keys (based on the number of entities in the relationship) that specify how the entities relate to one another. This is how many-to-many relationships (as defined in Chapter 2) are actually implemented. Thus, if a many-to-many relationship is properly implemented, the database is in 5NF.

Frequently, you can avoid the complexity of 5NF by properly implementing foreign keys in the entities that relate to one another, so 4NF plus these keys generally avoids the physical implementation of a 5NF data model. However, because this alternative is not always realistic, 5NF is defined to help formalize this scenario.

Determining Normal Forms

As designers and developers, we are often tasked with creating a fresh data model for use by a new application that is being developed for a specific project. However, in many cases we are asked to review an existing model or physical implementation to identify potential performance improvements. Additionally, we are occasionally asked to solve logic problems in the original design. Whether you are reviewing a current design you are working on or evaluating another design that has already been implemented, there are a few common steps that you must perform regardless of the project or environment. One of the very first steps is to determine the normal form of the existing database. This information helps you identify logical errors in the design as well as ways to improve performance.

To determine the normal form of an existing model, follow these steps.

1. Conduct requirements interviews.

   As with the interviews you conduct when starting a fresh design, it is important to talk with key stakeholders and end users who use the application being supported by the database. There are two key concepts to remember. First, do this work before reviewing the design in depth. Although this may seem counterintuitive, it helps prevent you from forming a prejudice regarding the existing design when speaking with the various individuals involved in the project. Second, generate as much documentation for this review as you would for a new project. Skipping steps in this process will lead to poor design decisions, just as it would during a new project.

2. Develop a basic model.

   Based on the requirements and information you gathered from the interviews, construct a basic logical model. You'll identify key entities and their relationships, further solidifying your understanding of the basic database design.

3. Find the normal form.

   Compare your model to the existing model or database. Where are the differences? Why do those differences exist? Remember not to disregard the design decisions in the legacy database. It's important to focus on those differences, because they may stem from specific denormalization steps taken during the initial design, or

they may exist because of information not available to the original designer. Specifically, identify the key entities, foreign key relationships, and any entities and tables that exist only in the physical model that are purely for relationship support (such as many-to-many relationships). You can then review the key and non-key attributes of every entity, evaluating for each normal form. Ask yourself whether or not each entity and its attributes follow the "The key, the whole key, and nothing but the key" ideal. For each entity that seems to be in 3NF, evaluate for BCNF and 4NF. This analysis will help you understand to what depth the original design was originally done. If there are many-to-many relationships, ensure that 5NF is met unless there is a specific reason that 5NF is not necessary.

Identifying the normal form of each entity in a database should be fairly easy once you understand the normal forms. Make sure to consider every attribute: does it depend entirely on the primary key? Does it depend only on the primary key? Is there only one candidate primary key in the entity? Whenever you find that the answer to these questions is no, be sure to look at creating a separate entity from the existing entity. This practice helps reduce redundancy and moves data to each element that is specific only to the entity that contains it.

If you follow these basic steps, you'll understand what forms the database meets, and you can identify areas of improvement. This will help you complete a thorough review—understanding where the existing design came from, where it's going, and how to get it there. As always, document your work. After you have finished, future designers and developers will thank you for leaving them a scalable, logical design.

Denormalization

Generally, most **online transactional processing** (OLTP) systems will perform well if they've been normalized to either 3NF or BCNF. However, certain conditions may require that data be intentionally duplicated or that unrelated attributes be combined into single entities to expedite certain operations. Additionally, **online analytical processing** (OLAP) systems, because of the way they are used, quite often require that data be denormalized to increase performance. **Denormalization,** as the term implies,

is the process of reversing the steps taken to achieve a normal form. Often, it becomes necessary to violate certain normalization rules to satisfy the real-world requirements of specific queries. Let's look at some examples.

In data models that have a completely normalized structure, there tend to be a great many entities and relationships. To retrieve logical sets of data, you often need a great many joins to retrieve all the pertinent information about a given object. Logically this is not a problem, but in the physical implementation of a database, joins tend to incur overhead in query processing time. For every table that is joined, there is usually a cost to scan the indexes on that table and then retrieve the matching data from each object, combine the resulting data, and deliver it to the end user (for more on indexes and query optimization, see Chapter 10).

When millions of rows are being scanned and tens or hundreds of rows are being returned, it is costly. In these situations, creating a denormalized entity may offer a performance benefit, at the cost of violating one of the normal forms. The trade-off is usually a matter of having redundant data, because you are storing an additional physical table that duplicates data being stored in other tables. To mitigate the storage effects of this technique, you can often store subsets of data in the duplicate table, clearing it out and repopulating it based on the queries you know are running against it. Additionally, this means that you have additional physical objects to maintain if there are schema changes in the original tables. In this case, accurate documentation and a managed change control process are the only practices that can ensure that all the relevant denormalized objects stay in sync.

Denormalization also can help when you're working on reporting applications. In larger environments, it is often necessary to generate reports based on application data. Reporting queries often return large historical data sets, and when you join various types of data in a single report it incurs a lot of overhead on standard OLTP systems. Running these queries on exactly the same databases that the applications are trying to use can result in an overloaded system, creating blocking situations and causing end users to wait an unacceptable amount of time for the data. Additionally, it means storing large amounts of historical data in the OLTP system, something that may have other adverse effects, both internally to the database management system and to the physical server resources.

Denormalizing the data in the database to a set of tables (or even to a different physical database) specifically used for reporting can alleviate the

pressure on the primary OLTP system while ensuring that the reporting needs are being met. It allows you to customize the tables being used by the reporting system to combine the data sets, thereby satisfying the queries being run in the most efficient way possible. Again, this means incurring overhead to store data that is already being stored, but often the trade-off is worthwhile in terms of performance both on the OLTP system and the reporting system.

Now let's look at OLAP systems, which are used primarily for decision support and reporting. These types of systems are based on the concept of providing a **cube** of data, whereby the dimensions of the cube are based on fact tables provided by an OLTP system. These **fact tables** are derived from the OLTP versions of data being stored in the relational database. These tables are often denormalized versions, however, and they are optimized for the OLAP system to retrieve the data that eventually is loaded into the cube. Because OLAP is outside the scope of this book, it's enough for now to know that if you're working on a system in which OLAP will be used, you will probably go through the exercise of building fact tables that are, in some respects, denormalized versions of your normalized tables.

When identifying entities that should be denormalized, you should rely heavily on the actual queries that are being used to retrieve data from these entities. You should evaluate all the existing join conditions and search arguments, and you should look closely at the data retrieval needs of the end users. Only after performing adequate analysis on these queries will you be able to correctly identify the entities that need to be denormalized, as well as the attributes that will be combined into the new entities. You'll also want to be very aware of the overhead the system will incur when you denormalize these objects. Remember that you will have to store not only the rows of data but also (potentially) index data, and keep in mind that the size of the data being backed up will increase.

Overall, denormalization could be considered the final step of the normalization process. Some OLTP systems have denormalized entities to improve the performance of very specific queries, but more than likely you will be responsible for developing an additional data model outside the actual application, which may be used for reporting, or even OLAP. Either way, understanding the normal forms, denormalization, and their implications for data storage and manipulation will help you design an efficient, logical, and scalable data model.

Summary

Every relational database must be designed to meet data quality, performance, and scalability requirements. For a database to be efficient, the data it contains must be maintained in a consistent and logical state. Normalization helps reveal design requirements that remove potential data manipulation anomalies.

However, strict normalization must often be balanced against specialized query needs and must be tested for performance. It may be necessary to denormalize certain aspects of a database to ensure that queries return in an acceptable time while still maintaining data integrity. Every design you work on should include phases to identify normal forms and a phase to identify denormalization needs. This practice will ensure that you've removed data consistency flaws while preserving the elements of a high-performance system.

This completes Part I, which has laid the foundation for building an effective data model. Part II begins with Chapter 5, Requirements Gathering, which launches the overall business process of designing and deploying a data model.

INDEX