Describing how to tune a relational database model would not be complete without a description of normalization. I have attempted to simplify normalization. When I attended university, I found myself somewhat befuddled by the complexity of the rules of normalization. Before returning to university as a computer science undergraduate, I had worked with relational databases for the three years before, so I had a good basic understanding of how to build relational database models. I worked with a relational database dictionary-based application. Moreover, I found the rules of the five normal forms to be obtuse, overcomplicated, and simply impossible to decipher at first glance. The usual response of a college student's first reading of the descriptions of each normal form is usually, “Huh?” These descriptions have a tendency to cause immense confusion for both the trained and the untrained eye alike. I probably read those rules two or three times and wondered what all the fuss was about. [1]

Before we examine my off-the-wall interpretation of normalization, let’s look briefly at how normalization and normal forms are formally defined.

1.1 The Formal Definition of Normalization

Normalization is the sequence of steps by which a relational database model is both created and improved upon. The sequence of steps involved in the normalization process is called normal forms. Essentially, normal forms applied during a process of normalization allow creation of a relational database model as a step-by-step progression.

Tuning of a database installation, and particularly, making SQL coding perform well, is heavily dependent on effective entity design. So a good understanding of normalization is essential.
The most commonly applied normal forms are first, second, and third normal forms. Additionally, there are the rarely commercially implemented Boyce-Codd, fourth, fifth, and Domain Key normal forms. The normal forms steps are cumulative upon each other. In other words, each one is applied to the previous step, and the next step cannot be applied until the previous one is implemented. The following examples apply:

- A database model can only have third normal applied when it is in second normal form.
- If a database model is only in first normal form, then third normal form cannot be applied.

That is what is meant by cumulative. The overall result of normalization is removal of duplication and minimizing on redundant chunks of data. The result is better organization and more effective use of physical space, among other factors.

Normalization is not always the best solution. In data warehouses, there is often a completely different approach to database mode design. Normalization is not the only solution. The formal approach to normalization insists on expecting a designer to apply every normal form layer, in every situation. In a commercial environment this is often overzealous application of detail. The trouble with the deeper and more precisely refined aspects of normalization is that normalization tends to overdefine itself simply for the sake of defining itself further.

Before going into the details of normalization, some specifics should be covered briefly, because they are used in the process of explaining normalization and the different normal forms. These specifics include the concepts of anomalies, dependence, and determinance.

### 1.1.1 Anomalies

Relational database modeling eliminates what are called anomalies from occurring in a database. Anomalies can potentially occur during changes to a database. An anomaly is a bad thing because data can become logically corrupted. An anomaly, with respect to relational database design, is essentially an erroneous change to data, more specifically to a single record.

Anomalies apply to any changes to data. Thus insert, delete, and update anomalies can occur as follows:
Insert Anomaly. This is caused when a record is added to a detail table, with no related record existing in a master table. For example, adding a new book first, as written by a specific author, creates a book written by nobody. This is senseless.

Delete Anomaly. This is caused when a record is deleted from a master table, without first deleting all sibling records from detail tables. For example, deleting authors without deleting books first will result in authorless books. A special case is a cascade deletion, where deletion of a master record automatically deletes all child records in all related detail tables, before deleting the parent record in the master table.

Update Anomaly. This anomaly is similar to deletion in that both master and detail records must be updated, in order to avoid orphaned detail records. When cascading, it needs to be ensured that any primary key updates are propagated to related child table foreign keys.

1.1.2 Dependence and Determinance

Dependence implies that a value is dependent on another value. Determinance implies that a value will help to determine the value of another value. These are the details:

- Functional Dependence. Y is functionally dependent on X, if the value of Y is determined by X. In other words, if \( Y = X + 1 \), then the value of X will help to determine the value of Y. Thus, Y is dependent on X as a function of the value of X.

- Determinant. A determinant is the inversion (opposite) of functional dependency. Therefore, if \( Y = X + 1 \), then X is a determinant of Y. This is because X determines the value Y, at least partially because 1 is added to X as well.

- Transitive Dependence. Z is transitively dependent on X when X determines Y, and Y determines Z. Therefore, Z is indirectly dependent on X through its relationship with Y.

- Candidate Key. A candidate key is any column or combination of columns that can be used as a primary key for an entity. A primary key uniquely identifies each record in an entity.

- Full Functional Dependence. This situation occurs where X determines Y, but X combined with Z does not determine Y. In other words, Y
depends on X and X alone. If Y depends on X with anything else, then there is not full functional dependence. Thus, following on from the description of what a candidate key is, if X is the determinant, it cannot be a composite key because a composite key contains more than one column (the equivalent of X with Z).

- **Multivalued Dependence.** A column containing a comma-delimited list (a collection) is a multivalued dependency. All values are dependent as a whole on the primary key.

- **Trivial Multivalued Dependence.** This occurs between two columns when they are the only two columns in the entity. One is the primary key and the other a multivalued list.

- **Nontrivial Multivalued Dependence.** This occurs when there are other columns in an entity in addition to the primary key and a collection.

- **Cyclic Dependence.** This is when X is dependent on Y, which in turn is also dependent on X, directly or indirectly. Cyclic dependence, therefore, indicates a logically circular pattern of interdependence. Cyclic dependence typically occurs with entities containing a composite primary key of three or more columns. For example, three columns in an entity are related in pairs to each other. In other words, X relates to Y, Y relates to Z, and X relates to Z. Ultimately, Z relates back to X.

That covers the definitions of anomalies, dependence, and determinance. Now let’s examine the definitions of normal forms from a formal perspective.

### 1.1.3 First Normal Form (1NF)

First normal form eliminates repeating groups where all records in all entities are identified uniquely by a primary key in each entity. All columns other than the primary key must be dependent on the primary key. First normal form achieves the following:

- Eliminates repeating groups
- Defines primary keys
- All records must be uniquely identified by a primary key. The primary key is unique, prohibiting duplicate values.
1.1 The Formal Definition of Normalization

- All columns other than the primary key must depend on the primary key, either directly or indirectly.
- All columns must contain a single value.
- All values in each column must be of the same datatype.
- Creates a new entity and moves repeating groups to the new entity, removing them from the original entity.

1.1.4 Second Normal Form (2NF)

Second normal form requires that all nonkey values must be fully functionally dependent on the primary key. No partial dependencies are allowed. A partial dependency will exist when a column is fully dependent on a part of a composite primary key. A composite primary key is a primary consisting of more than one column in an entity. Second normal form achieves the following:

- The entity must be in first normal form.
- Removes columns to other entities that are independent of the primary key.
- All nonkey values must be fully functionally dependent on the primary key. In other words, nonkey columns that are not completely and individually dependent on the primary key are not allowed.
- Partial dependencies must be removed. A partial dependency is a special type of functional dependency that exists when a column is fully dependent on a part of a composite primary key.
- Creates a new entity to separate the partially dependent part of the primary key and its dependent columns.

1.1.5 Third Normal Form (3NF)

Third normal form eliminates transitive dependencies where a column is indirectly determined by the primary key. This is because the column is functionally dependent on another column, whereas the other column is dependent on the primary key. Third normal form achieves the following:
The entity must be in second normal form.

- Eliminates transitive dependencies, where a column is indirectly determined by the primary key. This is because that column is functionally dependent on a second column, where that second column is dependent on the primary key.
- Creates a new entity to contain any separated columns.

### 1.1.6 Boyce-Codd Normal Form (BCNF)

In Boyce-Codd normal form, every determinant in an entity is a candidate key. If there is only one candidate key, then third normal form and BCNF are the same. Boyce-Codd normal form achieves the following:

- An entity must be in third normal form.
- An entity can have only one candidate key, where all potential primary keys are separated into separate entities.

### 1.1.7 Fourth Normal Form (4NF)

Fourth normal form eliminates multiple sets of multivalued dependencies. Fourth normal form achieves the following:

- An entity must be in third normal form or Boyce-Codd normal form.
- Multivalued dependencies must be transformed into functional dependencies. This implies that a single value is dependent on the primary key, as opposed to multiple values (a collection) being dependent on the primary key.
- Eliminates multiple sets of multivalued dependencies, sometimes described as nontrivial multivalued dependencies.

### 1.1.8 Fifth Normal Form (5NF)

Fifth normal form eliminates cyclic dependencies. 5NF is also known as Projection normal form (PJNF). The term projection is used to describe new entities containing subsets of data from the original entity. A cyclic dependency is a form of circular dependency, where three pairs result as a combination of a single three-column composite primary key entity, those three pairs being column 1 with column 2, column 2 with column 3, and col-
umn 1 with column 3. In other words, everything is related to everything else, including itself. If normalized entities are joined again using a three-entity join, the resulting records will be the same as that present in the original entity. Fifth normal form achieves the following:

- An entity must be in fourth normal form.
- Cyclic dependencies must be eliminated, where a cyclic dependency is a column that depends on a second column, where the first column is either directly or indirectly dependent on itself.
- The post-transformation join must match records for a query on the pretransformation entity.

### 1.1.9 Domain Key Normal Form (DKNF)

Domain key normal form is the ultimate application of normalization and is more a measurement of a conceptual state of a database model, as opposed to a transformation process in itself. DKNF is the ultimate normal form and describes how a completely normalized database model should be structured:

- Anomalies are not allowed, including insertion, update, or deletion.
- Every record in the database must be directly accessible in all manners, such that no errors can result.
- Every record in every entity must be uniquely identifiable and directly related to the primary key in its entity. Therefore, all columns in all entities are directly determined by the primary keys in their respective entities.
- All validation of data is done within the database model. From a practical perspective, it is prudent to split functionality between database and front-end applications.

Now let’s take a step sideways and try to simplify normalization.

### 1.2 A Layperson’s Approach to Normalization

Many existing commercial relational databases do not go beyond the implementation of third normal form. This is often true of online transaction processing (OLTP) databases and nearly always true in properly designed
data warehouse databases. Application of normal forms beyond third normal form can tend to produce too many entities, resulting in too many entities in SQL query joins. Too many entities in SQL query joins can reduce system performance for any type of database. The more entities in a join, the more difficult queries are to tune. Also, more query complexity makes it more difficult for a database query optimizer to make a best guess at the fastest execution path for a query. The result is poor performance.

From a purely commercial perspective, good performance is much more important than granular perfection in relational database design. It’s not about the design but more about satisfied customers and end users. Poor response time from a computer system will upset people. In fact, poor response time can be much more than simply upsetting because it can impact business and the bottom line for a company.

How can normalization be made simple? Why is it easy? I like to offer a simplified interpretation of normalization to get the novice started. In a perfect world, most relational database model designs are very similar. As a result, much of the basic database design for many applications, such as accounting or manufacturing, is all more or less the same. Some of the common factors are separating repeated columns in master-detail relationships using first normal form, pushing static data into new entities using my version of second normal form, and doing various interesting things with third normal form and beyond.

Normalization is for the most part easy, and largely common sense, with some business knowledge experience thrown in. There are, of course, numerous exceptional circumstances and special cases where my basic interpretation of normalization does not fill all needs up to 100 percent. In these situations, parts of the more refined formal interpretation can be used.

The result is that I have partially redefined the normal forms of normalization, slightly different from what I like to call the formal form of normalization. I have thus redefined normalization into fewer normal forms, which I consider practical for use in a commercial environment. If you find any of my definitions to be contrary to the accepted definitions of normalization, that is because I have deliberately attempted to simplify the various normal form layers for the layperson.

Application of the relational database model to a data set involves the removal of duplication, which is performed using a process called normalization. Normalization consists of a set of rules called Normal Forms. Normalization is applied to a set of data in a database to form entities. Entities are for placing directly associated data into. Entities can be related or linked
to each other through the use of key or index identifiers, which describe a row of data in an entity much like an index is used in a book. An index in a book is used to locate an item of interest without having to read the entire book from cover to cover.

In my version of normalization, there are four levels or layers of normalization I like to call first, second, third, and beyond third normal forms. Each normal form may be a refinement of the previous normal form, although that is not strictly a requirement. In other words, my simple method does not necessarily require cumulative normal forms, although, in most cases, cumulative application of each successive layer makes sense. In designing entities for performance, it is common practice for designers to ignore the steps of normalization and jump directly to second normal form. Third normal form is often not applied either unless many-to-many joins cause an absolute need for unique values at the application level.

Experienced designers make it more of an instinctive process because they have seen similar patterns in data, time and again. This is why I think it is possible to partially rewrite and simplify the normal forms of normalization. My intention is by no means to be bombastic, but only to try to make this process a little easier to understand, and in the context of this book, to allow for ultimately better-performing SQL queries, by reducing the granularity and complexity of the underlying data structures.

**Note:** Overnormalization using third normal forms and beyond can lead to poor performance in both OLTP and data warehouse type databases. Over-normalization is more commercially in top-down designed Java object applications. In this situation, an object structure is imposed onto a relational database. Object and relational data structures are completely different methodologies, because the fine details of granularity are inherent in object modeling. The same is true of extreme application of normal forms, but that creates too many entities, too much processing built into a database model, and ultimately highly complex SQL coding and poor performance as a result.

So I am assuming that normalization in its strictest form is generally impractical because of its adverse effect on performance in a commercial environment, especially fourth normal form and beyond. The simplest way to describe what normalization attempts to achieve can be explained in three ways:
1. Divide the whole into smaller, more manageable parts.

**Note:** The key phrase is manageable parts. There is a sensible balance somewhere between placing detailed granularity in both the database and front-end applications. The most practical form is basic structures in the database model (first, second, and third normal forms), and all other normal forms applied in applications, using application coding.

2. Remove duplicated data into related subsets.

3. Link two indirectly related entities by creating a new entity. The new entity contains indexes (keys) from the two indirectly related entities. This is commonly known as a many-to-many join.

These three points are meaningless without further explanation of normalization, so let’s go through the rules and try to explain normalization in an informal fashion. Let’s start with some relational database buzzwords.

- **An entity** contains many repetitions of the same row. An entity defines the structure for a row. An example of an entity is a list of customer names and addresses.

  **Note:** An entity is also known as a table.

- **A row** is a line of data. Many rows make up the data in an entity. An example of a row is a single customer name and address within an entity of many customers. A row is also known as a record or a tuple.

  The structure of a row in an entity is divided up into **columns**. Each column contains a single item of data such as a name or address. A column can also be called a field or attribute.

- **Referential integrity** is a process of validation between related entities where references between different entities are checked against each other. A **primary key** is placed on a parent or superset entity as the primary identifier or key to each row in the entity. The primary key will always point to a single row only, and it is unique within the entity. A **foreign key** is a copy of a primary key value in a subset or child entity. An example of a function of referential integrity is that it will not allow the deletion of a primary key entity row where a for-
eign key value exists in a child entity. Primary keys are often referred to as PK and foreign keys as FK. Note that both primary and foreign keys can consist of more than one column. A key consisting of more than one column is known as a composite key.

- An index is used to gain fast access to an entity. A key is a special form of an index used to enforce referential integrity relationships between entities. An index allows direct access to rows by duplicating a small part of each row to an additional (index) file. An index is a copy of the contents of a small number of columns in an entity, occupying less physical space and therefore faster to search through than an entity. The most efficient unique indexes are usually made up of single columns containing integers. There are many other types of indexes of various shapes and forms, but specialized indexes such as bitmap indexes have very specific applications.

**Note:** Primary and foreign keys are special types of indexes applying referential integrity. Oracle Database automatically indexes primary keys but not foreign keys.

### 1.2.1 First Normal Form

*First normal form removes repetition by creating one-to-many relationships.* Data repeated many times in one entity is removed to a subset entity, which becomes the container for the removed repeating data. Each row in the subset entity will contain a single reference to each row in the original entity. The original entity will then contain only nonduplicated data. This one-to-many relationship is commonly known as a master-detail relationship, where repeating columns are removed to a new entity. The new entity gets a primary key consisting of a composite of the primary key in the master entity and a unique identifier (within each master primary key) on the detail entity.

In the example in Figure 1.1, a first normal form transformation is shown. The sales order entity on the left contains customer details, sales order details, and descriptions of multiple items on the sales order. Application of first normal form removes the multiple items from the sales order entity by creating a one-to-many relationship between the sales order and the sales order item entities. This has three benefits:
1. Saves space
2. Reduces complexity
3. Ensures that every sales order item will belong to a sales order

In Figure 1.1, the crow’s foot pointing to the sales order item entity indicates that for a sales order to exist, the sales order has to have at least one sales order item. The line across the pointer to the sales order entity signifies that at least one sales order is required in this relationship. The crow’s foot is used to denote an inter-entity relationship.

**Note:** Inter-entity relationships can be zero, one, or many to zero, one, or many.

The relationship shown in Figure 1.1 between the sales order and sales order item entity is that of one to one-or-many.

Example rows for the first normal form structure in Figure 1.1 are shown in Figure 1.2. Notice how the master and detail rows are now separated.

### 1.2.2 Second Normal Form

Second normal form creates not one-to-many relationships but many-to-one relationships, effectively separating static from dynamic information. Static information is potentially repeatable. This repeatable static information is moved into separate entities. In Figure 1.3, the customer information is removed from the sales order entity. Customer information
can be duplicated for many sales orders or have no sales orders, thus the one-and-only-one to zero-one-or-many relationship between customers and sales orders. This many-to-one relationship, as opposed to the one-to-many relationship created by a first normal form transformation, is commonly known as a dynamic-static relationship where repeating values, rather than repeating columns (first normal form), are removed to a new entity. The new entity, containing static data, gets a single column primary key, which is copied to a foreign key in the dynamic entity.

Example rows for the second normal form structure in Figure 1.3 are shown in Figure 1.4. Now we have separation of master and detail rows and a single entry for our customer name; there is no duplication of information. On creation of the Customer entity, one would create a primary key on the
CUSTOMER_NUMBER column, as shown on the right side of the diagram in Figure 1.3. Figure 1.4 does not show a CUSTOMER_NUMBER, but merely a customer name for explanatory purposes. In Figure 1.4, a primary key would be created on the name of the Customer entity. One of the most significant realistic benefits of excessive normalization is saving physical space. However, with the low prices of disk space in the modern world, this is not really too much of an important factor anymore. Processor and memory costs are relatively much more expensive than the costs of storage on disk.

Note: In the previous edition of this book, one of the readers pointed out that the normal form transformation shown in Figures 1.3 and 1.4 is actually a second to third normal form transformation, rather than a first to second normal form transformation. This is because the transformation splits a composite key into two entities. I am attempting to simplify the process of normalization, not rewrite it. Separating static and dynamic data into separate entities is looking at the process from a business operational perspective. The objective is not to contradict the accepted process of normalization, but to make it a little easier for the layperson. This is a simplified interpretation.
1.2.3 Third Normal Form

Third normal form is used to resolve many-to-many relationships into unique values. In Figure 1.5, a student can be enrolled in many courses, and a course can have many students enrolled. It is impossible to find a unique course-student item without joining every student with every course. Therefore, each unique item can be found with the combination of values. Thus the CourseStudent entity in Figure 1.5 is a many-to-many join resolution entity. In a commercial environment, it is very unlikely that an application will ever need to find this unique item, especially not a modern-day Java object Web application, where the tendency is to drill down through list collections rather than display individual items. Many-to-many join resolutions should only be created when they are specifically required by the application. It can sometimes be better to resolve these joins in the application to improve database performance and not create new entities at all.

Note: Be very careful using third normal form and beyond.

Example rows for the third normal form structure in Figure 1.5 are shown in Figure 1.6. Notice how the containment of both students within courses, and courses within students, is provided by the application of third normal form. The question you should ask yourself when using third normal form is this: Does your application need both of these one-to-many relationships? If not, then do not create the new entity, because more entities lead to more complex joins, and thus slower SQL statements. Theoretically, application of third normal form under these circumstances is correct. However, in a commercial application, you will not necessarily need to access the information in both orders.

Now let’s understand this a little further. Look at Figure 1.7. That many-to-many relationship we had between the two entities on the left in...
1.2 A Layperson's Approach to Normalization

Figure 1.6 has disappeared, because the two entities on the left in Figure 1.6 are the same entity; they contain the same data, only in a different order. The courses and students with the contained intra-relationships can be retrieved from a single entity simply by applying a different sort order. Figure 1.7 shows the entities and the number of rows increasing with the application of third normal form. So, not only will we have more complex joins, but we will also have more rows and thus more physical space usage, also leading to slower SQL statements.

There are other applications of the formal method of third normal form, which I have not included here. My simplified interpretation of normalization allows me to push these other applications of third normal form into the realm of beyond third normal form. For example, the formal version of third normal form can be used to extract common columns from two similar entities and create a third entity containing the common columns, including a reference back to the original two entities. Those common columns are then removed from the first two entities. A second example of third normal form application is removal of a transitive dependency. A transitive dependency exists where one column is not completely dependent on the primary key of an entity, perhaps dependent on another column. In other words, an entity contains three columns: column 1 is the primary key; column 2 depends on column 1; and column 3 depends on column 2.
Therefore, column 3 is indirectly dependent on column 1. Thus, column 3 is transitively dependent on column 1 (the primary key), through its dependence on column 2. A third example of third normal form application is that of calculated columns removed altogether from an entity, because they can be recalculated by the columns in the entity that comprise the expression for the calculation. There are also other possibilities.

1.2.4 Beyond Third Normal Form

Many modern relational database models do not extend beyond third normal form. Sometimes, not even third normal form is used, because of the generation of too many entities and resulting complex SQL code joins. The result is poor database response times. This approach applies not only to devolved data warehouse models, but also to very busy OLTP databases, where even in highly accurate, single-record update environments, the extra functionality and accuracy given by beyond third normal form structures can usually be better provided by application coding.

On the contrary, maintenance of data with respect to accessing of individual records in a database can be more effectively and easily managed using beyond third normal form layers. However, any form of querying can be so adversely affected by too many entities that, in most cases, it is better to make single row access work a little bit harder than to cause even the smallest of queries to take an unacceptable amount of time to execute.

Once again, application software development kits (SDKs) are far more powerful as number crunchers than database engine structural and functional capabilities. Extreme implementation of normalization using layers beyond third normal forms can tend to place too much functionality into the database. Often, the most effective approach is to utilize the best of both worlds—combining the benefits of both database and application capabilities. Use the database to mostly just store data, perhaps with some manipulation capabilities. Allow applications to manipulate and verify data to a large extent.

Extreme levels of granularity in relational database modeling are a form of mathematical perfection. It looks nice and feels good to the trained mind. These extremes rarely apply in fast-paced commercial environments. Commercial operations require that a job is done efficiently and cost effectively. Perfection in database model design is a side issue to that of making a profit.

What I like to do with beyond third normal form is to present several cases, where those cases very likely slot into the formal definitions of fourth
normal form, Boyce-Codd normal form, and fifth normal form. Domain key normal form is more a measure of perfection rather than a transformation in itself, so it does not apply in this simplified explanation. In essence, these beyond third normal form transformations should be applied when application requirements make them necessary, rather than for the sake of applying all possible normal forms to a database model.

1.2.4.1 One-To-One NULL Separation Relationships

It is possible to remove potentially NULL-valued fields into a separate entity, creating a one-to-one or zero relationship between parent and child entity. This implies that the child row does not always have to exist. This is usually done to save space in the parent entity. Oracle Database permits variable row lengths because NULL values occupy almost zero physical space. This makes the reduction in physical space usage negligible. This type of normalization therefore does not save any physical space and is pointless. The overall effect of creating these one-to-one NULL separation relationships is yet one more entity to join in queries. Also, because the child entity does not have to contain every row in the parent entity, outer joins will probably result. In my experience, outer joins are often the result of poor entity design, or in this case overgranular (overdetailed) entity design. Disk space is cheap and, as already stated, increased numbers of entities lead to bigger SQL joins and poorer performance.

Figure 1.8 shows an example where it is assumed that because not all customers’ addresses and phone numbers are known, the CUSTOMER_ADDRESS and CUSTOMER_PHONE columns can be moved to a separate entity. Note in this extreme case how the new entity is actually exactly the same structurally as the pretransformation entity, although it could contain fewer rows. In this situation, the amount of data has likely been increased, rather than decreased.
Figure 1.9 shows an even further extreme of one-to-one NULL separation where the CUSTOMER_ADDRESS and CUSTOMER_PHONE columns can be separated into two new entities, resulting in three entities. This can be done because (1) either CUSTOMER_ADDRESS or CUSTOMER_PHONE can be NULL, and (2) the two fields are not directly related to each other, but they are individually dependent on the CUSTOMER_NUMBER primary key column.

Example rows for the diagram shown in Figure 1.8 are shown in Figure 1.10. The amount of data to be stored has actually increased in this case because addresses and phone numbers for all customers are known and stored in the database. Figure 1.9 would be worse given that there would be three tables, instead of two.
1.2.4.2 Separating Object Collections in Entities

Object collections in entities imply multivalued lists, or comma-delimited lists, of same datatype values repeated within a single column of an entity. They are all directly dependent on the primary key as a whole, but not as individual collection elements. In Figure 1.11, employee skill and certification collections are removed into separate entities. An employee could have skills or certifications or both. Thus, there is no connection between the attributes of the Employees entity, other than the employee number, and the details of skills or certifications for each employee.

Example rows for the normalized structure Figure 1.11 are shown in Figure 1.12. The rows are divided from one into three separate entities. Because skills and certifications are duplicated in the normalization of the Employees entity, further normalization is possible but not advisable. Once again, creation of too many entities creates a propensity for complex and slow SQL statements.

**Note:** Oracle Database will allow creation of object collections within entities. The skills and certifications attributes shown on the left of Figure 1.11 are candidates for a contained object collection structure, namely a TABLE, VARRAY, or associated array object datatypes. However, because object and relational structures do not always mix well using an Oracle collection object datatype, this is not necessarily the most efficient storage method for later fast SQL access, even if it is an elegant solution.

1.2.4.3 Multicolumn Composite Keys

Multicolumn composite primary keys normalization divides related columns into separate entities based on those relationships. In Figure 1.13, products, managers, and employees are all related to each other. Thus, three
1.2 A Layperson’s Approach to Normalization

The result is information that can be reconstructed from smaller parts.

Note: All of the columns in the transformed entities on the right side of the diagram in Figure 1.13 consist of composite primary keys.

Example rows for the normalized structure in Figure 1.13 are shown in Figure 1.14. Note how there are two one-to-many relationships on the right side of Figure 1.14; the nature of the data has caused this. In Figure 1.14, it
should be clear that entities divide up data in an elegant and logical way. Once again, this level of normalization will be detrimental to SQL performance because of more complex SQL statement joins and more physical storage space used. There are just too many entities created. Once again, more entities in a data model result in larger and more complex SQL join queries. The more entities in a join there are, the less efficient a query will be, not only because the database optimizer can be overloaded, but also simply because bigger joins are more difficult to write coding for.

### 1.2.4.4 Summarizing a Layperson’s Form of Normalization

- **First normal form** removes repetition by creating one-to-many relationships, separating duplicated columns into a master-detail relationship between two entities.

- **Second normal form** creates not one-to-many relationships but many-to-one relationships, dividing static from dynamic information by removing duplicated values (not columns), into a dynamic-static relationship between two entities.

- **Third normal form** is used to resolve many-to-many relationships into unique values. Third normal form allows for uniqueness of information by creating additional many-to-many join resolution entities. These entities are rarely required in modern-day applications.

- My representation of beyond third normal form is intended to show various specific scenarios that I have seen repeatedly in the past. There are likely many other situations not covered here. Those cov-
1.4 Endnotes