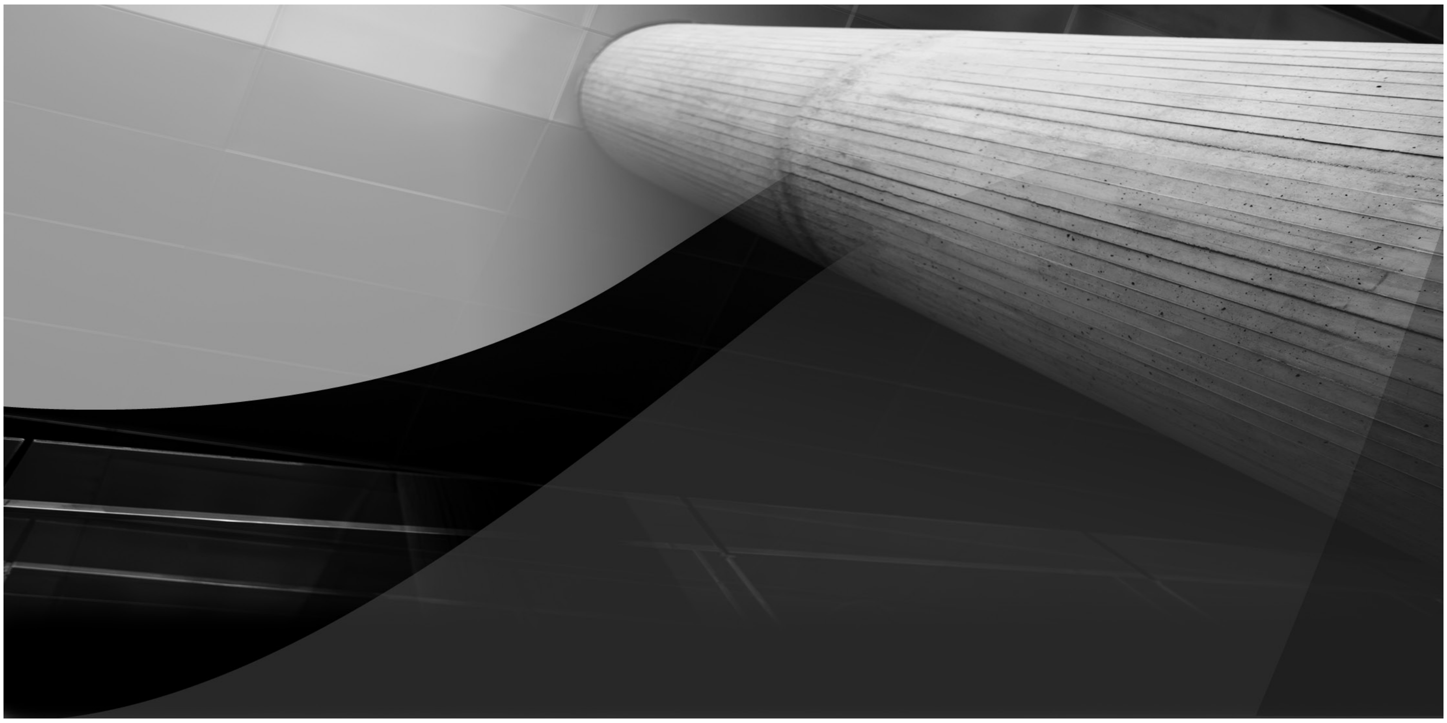
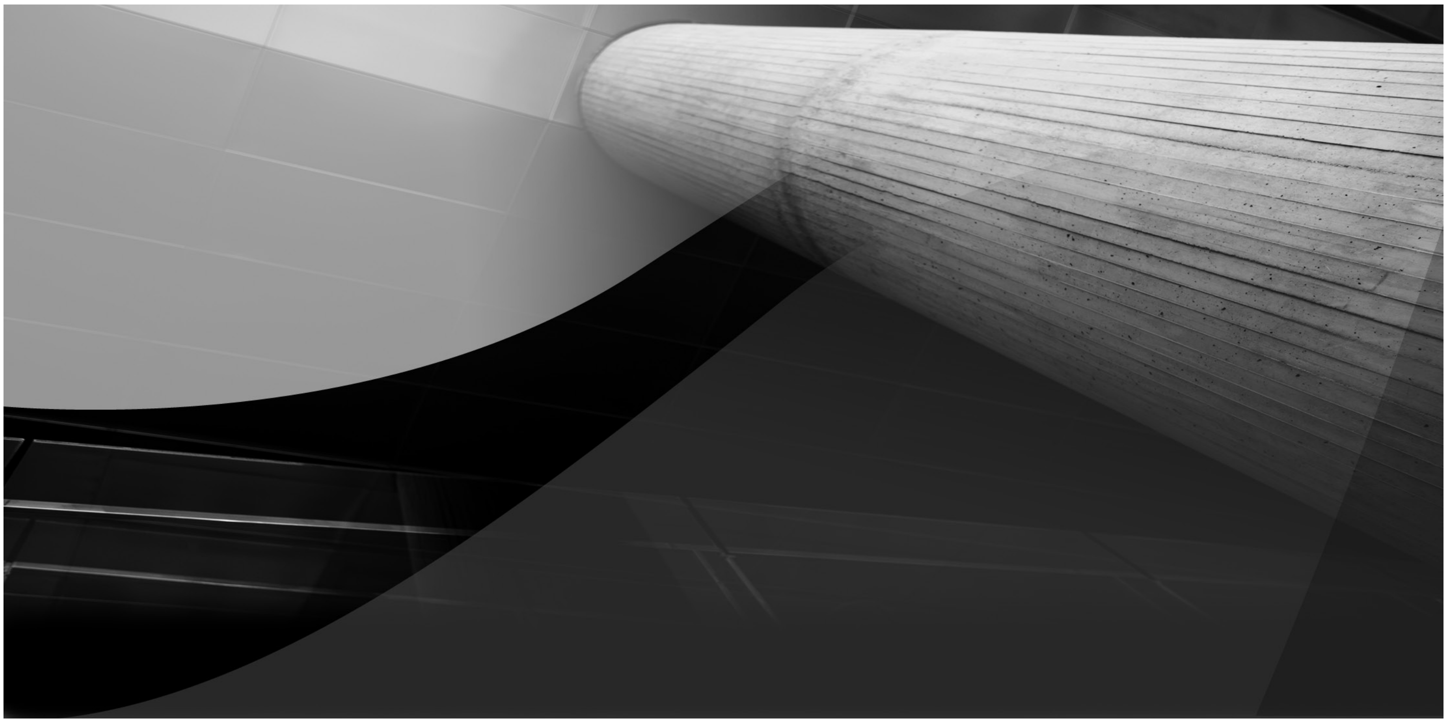


PART II

Data Modeling



CHAPTER
3

Data Modeling Concepts

Data modeling is a process that helps you organize data into relational tables or object types. Together, we'll focus on how to model data into relational tables because MySQL is a relational database.

MySQL Workbench supports SQL development, modeling, and server administration. This chapter moves beyond SQL connections and discusses the modeling component of MySQL Workbench. Here I introduce terms, definitions, and historical details that support how you perform database modeling. Where possible, sidebars hold the historical asides about database modeling and design.

While I considered leaving the asides out as so much baggage, they are in the book for a purpose. They provide key background that supports concepts and ideas of the chapter and details that a number of readers may lack.

This chapter covers the following:

- Data modeling theory
- Data modeling systems
- Classic normalization

It's important to understand that MySQL Workbench can reverse- or forward-engineer relational data models. MySQL Workbench maintains modeling data externally from the database in a single proprietary file format. Data models are stored in this proprietary file format with an `.mwb` file extension.

The three sections of this chapter define concepts for the rest of the book. I strongly encourage you to read this chapter before moving on to the specifics of how to use MySQL Workbench features.

Data Modeling Theory

Data modeling deals with how you organize information into subjects, known as tables. It follows rules that come from academic disciplines, like measurement science, linear algebra, and relational calculus. These are subdisciplines in mathematics. The traditional math approach often makes data modeling too hard for most people who simply want to solve practical problems. My solution to that is working through these concepts with words and analogies.

This section breaks data modeling theory down into a series of questions and answers. It introduces the rule of one, which is critical to modeling data correctly. This section also intends to lay a foundation that lets you understand symbol sets. Symbol sets are how you draw pictures to represent data models. Symbol sets define relationships in math and both classic and domain key database normalization.

The questions for this section are

- What is data modeling?
- Why is data modeling important?
- How do we accomplish it?

After we examine these questions and their answers, the mechanics throughout the rest of the chapter should make logical sense. They should also encourage you to think about data before you rush to organize it.

What Is Data Modeling?

Data modeling is finding *"the one"* in every aspect of information that runs your business. The one means one thing, theme, or subject. You find the one because relational models require that we decompose (break down into little pieces) information into its smallest parts. Only the smallest parts ensure that you can read and write data without the risk of insertion, update, or deletion anomalies, which cause errors or data corruption.

Data modeling is the process of analyzing data required to perform tasks. Let's examine the task of mailing a package to someone. It includes the process of choosing a carrier (the post office, FedEx, or some other carrier) and tracking the package to a delivery point. If we're mailing a gift to a friend, the data may be simple and not worth tracking, but if we're in the business of shipping goods sold by our business, we have a cost to many aspects of this process. That makes the details of the process very important to the data model. For example, one aspect of shipping a package as a vendor is an RMA (Return Merchandise Authorization). RMAs implement a business process when the customer decides they don't want one or more items you've already shipped. An RMA isn't something you consider when mailing a gift to a friend.

Each aspect, characteristic, or attribute of a business process is a task within a shipping process. These characteristics are objects when you analyze a problem using Object-oriented Analysis and Design (OOAD) methods; they are tasks when we apply project management skills. Like most project management tasks, they are composed of subtasks; and the collection of subtasks defines the domain of the task.

A domain defines all the process facts and their relationships. The individual facts become attributes (or columns), collections of related attributes become table structures, and collections of related tables become our domain of knowledge for a relational model. An OOAD approach would also qualify the methods that act on the data. Fortunately, these methods don't fit in the data models of a relational database, and you won't have to model them in the examples of this book.

Scalar Data Types

Scalar data types hold one thing among a set of like things. This means a scalar data type could apply to integers, real numbers, characters, dates, or date-time values. The Java programming language labels scalar data types as primitives.

Attributes are indivisible facts stored as scalar data types. That means you store a single string in a single-variable-length string data type or column, not a list of comma-delimited names in a single column. Indivisible facts are often expressed as atomic data, which means it can't be broken up into parts. That having been said, atomic is a poor word choice because you can now break atoms apart into quarks. When database modelers chose it, atomic was both a descriptive label and the smallest irreducible piece of matter. That's why indivisible fact substitutes for atomic.

Tables define a group or collection of related facts, which are also known as attributes. This makes the table into a structure of a single subject.

The structure defines an object type, and each table can contain many rows that meet the table definition or implement the object type. This makes each row in a table an instance of an object, and the table as a whole an unordered list of objects. It's important to note that tables are unordered lists because the SQL definition says there should be no ordering of rows in a table.

The rule on no ordering covers how databases store and access rows. It doesn't prohibit the ordering of result sets from queries. In fact, the purpose of the `ORDER BY` clause is to order result sets.

Object Types Are Blueprints

Object types are definitions of structures, which are typically a collection of attributes and related methods in OOAD. As such, they serve as a blueprint for building instances of objects, just like a blueprint serves to let builders assemble a Lego set, a ready-to-assemble piece of furniture, or a home.

Following the *rule of one* (finding "the one") means each column holds one fact, each row of a table holds one instance of a subject, and all rows in the table are distinct or unique instances of the subject. Data modeling applies the rule of one to our domain knowledge of a task, like shipping a package. Discovering domain of knowledge is hard and not a natural task for most people. You need one or more subject matter experts when you define subject domains, and at least one skilled business member on the team to assist you as you qualify the details of a business domain subject into an individual subject—that is, where the individual subject is a fact, or table definitions.

Subject-facts or table definitions are the positive outcome of data modeling. Sometimes subject-facts are subdomains of some business object, like negative, positive, and zero integers are a subset of all integers.

A subject-fact approach ensures that the table holds only one thing, which means it follows the rule of one. As you model to achieve the rule of one, you find that unique subjects fit into a table to hold like subjects. Tables like these contain rows with descriptive data that isn't duplicated in the table because of a natural key.

Organizing the subject-facts into tables is the first and certainly the hardest part of data modeling because as humans we sometimes have an imperfect knowledge of the subject domain. That's also true for business executives who have worked in an industry for years or even decades. This incomplete knowledge can lead to designs that may evolve over time, and data modeling practices should anticipate evolutionary understanding of the domain at hand.

Why Is Data Modeling Important?

Data modeling is the key to organizing data so that you avoid insertion, update, and deletion anomalies. Technically, anomalies like these are deviations from the *common* rules. There are three common rules for good design (check the glossary if these terms are new to you):

- Natural keys should always define unique rows in a table.
- Nonunique keys (like foreign keys) should always access a list of valid primary keys from another table or a copy of the same table (that would be a self-referencing join).
- Foreign keys should always reference the correct primary key.

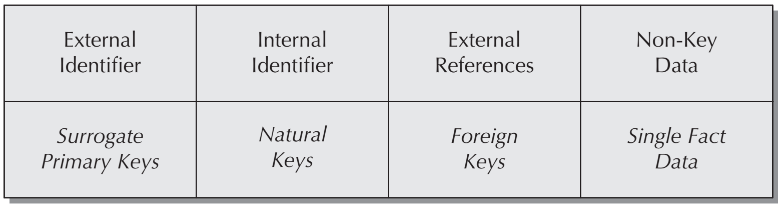
When you don't follow these common rules, insertion, update, or delete anomalies add, change, or remove data differently than expected. These deviations from the common rules corrupt data because they violate the rules that underpin how you should organize data.

Good data modeling prevents anomalies by ensuring that *unique* rows or object instances are always managed properly by data manipulation commands. This means data modeling guarantees the structure of any object type (or row) and adheres to the rule of one.

The rule of one ensures that every attribute contains either one thing or nothing, and every row ensures that the record structure (or collection of attributes) is unique among all rows in the table. The rule of one also ensures that every table defines a single subject or has qualified a subject-fact.

A subject-fact differs from a subject because it is an irreducible fact, like a datum—one fact. Let's examine the components of a subject-fact. It contains the following four subgroups (or more specifically subdomains) when modeled well, as shown in Figure 3-1:

- **External identifiers** These are typically surrogate keys, which are generated from auto-incrementing sequences. A surrogate key stands in for the natural key; and the natural key is a column or set of columns that uniquely describes each row in a table. Surrogate and natural keys are both candidate keys, which are candidates for selection as the primary key. You should always chose the surrogate key as a primary key provided the surrogate key maps to *one and only one natural key* value. A database sequence is a numbering schema that acts like a counter where the first row insert is 1 and the next is 2 and so forth. Surrogate also describes the fact that a sequence isn't descriptive of anything in a table. Surrogate keys should never determine uniqueness that the natural key doesn't. Any attempt to do so means you don't understand the table's domain and can't identify an internally unique identifier (or natural key) among the table's columns. Recognizing this fact, you should have two unique indexes. One should start with the surrogate key and include all natural key columns, and the other should include all the natural key columns. A good design should always elect the surrogate key as the primary key.
- **Internal identifiers** These are typically natural keys, which can be a single column or set of columns that naturally defines unique rows in the table. The idea is that a natural key holds column values that describe the row's data. It should be the values you look for when you want to update a single row in the table. The natural key should also become a unique index by itself, as covered in the external identifier bullet.
- **External references** These are foreign keys, which should reference to primary keys. The only problem with foreign keys typically occurs when they are valid possible foreign keys but incorrect references. This happens through application programming interface (API) errors, where a value is inserted that may reference the wrong primary key value. This type of error occurs when the development team is overly reliant on foreign key



External Identifier	Internal Identifier	External References	Non-Key Data
Surrogate Primary Keys	*Natural Keys*	*Foreign Keys*	*Single Fact Data*

FIGURE 3-1. *Data modeling subdomains*

constraints, which don't guarantee the right foreign key—only a possibly correct foreign key. Any value in a list of primary keys is a valid possible value for the foreign key because it meets the typical foreign key constraint rule, but it doesn't guarantee that a correct foreign key exists in any row. You must use the internal identifier to guarantee the correct foreign key in any row when you insert or update data.

- **Non-key data** These are descriptive columns that don't participate in uniquely identifying rows in the table and don't reference other rows in another table (or another row in the same table in a self-referencing relationship).

External identifiers are the surrogate keys or artificially unique numeric identifiers assigned to all rows. Internal identifiers should be the natural key or set of columns that uniquely qualifies an instance (or row) in the table. External references are foreign key columns, and they hold references that link one subject-fact table to another related subject-fact table. Non-key data are all the other columns that come along for the ride.

Primary, Surrogate, and Natural Keys

A *surrogate key* is an artificial numbering sequence that uniquely identifies every row in a table externally to other tables. A *natural key* is one or more columns in a table that uniquely identify every row, and it allows you to find any row inside a table. Natural keys are also called unique keys. It is a good practice to ensure that a surrogate key matches each unique or natural key. Failure to ensure that match means you invite *insertion, update, and delete anomalies* because the design fails to achieve a minimum of third normal form (3NF).

3NF is the third level of normalization. There are eight levels currently in the normalization process, which organizes data in relational models. 3NF requires two things: no transitive dependencies should exist between attributes—that is, no dependency exists between non-key columns; and the table must already be in second normal form (2NF). 2NF also requires two things: All non-key attributes must depend on all of the *natural keys,* which means a natural key defines uniqueness for all rows in a table. Also, the table should already be in first normal form (1NF). 1NF is the basic foundation layer of normalization and requires that all columns together uniquely define an instance of the table, or row, and that every column is atomic. Atomic columns have a single data type and no more than one value of that data type.

Both surrogate and natural keys become candidates for selection as the primary key because they guarantee unique row selection. You should choose the surrogate key as the primary key because its role is external identification,

(Continued)

which supports join operations to other tables. Then, you create an index composed of the surrogate and natural keys to help the database engine find rows in a table more quickly than a full table scan.

This practice is important for normalization, because occasionally design teams may evolve their understanding of the natural key. When you've based joins on a surrogate key and index, you can simply drop and re-create the index with the new knowledge. However, joins based on the natural key often require rewriting all SQL join statements between tables. The latter is simply too expensive and can be avoided by the use of surrogate keys.

The next subsections examine the roles of these subgroups and how they should work together to create a valid data model. After reading them, you should be in a position to avoid some of the most common pitfalls of design. You can think of the external identification as your published phone number, the internal identification as the listing information in a directory, the external references as your place of business or residence, and the non-key data descriptive entries somebody may or may not maintain in an address book.

External Identification

External identification is the easiest subgroup to discuss when the table uses a surrogate key. A surrogate key is a single column with a unique identifier; typically generated by a database sequence. The identifier is known as a sequence, and in MySQL, sequences are conveniently properties of tables. It's commonly called a surrogate key because it *stands in* for the natural key and contains nothing about the subject-fact in the table. In other words, sequence values don't describe anything about the table except the unique row number.

These columns are typically defined using an unsigned integer data type. Unsigned integers are recommended because they take less space and meet the identification need. The column's data type changes to an unsigned double as the data approaches the maximum range of unsigned integers.

Unsigned Numbers as Primary Keys

Unsigned integers or doubles are the best solution for positive numbers because you can use twice as many of them as signed integers, but that doesn't mean you can't use a signed integer or double to manage positive numbers.

It's important that you use the same data type for primary and foreign key values in MySQL. Failure to match unsigned primary key numbers with unsigned foreign key numbers raises an `ERROR 1005 (HY000)` when using an InnoDB database engine.

The external identifier should always enjoy a one-to-one relationship with the natural key. That means the surrogate and natural keys are qualified as candidate keys. A candidate key has only requirement—it must uniquely identify all rows in the table.

You choose the table's primary key from the list of candidate keys. Typically, there's only one natural key, but in some rare situations, two or more may exist. My recommendation is simple: Always pick the surrogate key because, over time, your knowledge of the domain may evolve and change the definition of the natural key!

I'd like to leave it at that, but here's the pro and con of the argument for using surrogate keys.

Pro Argument You should choose the surrogate key as the primary key because it becomes an immutable single point of reference to other tables that hold a foreign key value. This works, provided each surrogate key maps to a unique natural key and you create an index that prepends the surrogate key to the natural key. SQL joins between tables are unaffected by changes to the index when the natural key evolves (or adds a column).

Con Argument Surrogate keys are overhead because the natural key holds the values you need to find a unique row in the table. While natural keys evolve (add columns) over time, changes to the natural key should be made to all foreign keys and joining statements.

If you don't see the benefit of the pro argument, please note that it's almost a guarantee that any natural key changes over time. That's true because your knowledge of the table's domain increases over time and enables you to discover better ways to qualify the data. Clearly, the cost of changing SQL joins between tables disappears when you use surrogate keys as the primary and foreign keys. Surrogate keys abstract or hide the natural key from the join operation, which enables you to make alterations to the table structure without impacting joins.

Internal Identification

The internal identification columns are the natural key. The natural key is a column or set of columns (also known as attributes) that uniquely describe instances (or rows) of the table. The natural key columns contain business information that allows programmers and business users to identify instances of data in a table.

When you select, update, or delete a unique row in a table, you use the natural key columns in the `WHERE` clause to isolate a row instance (or record). The natural key should always map to one surrogate key value, and that guarantees that the surrogate key identifies a naturally unique row.

Over time, it is possible that your understanding of a table's subject, or domain, may evolve. Let's take a look at how you model a table. First you choose a subject, like an application user's information. Having chosen the subject, we qualify all the attributes that belong in the subject and then we pick attributes that uniquely identify each row. The collection of attributes defines a natural key.

Somebody in the design group might suggest first and last name as the composite natural key before somebody interjects that there may be two people with the same first and last name. Adding a middle name doesn't solve the problem because two individuals could have all three names in common. While this type of discussion typically happens in early design and by beginners, it illustrates that what we think may be a unique key isn't always a complete key. Adding a middle name to the first and last names evolves the natural key by adding an attribute to it, but it stills fails to guarantee uniqueness. A more involved iteration of this problem occurs when you globalize names, because various countries follow different naming conventions, patterns, and character sets.

NOTE
The best solution in the application user case requires creating a user-defined user name attribute. This user name can be constrained as unique in the table, which means only one person can ever use it. It becomes a unique, natural, and candidate key all at once.

As your understanding evolves, the set of columns that defines the natural key changes and defines a new candidate key. If you opted to use the natural key as the primary key, all foreign keys must change to reflect the new primary key column set. On the other hand, you need drop and re-create only the table's unique index when you choose the surrogate key as the primary key.

The only problem with evolving natural keys occurs when the set of columns isn't unique with an existing data set. This type of change may mean that the natural key's set of columns must grow by including one or more columns to qualify uniqueness. On occasion, it also may mean that you need to change a column data type to qualify uniqueness. The most frequent change is adding columns to the natural key's set of columns. Sometimes you can simply change a column's data type to increase precision, like changing a `DATE` to a date-time data type, like `DATETIME` or `TIMESTAMP`. This type of change to a natural key column works when no two records may occur within the same date-time but can occur on the same day. The change from a discrete date to a continuous date-time data type shouldn't require a change to existing joins when a natural key is chosen as the primary key, provided you make the same change to any referencing foreign key columns. The change will also require rebuilding the index on the natural key whether or not you use a surrogate key as the primary key.

External References

The external references section can be empty in some tables, but that's usually not the case. That's because there are few tables that are independent of other facts. Most tables have external reference columns, known as foreign key columns, and they point to the primary key of other tables. Some foreign keys point to the primary key found in the same table, and that behavior makes their relationship recursive or self-referencing.

Foreign key columns hold copies of a value from the primary key column. A join between the foreign and primary key columns lets you link the facts from two tables into one result set. Joins like this are made on the basis of equality between the values of the shared column and are called equijoins (a formal word for equality joins).

Many rows in a table may share the same value in a foreign key column or set of columns (for example, when the primary key uses a natural key). Natural and surrogate keys are unique, but foreign key columns frequently hold values that repeat across many rows. The fact that foreign keys hold repeating values helps identify that their purpose isn't to describe the subject of the table. Therefore, the purpose of foreign key columns is to associate the values in one row to another row in that table or a different table.

Non-key Data Non-key data is the easiest to create and the hardest to describe for two reasons. One is that non-key columns hold descriptive values that aren't part of the unique natural key, and the other is that they don't point to other columns inside or outside of a table.

There are three classifications for non-key columns. One class describes something beyond the subject-fact of the table that doesn't merit being placed in a table of its own. An example of this classification type would be a product type column that holds a product's type, a product column that holds a product's code, a product label column that holds a product's language-friendly label, and a product definition that describes the product. Although, you could define a table for these four columns and populate the table with a finite set of rows, this type of information fits a common information pattern like a VIN (vehicle identification number). VIN or manufacturing serial number columns contain keys to unlocking product information typically stored in another database model. You shouldn't actually model your product information into tables because it creates the risk of insertion and deletion anomalies.

Another common information pattern, the *common lookup table*, lets you group several small tables into a large lookup table by recognizing that they follow a generalized pattern. This type of generalized pattern is a common reference or lookup table of product type, code, language-friendly label, and definition (or meaning). I'd strongly recommend this information pattern when you need to describe product or item descriptions and leave an external reference (or foreign key) in its place.

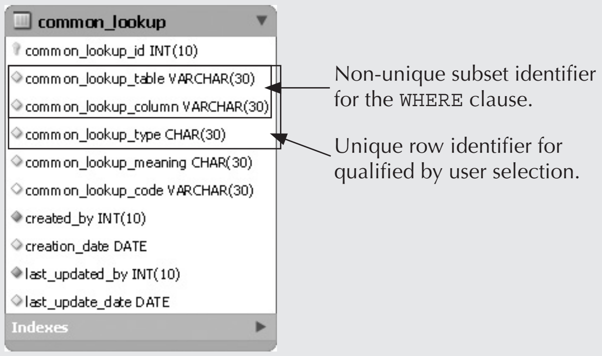
A second class of non-key columns would be descriptive columns. For example, the table may contain gender and gender description columns. The two columns would hold, respectively, an abbreviation, like "M" or "F," and description of gender, like "Male" or "Female." They would qualify the same attribute of the fact-subject table, and when gender is required as part of the primary key, you would most likely choose the abbreviation column over the description column.

Another class of non-key columns describes details of the subject-fact table that aren't part of the primary key because presently they're unnecessary to qualify unique rows in the table. These non-key columns may become part of the natural key as your understanding of the domain increases, but in some cases, they're simply related content that doesn't merit its own table. These types of columns are often optional values when inserting rows; and those columns that belong in the natural key evolve into it as your domain understanding increases.

A Common Lookup Table

A common lookup table contains a collection of small tables that describe common values, like gender and other frequently used values. A common lookup table is a generalization of little tables, and you create it by using a three-column natural key. The natural key columns are table name, column name, and a column type value.

The set of possible table and column name column values would qualify rows within a specialization or small logical table. The combination of table, column, and type column values should uniquely identify every row in the design, like that in the next illustration.



```
common_lookup
common_lookup_id INT(10)
common_lookup_table VARCHAR(30)
common_lookup_column VARCHAR(30)
common_lookup_type CHAR(30)
common_lookup_meaning CHAR(30)
common_lookup_code VARCHAR(30)
created_by INT(10)
creation_date DATE
last_updated_by INT(10)
last_update_date DATE
Indexes
```

Non-unique subset identifier for the WHERE clause.
Unique row identifier for qualified by user selection.

In addition, you generally have non-key columns for the lookup code, language-friendly label, and description columns. These non-key columns are the ones displayed while your programs use the key columns to gather the rows for display.

This section has explained the four subdomains of external identification, internal identification, external reference, and non-key data. It also examined how they interact and work. The next section discusses how you guarantee good data modeling design by applying the rule of one.

How Do We Accomplish It?

This section discusses how you can design an effective database. There are three phases to effective design. They are creating an Entity Relationship Diagram (ERD) or Entity Relationship Model (ER Model), creating a business interaction model, and validating the business interaction and ER models are mutually supportive.

The terms ER Model and ERD are really synonymous terms, and their use is interchangeable in this chapter. MySQL Workbench uses Enhanced Entity Relationship Model (EER Model), and the book uses that term when describing product functionality. That means you have three acronyms that you should associate in your mind to make reading easier—ERD, ER Model, and EER Model.

Create an ER Model

Creating an ER Model is essential to designing how a database supports a business process. You need to create an ER Model as the first step in working with business users.

The key element of a good design for an ER Model (or ERD) requires that you have a team of individuals who have a deep, or profound, understanding of the business process and the data that supports it. You also need to have a data-modeling person who has a solid understanding of how to design normalized tables and relationships between tables. Together, you create a map of tables and their relationships, which is known as an ER Model.

NOTE
Entity is another word for table, as is object type. MySQL Workbench takes the less formal approach and simply calls them tables.

Create a Business Interaction Model

After completing the model, you should invite developers into the forum and create mockups of the user-interface (UI) forms and reports. These mockup sessions are more or less what is sometimes called a *blue sky* exercise. *Blue sky* indicates that there are no limits technologically or financially to developing the envisioned solution. The blue sky design, or first pass through designing mockups, should not attempt to verify that the fields from the UI and reports correlate to the ERD Model; rather, it should be considered a brainstorming session. A brainstorming session records ideas without criticism that may discourage recording the information.

Business interaction models like these help engage the business user in qualifying what may subsequently become the Unified Modeling Language (UML) use cases. It is always more engaging for business-centric people to work with the UI and report mockups over UML diagrams, and the UML can be derived from the UI and reports later by a technical-only staff during the unified process (UP) or Scrum Agile Development project plan. (Both of these are Agile development methodologies associated with iterative System Development Life-Cycle (SDLC) processes.

Validate the ERD Supports the Business Interaction Model

Validating the ERD isn't very difficult when you recognize that you can walk through the data by using the business interaction model's UI and reports. You simply walk through the logic from data entry to support for Management Information System (MIS) reports.

This step is where you have to ensure that your team has a stake holder, someone with profit and loss (P&L) responsibility for the business process. The stake holder is also called the product owner in a Scrum project management method. You'll also need to have a Scrum Master who oversees the process.

Agile Manifesto Rules

There are four key principles involved in Agile Software Development, and they prioritize:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The principles state what should be first in four areas but they don't say disregard process and tools, documentation, contract negotiation, or following a plan. How an organization chooses to prioritize and balance these areas indicates the SDLC maturity. A healthy balance between these ensures more frequent success than failure, and a fully documented and budgeted solution at product release.

Together the teams validate whether their models are mutually supportive by qualifying test cases from a business perspective. The Scrum Master should have these translated into UML activity diagrams, which support how the quality and assurance (QA) teams build test cases. However, it should be noted that UML diagrams are not a hallmark of success and may be avoided by project teams because they create communication hurdles between the business users and data modelers. That's a choice of working software over complete communication from the Agile Manifesto (see the "Agile Manifesto Rules" sidebar).

During the development of the business interaction model, the real domain analysis occurs. That domain analysis decides what tables should exist, how columns should define tables, and what relationships exist between tables. You'll find that the natural keys of tables evolve here, and hopefully they will support the real-world application. The more accurate the natural key at this level of the project, the fewer changes required later in the project. Although a natural key can contain all columns in a table at the 1NF level, good natural keys typically hold between three and eight columns. Natural keys with more than eight columns typically indicate that you have more than a single subject in the table.

It would be terrific to say that this process is smooth and always consistent, but it isn't, and this validation process is often an iterative process until enough is understood to build a small prototype of the application. Initial prototypes tend to evolve throughout each cycle of an iterative process, especially in an Agile software development process.

This completes the basics of data modeling, and positions you with a foundation of what it is, why it's important, and how to accomplish it. The next section shows you how to create the ERDs using traditional information engineering and UML.

Data Modeling Systems

Data modeling system sets are numerous. Here, you'll only cover the basics of Chen's modeling system, the modified Chen-Martin modeling system, Martin's Information Engineering, and UML notation. Reviewing these three systems, however, will help illustrate the basic concepts of data modeling.

NOTE
MySQL Workbench uses Information Engineering symbols when rendering EER Models.

These modeling symbol sets let you examine the table, its columns, and the relationship between tables in your database model. A relationship can be logical or physical and can exist between two tables or more than two tables. Relationships between two tables are binary relationships; relationships between three tables can be called ternary, but they are usually treated as n-ary relationships. An n-ary relationship exists between *n* number of tables, which traditionally is three or more tables.

There are three possible *binary* relationships between two tables: one-to-one, one-to-many, and many-to-many. This can be confusing because the *one* and the *many* have nothing to do with the two tables. They describe the number of rows involved in a relationship between two tables, and you could more aptly say binary relationships describe how many rows in one table share values that match values in the rows of another table. Binary relationships can also describe how many rows in one copy of a table share values that match rows in a copy of the same table. The binary relationship between rows of the same table is self-referencing because one copy matches against another copy of the same table.

One-to-One It means one row in a table relates to one row in another table through a relationship in its simplest form. In the case of a self-referencing relationship, it means one row in a copy of a table relates to one row in another copy of the same table through a relationship. The relationship may be between two copies of the same row, or one row and another row or set of rows, in a self-referencing table, and the relationship is between a primary and foreign key.

One-to-Many It means one row in a table relates to many rows in another table through a relationship. The one side holds the primary key (which is always unique) and the many side holds foreign keys that map to primary key values. Self-referencing or recursive relationships may also exist in one-to-many binary relations, and exist when one copy of a table holds the primary key and the other side holds copies of rows with foreign key values that match the primary key value.

Many-to-Many It means many rows in a table relate to many rows in another table through a relationship, but it's not a direct relationship like primary-to-foreign key relationships. That's why many-to-many binary relationships are *logical*, not physical, realities. Many-to-many relationships are always logical data models, and that means you must convert them to physical models before implementing them. MySQL Workbench is designed to support physical not logical models.

A physical implementation of a many-to-many relationship would require a primary key column with a scalar value and a foreign key column with a list of values to exist in both tables. The list of values enables mapping between the two tables' primary key values. For example, you start in the scalar column that holds primary keys, then map through a set of foreign keys in the list column to the primary key scalar column in the other table, and vice-versa. The problem is that introducing lists makes it a nonrelational solution to the problem.

It's nonrelational for two reasons. First, relational models don't contain lists, arrays, or serialized values because they're not atomic values. Atomic values means that they hold one thing, like a scalar variable. Second, relational models hold one column value or a null in each row of a table, and a scalar column with a list is unbalanced.

The solution requires you to decompose (break into smaller pieces) many-to-many binary relationships into two one-to-many relationships. That requires you to create an intermediary table called an association table. It holds a foreign key from each table that allows you to connect many rows from one table with many rows from another table.

Like many-to-many binary relationships, n-ary relationships follow the same pattern, and require you to break them down into smaller parts. You break the logical n-ary model into two or more sets of one-to-many or one-to-one relationships. For example, an n-ary model of three tables would have an association table that holds three foreign key columns. That means there's one foreign key column in the association for each subject table in the logical n-ary model. This creates three physical one-to-one or one-to-many binary relationships where the one side is always the subject-fact table and the many side is always the association table (which infrequently may also be the one side).

This means ER Model implementations require physical data models, and all physical data models only work with binary relationships. MySQL Workbench is a great tool to use when creating physical ER Models.

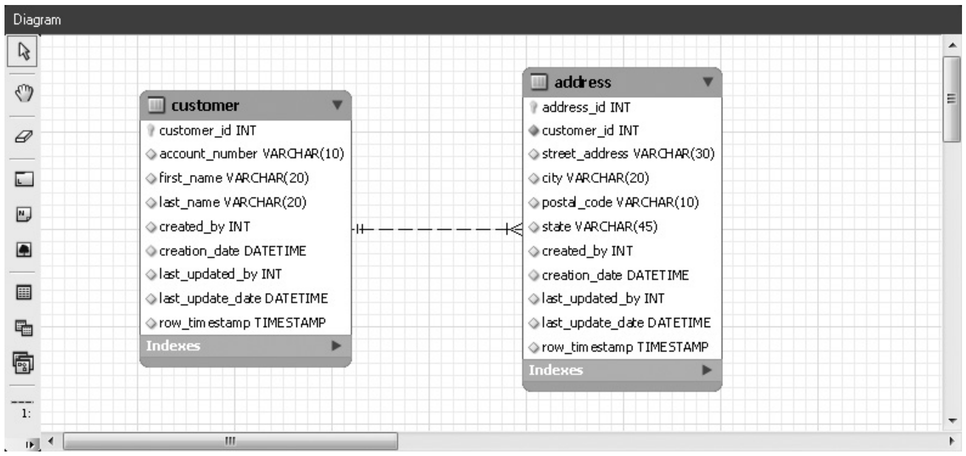
Binary Relationships

One-to-one relationships are rare, but may exist. A one-to-one binary relationship between tables identifies one table as unconstrained and the other as constrained in a relationship. The unconstrained table holds a primary key that uniquely identifies each row, and the constrained table holds a primary key of its own and a foreign key that holds a copy of the unconstrained table's primary key.

You pick the unconstrained and constrained tables during your ERD design. You control which table fills which role because there's only one copy of the primary key held as foreign key in a one-to-one relationship. This means you could switch the roles by changing which table holds the foreign key. SQL joins the two related rows from different tables by comparing the values in the primary and foreign key columns.

The *flexibility* of one-to-one relationships is a two-edged sword because one-to-one relationships frequently evolve into one-to-many relationships. In one-to-many relationships, the foreign key is always held by the many side and the primary key by the one side of the relationship. This means you should ask yourself which table is the driving or key table in the business relationship because that table is less likely to evolve into the many side of the relationship. If you chose the wrong driving table during design, fixing the mistake can be expensive, and progressively more expensive to fix as the process continues.

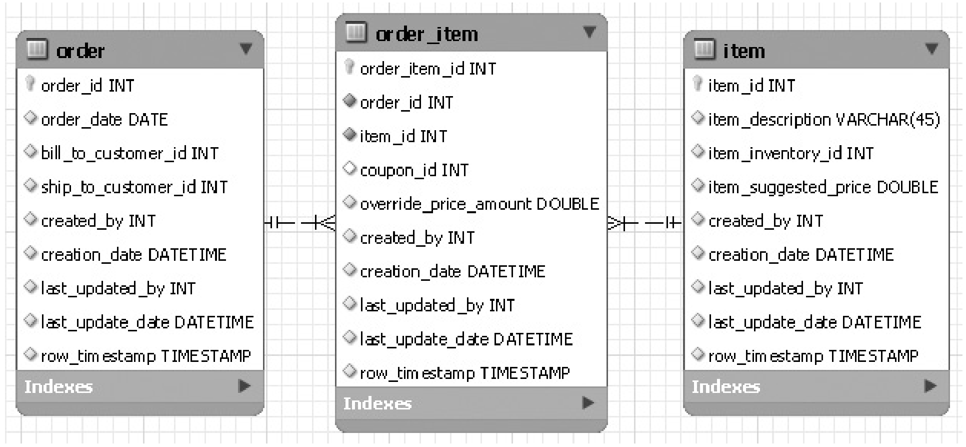
Assume you're new at data modeling and you want to map a customer to an address. You have two options. One assumes the customer has only one address, and the other assumes the customer may have two or more addresses. You choose a one-to-one relationship when the business model states you only care about the customer's current home address and a one-to-many relationship when the business model states you want the current and past addresses of the customer. In both cases, you assign the primary key of the `CUSTOMER` table as a foreign key in the `ADDRESS` table because it works for either model because a SQL statement can join on matches between the primary and foreign key columns.



As discussed, you implement logical many-to-many relationships as two one-to-many relationships. Take, for example, the natural relationship between orders and items in a business model. An order may contain any item and more than one item per order, while an item may be part of one or more orders. This is a logical many-to-many scenario with relationships between many orders to many items.

The problem with a many-to-many relationship is that the table holding orders would need a foreign key from the table holding items for every item in an order, and likewise the items table would need to hold a foreign key from the orders table for every table that ordered the item. As discussed earlier, that's the physical reality of a primary key scalar data type column and a list column with multiple foreign key values. It isn't a relational model possibility. That's why an association table holds copies of the primary keys from both tables.

For example, you create an `ORDER_ITEM` association table between an `ITEM` table and `ORDER` table. The `ITEM` table donates a copy of its primary key to the `ORDER_ITEM` table for every `ORDER` that includes it, and the `ORDER` table donates a copy of its primary key to the `ORDER_ITEM` table for every `ITEM` on the order. This lets you create a customer order by associating any number of items with any number of orders because the association table holds the matching foreign key values. It holds one foreign key for the `ORDER` table and another for the `ITEM` table. A surrogate primary key may be added to the `ORDER_ITEM` table because it may hold shared attributes that describe neither the subject-fact of the `ORDER` nor `ITEM` tables. The composite natural key is the set of foreign key columns, and they should be uniquely constrained in any good design.



That was simple, but what happens when each order item holds both a suggested retail and actual price value for each item? You would typically capture the actual price from a `PRICE` table, but the actual price may be provided by a clerk through an override activity. That means the actual price belongs to the order for the item but doesn't belong to any subject-fact table. This is what's known as a shared attribute.

Shared attributes only have context in the midst of a relationship between two subject-fact tables.

Since clerks may assign actual prices that have no relation to anything in the data model, such prices would be raw entry at time of sale. That's why the foregoing model shows the `COUPON_ID` column as optional (null allowed or nullable) because a simple override price doesn't require a coupon. Alternatively, customers may present discount coupons that discount suggested-retail prices. The latter requires that you store the percentage and type of coupon presented. The percentage and coupon type become shared attributes in the `ORDER_ITEM` association table. While I'd love to leave it at that, there's one more twist that frequently occurs. Generally, the coupon type is set in a `COUPON` table or a `COMMON_LOOKUP` table, and the shared attribute is a foreign key to only one of those tables. My suggestion is that you should put the coupon information in a `COMMON_LOOKUP` table when you often use the same type of discount coupons and they change infrequently. If you often use coupons and they change frequently, put them in a `COUPON` table with temporal columns like `START_DATE` and `END_DATE`. This makes `COUPON` a coupon-only lookup table.

TIP
Never design a column that may hold a foreign key that belongs to one of two or more tables because it invariably leads to confusion and usage errors.

As discussed, there are different types of business rules that require shared attributes in association tables; and don't shy away from using association tables that may hold more shared attributes. The presence of shared attributes in an association table changes the natural key from the set of foreign keys to the set of foreign keys and all (one or more) of the shared attributes. That's something not immediately visible to new designers. The business reason for including the shared attributes in the natural key exists in a coupon scenario or a two-for-one sale where one item carries the discount and another doesn't.

An association table that supports coupons or two-for-one sales presents a challenge. It introduces the need for a `QUANTITY` column to support a two-for-one sale. Naturally, the `QUANTITY` column lets us track how many of a specific item are linked to an order and it is always a mandatory column (containing an integer of 1 to any number of items) during an insert or update. Unfortunately, the `COUPON_ID` column is typically optional because the customer may or may not present a coupon.

There's a very subtle design trick here because some databases don't support unique constraints unless all columns are also not null constrained. It requires you to insert a full-price indicator in a coupon type shared attribute. I'd strongly recommend you always do this to ensure possible portability of ER Model designs. The two-for-one, three-for-one, or other ratio relationship should be stored in the `COUPON` or `COMMON_LOOKUP` table. The presence of a ratio relation column is a strong argument for a coupon-only lookup table.

While association tables typically don't require surrogate keys because the two foreign keys provide a unique natural key, they should always have a surrogate key when you have shared attributes in them. Shared attributes by themselves appear to be along for the ride in the association between the two tables, but sometimes shared attributes become transactional events—for example, when you need to generate an RMA for a customer who returns a single item from an order where two or more items were required for a discount. The value in the `QUANTITY` column of the `ORDER_ITEM` table and the ratio in the `COUPON` table (typically implemented as `MINIMUM` and `MAXIMUM` columns) let you resolve most returns but not all.

There are two patterns for storing shared attributes in an `ORDER_ITEM` table. One uses a quantity counter and the other creates two or more rows in the association table. The quantity counter is the easiest and preferred solution but doesn't work when the item sold has a serial number associated with it. You can resolve serial numbers by adding a `SERIAL_NUMBER` column to the association table while entering a value of 1 in the `QUANTITY` column. Although, inserting a serial number in the association table isn't a great idea because it probably duplicates the serial number from your `INVENTORY` table.

In lieu of adding a `SERIAL_NUMBER` column, you should define an `INVENTORY_ID` column that holds a foreign key to the `INVENTORY` table. That way, you store the serial number in only one place in the database and eliminate the possibility of incongruent serial numbers in the two tables (that is, different values for the same serial number). A serial number fix like this promotes the association table solution from a two-table to three-table solution.

The RMA process would first check if two or more rows are involved in the purchase order for the same item and if they have serial numbers. The `SERIAL_NUMBER` column is typically an optional column in the `INVENTORY` table because not all merchandise has serial numbers. The RMA process would then authorize an RMA for the discounted price rather than the nondiscounted price and, in some cases, an average price.

Creating a surrogate key for all association tables is a best design practice because those without shared attributes may evolve to include them and those with shared attributes need them. Adding a surrogate key column to an association table is a low-cost design and development practice that may avoid higher modification costs during future application maintenance cycles.

TIP
Always define a surrogate key for association tables because they may hold shared attributes and become data instead of relationship maps.

As shown in the serial number example, association tables between two tables become association tables between three or more tables. This occurs naturally when the foreign key of another table is stored as a shared attribute. A change like this promotes the binary association table to an n-ary association table, and they're covered in the next section.

N-ary Relationships

N-ary relationships are logical relationships that exist between more than two tables. They're resolved into a series of binary relationships like the many-to-many relationships. The key difference between the two is the number of foreign key columns in the natural key. A many-to-many binary relationship always has two foreign keys as a composite primary key. An n-ary relationship has *n* (the number of tables in the relationship) foreign keys in the composite primary key.

You should always define surrogate keys for association tables, and that rule becomes more important with n-ary association tables because they're more likely to include shared attributes. The problem with shared attributes in n-ary tables is more complex than in binary association tables. N-ary relationships are rare, and you should ensure that other simpler modeling solutions are carefully examined before adopting the complexity of n-ary relationships.

Notation Sets

There are several notation sets that represent relationships and tables. Since MySQL Workbench uses the Information Engineering model, the discussion focuses on the notation sets that led to it as the predominant notation set for ERDs. After describing the ones that help you use MySQL Workbench, there's a comparison of using UML for ERDs.

The notation sets explain how symbols enable you to read the cardinality of relationships between tables. Cardinality measures the minimum and maximum number of rows one table will be related to in another table. Check the "Cardinality" sidebar for more on the subject, which applies to columns, relationships, and the distribution of uniqueness in tables.

The book exposes you briefly to the main styles of ERD notation to ensure you can read older diagrams as you convert them into the EER Model of MySQL Workbench. The book covers Chen, Chen-Martin, and information engineering symbol sets. It excludes many others, like the DoD (Department of Defense) IDEF1X (Integration Definition for Information Modeling).

Cardinality

Cardinality comes from set mathematics and simply means the number of elements in a set. For example, in an arbitrary set of five finite values, a cardinality of [1..5] qualifies the minimum of 1 and the maximum of 5. This set expresses a range of five values.

(Continued)

In databases, cardinality applies to the following:

- The number of values in an unconstrained column within a row has a default cardinality of [0..1] (zero-to-one) for nullable columns. (The minimum cardinality of zero applies only to nullable columns, which are also known as optional column values.)
- The number of values in a `NOT NULL` constrained column within a row has a cardinality of [1..1] (one-to-one).

When there's no upward limit on the number of values in a column, it holds a *collection*. Collections typically contain one-to-many elements and their cardinality is [0..*] (zero-to-many). This type of zero-to-many cardinality can't exist in a relation model, but does exist in Object Relational Database Management Systems (ORDBMS), like Oracle and PostgreSQL.

Developers often describe the frequency of repeating values in a table as having low or high cardinality. *High cardinality* means the frequency of repeating values is closer to unique, where unique is the highest cardinality. *Low cardinality* means values repeat many times in a table, such as a gender column where the distribution is often close to half and half. A column that always contains the same value, which shouldn't occur, is the lowest cardinality possible.

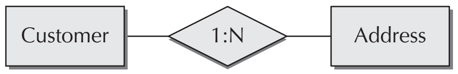
Cardinality also applies to binary relationships between tables. Two principal physical implementations of binary relationships exist: one-to-one and one-to-many. The one-to-many relationship is the most common pattern. In this pattern, the table on the one side of the relationship holds a primary key and the table on the many side holds a foreign key.

Chen Notation

Chen notation has more value as a historical item than as a practical tool. It uses rectangles as the symbol for tables and ovals as symbols for columns (or attributes). A line between rectangles qualifies a relationship, and a diamond in the line indicates the type of binary relationship. The symbol sets in the diamond are 1:1 for a one-to-one relationship, 1:N for a one-to-many relationship, and N:M for a many-to-many relationship. These binary relations qualify the maximum number of rows in a relationship.

The Chen model didn't last long with the relationship cardinality inside the diamond. It removed the binary notation from the diamonds and replaced them with a keyword describing the relationship. Then, the number 1 or letter *n* was placed on the appropriate sides of the relationship description diamond.

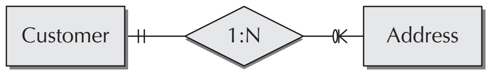
The original Chen notation doesn't support the minimum cardinality of relationships, and that omission is why it's typically not used. The replacement Chen-Martin method does provide for both minimum and maximum relationship cardinality. The next illustration actually depicts a one-to-many cardinality with the *N* inside the diamond qualifying many and, at the same time, maximum cardinality.



While the foregoing illustration is a summary diagram, you would see a list of column names with the natural key columns underlined in a detail diagram. The modeling technique does not enable identifying the candidate keys—the natural and surrogate keys.

Chen-Martin Notation

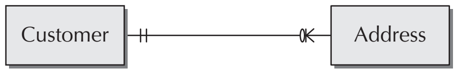
As mentioned, Chen-Martin notation added minimum relationship cardinality to the drawing set. It did this by adding a single perpendicular line for a minimum cardinality of one and a zero (or oval) for a minimum cardinality of zero.



The change makes Chen-Martin a viable modeling symbol set, but the relationship description diamonds clutter diagrams. They make it hard to create large ERDs that conveniently fit on printed documents, even large poster boards. The Martin method, or information engineering, removes these diamonds and introduces a more streamlined notation symbol set.

Information Engineering Notation

Martin's method, which is also known as information engineering, drops the diamonds for what has become known as crow's-foot notation. Information engineering is probably the most widely used method for ERDs because many tools support it and it allows you to fit a lot on a single page.

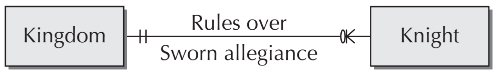


The symbol set for relationships are

- One oval (or zero) and a perpendicular line across the relationship line indicate a minimum cardinality of zero and a maximum cardinality of one. This is the default because foreign key columns are typically unconstrained or null allowed columns. This means you may insert a row in the table holding the foreign key without providing a value to the foreign key column; and the table with the foreign key is unconstrained in the relationship to the table holding the primary key.

- Two perpendicular lines across the relationship line indicate a minimum cardinality of one and a maximum cardinality of one. This would be the relationship when the constrained (or dependent) table in a relationship can't insert a row until one exists in the unconstrained table with a primary key value. This relationship exists whenever the foreign key column is constrained as a `NOT NULL` column.
- One oval and a less than (<) symbol read left to right or one oval and a greater than (>) symbol read right to left indicate a minimum cardinality of zero and a maximum cardinality of many. Like the zero-to-one relationship, the foreign key would be nullable or optional when inserting a row in the table.
- One perpendicular line and a less than (<) symbol read left to right or one perpendicular line and a greater than (>) symbol read right to left indicate a minimum cardinality of zero and a maximum cardinality of many. Like the zero-to-one relationship, this relationship would make the foreign key value mandatory or required when inserting a row in the table.

Information engineering also moves the relationship qualifier outside of the diamond and next to the relationship line. While Chen and Chen-Martin let you qualify only one relation (half of a relationship), Martin's method lets you annotate the relationship line with verb phrases that let you read two relations in any relationship like English sentences (or whatever language you're using for ERDs).



For example, you could read the relationship from `KINGDOM` to `KNIGHT` two ways because the minimum cardinality is zero. The first one qualifies that there may be *zero to many,* and the second uses *may* to qualify that the minimum cardinality is zero.

Any `KINGDOM` rules over zero to many rows of `KNIGHT`.

or, with the conditional *may* like this:

Any `KINGDOM` may rule over one to many rows of `KNIGHT`.

You can read the relationship back from the constrained `KNIGHT` table (assuming the foreign key is not null constrained) to the unconstrained `KINGDOM` table like this:

Any KNIGHT has sworn allegiance to one row of KINGDOM.

Information engineering covers all relationships patterns except one with rectangles. An ID-dependent relationship requires that you use a rectangle with rounded corners and an ordinary relationship line. It indicates that the table has no existence outside of the unconstrained table, and the unconstrained table holds the primary key. You can't implement an ID-dependent relationship easily in MySQL or relational databases. ID-dependent tables are typically implemented as nested tables—available in Oracle and PostgreSQL object relational databases.

UML Notation

UML (Unified Modeling Language) is an object-oriented analysis and design (OOAD) approach. There are only two differences worth noting.

One difference is that the rectangles have three subordinate rectangles. The top one is for the table name, the middle one is for the attributes of the table, and the bottom one is for any methods of the object (or table). The bottom one is the give-away that this doesn't work in relational databases but requires an object relational database like Oracle or PostgreSQL.

The other difference is that there are different symbols for the relationship line. One qualifies a non-ID–dependent relationship, which is the typical primary-to-foreign key relationship, and that symbol is an aggregation relationship line. The other qualifies the ID-dependent relationship, and it is the composition relationship line. An open (or unfilled) diamond represents aggregation, and a filled diamond represents composition.

This completes the data modeling system section.

Classic Normalization

Database normalization is the process of organizing data, as briefly discussed earlier in this chapter's "Primary, Surrogate, and Natural Keys" sidebar. There are a bunch of rules governing how you should do it, when you should undo it, and how you can't do it. My hope is to lay out what normalization means in Texas English, which means clear and simple.

There are now, as of 2010, seven or eight normal forms. They began when E.F. Codd first proposed the relational model in his *A Relational Model for Large Shared Data Banks* paper. There's also a concept of Domain-Key Normal Form (DKNF). According to some, this belongs between fifth and sixth normal forms. DKNF comes to us by way of Ronald Fagin, in his *A Normal Form for Relational Databases*.

Database normalization attempts to organize data in such a way as to prevent SQL statements from creating insertion, update, or deletion anomalies. As a practice *third normal* form (3NF) is often considered normalized because most 3NF tables are free of insertion, update, or delete anomalies. The key word is most, not all.

Therefore, normalization design attempts to achieve the *highest normal* form (HNF) possible. A table is in HNF whether it meets or fails to meet any normal form definition. Oddly enough, any HNF may also be a *zero normal* (0NF) or *unnormalized normal* form (UNF). UNF means that a table contains one or more repeating groups.

Normalization is the process of organizing data into tables that act as single subject-facts when acted upon individually or through external relationships. As mentioned earlier in this chapter, a single subject-fact is also known as a domain.

First Normal Form

First normal form (1NF) exists when all columns, or attributes, of a table have a single data type and there are no repeating rows of data, which means rows are unique. 1NF requires that

- Column data types be atomic, which means columns shouldn't have repeating groups, comma-delimited groups, or other subatomic parts. This raises a question whether compound variables like Oracle's arrays, lists, and objects violate first normal form. Oracle's compound array and list columns do not violate the atomic design rule because they act like nested tables or ID-dependent tables—accessible only through the external table.
- Column names are unique in tables and arbitrarily ordered, which means their order doesn't impose any constraint on the table.
- Rows in a table have no implicit or explicit ordering required for their access and use.
- The collection of columns should define unique rows.

Moreover, first normal form modeling finds the nonrepeating row columns and moves them from a base table to their own table. It removes all trace of the nonrepeating row columns from the base table. It also puts a foreign key in the new table, and the foreign key points back to the primary key of the base table. The foreign key in the new table is functionally dependent on the primary key in the base table.

Atomic Data Types

Atomic column values limited to scalar variables. Scalar variables only hold one thing at a time, like an integer. How I'd love to leave it at that!

Unfortunately, some MySQL designers create tables that store serialized strings in a single column, which introduces the concept of lists in a relational model. The only place lists belong is in an Oracle's database engine because it is an Object Relational Database Management System (ORDBMS) or Extended Relational Database Management System.

Likewise, some developers put numeric values separated by a delimiter that contain meaning; typically, the delimiter is a colon. This pattern is typical in the case of tickets, where the first, second, and third numbers map to the section, row, and seat. This force fitting of logic into a single column makes such a column nonatomic.

You should avoid putting more than one thing in any column. Likewise, you should try to use descriptive column names.

A table is in first normal form when all columns (or attributes) have a single data type and when there are no repeating rows or set of columns. The first two elements of this idea are generally understood and applied almost intuitively. The idea of a single data type is clear because most people understand any column can only have one data type at any given time. The idea of uniqueness makes sense because you generally only want to act on one copy of anything at a time.

The third point is sometimes misunderstood. The idea of not being able to have repeating group of columns can confuse people because they often look at tables as a group of unique rows. Repeating groups of columns indicates that you've got multiple subjects in your single table design. The repeating row columns indicate the base table is on the one side of a one-to-many relationship. The columns of nonrepeating rows typically become a new table because they are on the many side of a one-to-many binary relationship.

The subsections present an example and two possible solutions. A third solution exists for object-oriented relational databases but isn't provided because the MySQL database doesn't support nested tables.

UNF-to-1NF Problem

The example in this section uses a sample address table to show you how to move a table from UNF, or an HNF of 0NF, to 1NF. It has a design that lets you violate first normal form whenever the `STREET_ADDRESS` column requires more than a single entry or more than one row.

This type of table design lets you violate the atomic rule of first normal form because the `STREET_ADDRESS` column may need to accommodate one or more street address values. Typically, many developers see no harm in putting multiple address lines into a column as comma-separated values. The data would look like the following:

```
+-----------------------------------+---------+-------+-------------+
| street_address                    | city    | state | postal_code |
+-----------------------------------+---------+-------+-------------+
| 1111 Broadway, Suite 500, MS-5045 | Oakland | CA    | 94604       |
+-----------------------------------+---------+-------+-------------+
```

Another type of table design lets you violate two rules of first normal form. It modifies the previous example by adding a second column to the primary key. The second column doesn't describe the subject-fact of the table, but introduces a way to order row results. Such a change makes the primary key a compound or composite key. The following provides an example of one record spread across three rows of such a table.

```
+------------+-------------+---------+----------------+---------+-------+
| address_id | customer_id | line_id | street_address | city    | state |
+------------+-------------+---------+----------------+---------+-------+
|          1 |           1 |       1 | 1111 Broadway  | Oakland | CA    |
|          2 |           1 |       2 | Suite 500      | Oakland | CA    |
|          3 |           1 |       3 | MS-5045        | Oakland | CA    |
+------------+-------------+---------+----------------+---------+-------+
```

The CUSTOMER_ID column links the row to the CUSTOMER table, and the LINE_ID column allows for multiple rows because the LINE_ID column lets you order rows. The CUSTOMER_ID and LINE_ID columns are a composite natural key in this scenario, notwithstanding that the CUSTOMER_ID is a foreign key (external reference) to the CUSTOMER table. Such a design creates a model where you have repeating column values, like CITY and STATE for each unique STREET_ADDRESS column value. This type of design violates the 1NF rule of no repeating group of columns, and the stored data mimics a denormalized result set that should be returned from a join between two tables.

You would create ADDRESS and STREET_ADDRESS tables to model this problem correctly. The ADDRESS table would hold one value of city and state for each unique address linked to a customer, and the STREET_ADDRESS table would hold the three rows related to the ADDRESS table's one row. The two tables would allow for unique results in a single subject-fact table, and the rows of the ADDRESS table are unique until you join them with the rows of the STREET_ADDRESS table.

Separating data into single subject-facts is the only proper way to achieve 1NF normalized results. Online transaction processing (OLTP) systems typically require normalized tables to avoid insertion, update, and delete anomalies. Interestingly, joining two normalized result sets yields a denormalized result set.

If you query the data with an INNER JOIN from these tables, the data will look very similar to the original problem data. As shown in this query:

```
SELECT   a.contact_id
,        a.address_id
,        sa.line_id
,        sa.street_address
,        a.city
,        a.state_province AS state
FROM     address a INNER JOIN street_address sa
ON       a.address_id = sa.address_id;
```

It generates the following as a result from the join. You should note that the one side of the one-to-many binary relationship is repeating, while the many side is unique. Joining tables always repeats copies of the columns on the one side of a one-to-many binary relationship for rows of columns from the many side of the relationship.

```
+------------+------------+---------+----------------+---------+-------+
| contact_id | address_id | line_id | street_address | city    | state |
+------------+------------+---------+----------------+---------+-------+
|          1 |          1 |       1 | 1111 Broadway  | Oakland | CA    |
|          2 |          1 |       2 | Suite 500      | Oakland | CA    |
|          3 |          1 |       3 | MS-5045        | Oakland | CA    |
+------------+------------+---------+----------------+---------+-------+
```

NOTE
You need to perform a nested join with a procedural language to get a match between one row and many rows into one structure.

It's important to note that almost every join result is in UNF. While that's not the way you store data to avoid insertion, update, and deletion anomalies, it is the way the end user consumes the information.

1NF Data Warehousing Solution Data warehousing models are online analytical processing (OLAP) systems. OLAP systems differ from OLTP systems in many ways, but the principal design difference is that you query data rather than insert, update, or delete data. This fact opens up a design pattern suited for queries that is not effective for transactions—flattening.

Flattening a nonrepeating column into a series of columns requires creating a column for every unique row that may occur. This means you replace the rows in the `STREET_ADDRESS` table with the `STREET_ADDRESS1`, `STREET_ADDRESS2`, and `STREET_ADDRESS3` columns.

The following is a definition of such a table:

```
+-----------------+-----------------+------+-----+---------+
| Field           | Type            | Null | Key | Default |
+-----------------+-----------------+------+-----+---------+
| address_id      | int(10) unsigned | NO   | PRI | NULL    |
| contact_id      | int(10) unsigned | NO   |     | NULL    |
| address_type    | int(10) unsigned | NO   |     | NULL    |
| street_address1 | varchar(30)     | NO   |     | NULL    |
| street_address2 | varchar(30)     | NO   |     | NULL    |
| street_address3 | varchar(30)     | NO   |     | NULL    |
| city            | varchar(30)     | NO   |     | NULL    |
| state           | varchar(30)     | NO   |     | NULL    |
| postal_code     | varchar(20)     | NO   |     | NULL    |
+-----------------+-----------------+------+-----+---------+
```

The data in a flattened model appears like this:

```
+------------+-----------------+-----------------+-----------------+
| address_id | street_address1 | street_address2 | street_address3 |
+------------+-----------------+-----------------+-----------------+
|          1 | 1111 Broadway   | Suite 500       | MS-5045         |
+------------+-----------------+-----------------+-----------------+
```

It's got one obvious problem. When the data encounters a fourth street address value, you must change the structure of the table (a fact-table in a data warehouse). As rule, data warehouses understand the OLTP data and would never design without a complete set of columns, and that's why this type of solution works in data warehouse systems.

Second Normal Form

Second normal form (2NF) exists when a table is already in first normal form and all non-key columns depend on all of the key columns, where the list of key columns goes from 1 to n. The list of key columns makes up the natural key of a table, or the list of columns that makes any row unique in a table. A table that has only a single column as the natural key, like a vehicle identification number, is automatically in second normal form because there can't be a dependency on part of the key (or one column of a multiple column key).

A table that has two or more columns as a natural key may contain one or more non-key columns that has a partial dependency on one or a set of columns less than all the columns in the key (or primary key). This typically means you have created a table that contains two subjects. The following shows the definition of the 1NF RENTAL table before converting it to 2NF (and yes, it's a bad design):

```
+--------------------+------------------+------+-----+---------+
| Field              | Type             | Null | Key | Default |
+--------------------+------------------+------+-----+---------+
| rental_id          | int(10) unsigned | NO   | PRI | NULL    |
| customer_id        | int(10) unsigned | NO   |     | NULL    |
| check_out_date     | date             | NO   |     | NULL    |
| return_date        | date             | NO   |     | NULL    |
| item_barcode       | varchar(20)      | NO   |     | NULL    |
| item_type          | int(10) unsigned | NO   |     | NULL    |
| item_title         | varchar(60)      | NO   |     | NULL    |
| item_subtitle      | varchar(60)      | YES  |     | NULL    |
| item_rating        | varchar(8)       | NO   |     | NULL    |
| item_rating_agency | varchar(4)       | NO   |     | NULL    |
| item_release_date  | date             | NO   |     | NULL    |
+--------------------+------------------+------+-----+---------+
```

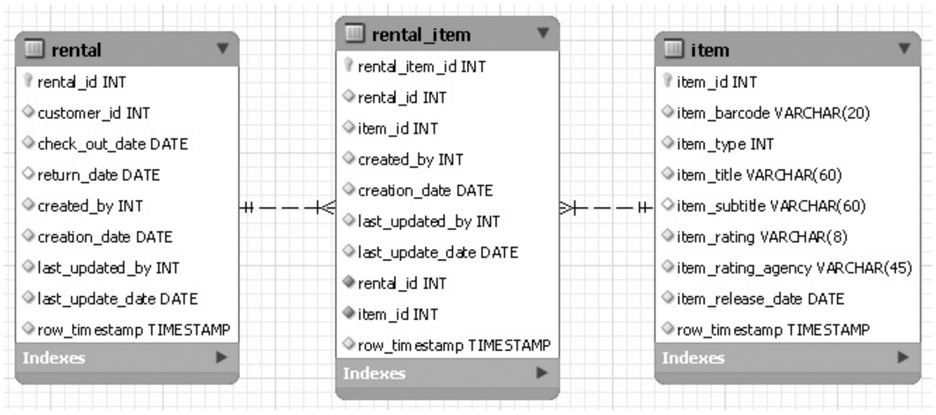
The preceding 1NF RENTAL table contains two subject-facts: one is the rental and the other is the item subject-fact. The natural key for the rental is a composite key of the CUSTOMER_ID, CHECK_OUT_DATE, and RETURN_DATE columns, and the natural key for the item is the ITEM_BARCODE column. Together the four columns could become the natural key for this table, but then the table would be in 1NF not 2NF.

The surrogate RENTAL_ID column maps to the four-column composite natural key. Unfortunately, all the other item columns depend on only part of the natural key, and this design limits a rental to a single item. You can move this table from 1NF to 2NF by dividing it into two subject-fact tables. Simply removing all the columns that start with item to a new ITEM table promotes the design from 1NF to 2NF.

Replacing those columns with an ITEM_ID foreign key provides an external reference to the RENTAL table, but limits rentals to a single item. While this type of resolution typically resolves one-to-one relationships between the subject-fact table and nested subject-fact table, it doesn't fix a one-to-many relationship between them.

Typically, business rules would support a rental of one to many items. That means you shouldn't add the foreign key to the RENTAL table to fix this problem.

You should create an association table that will hold a copy of the surrogate primary keys from the RENTAL and ITEM tables, like the following:



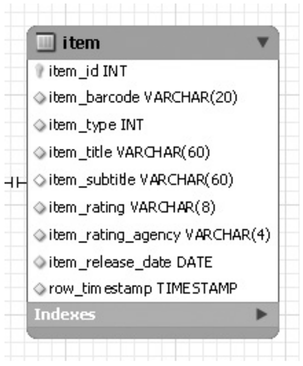
This section has explained how you fix partial dependencies by removing the nested subject-fact table and replacing it with a foreign key when there is a one-to-one relationship. It also explained how to fix a one-to-many relationship by removing the nested subject-fact table and creating an association table between the original and new tables.

Third Normal Form

Third normal form (3NF) exists when a table is already in second normal form and there are no transitive dependencies. A transitive dependency means a non-key column or set of columns is dependent on another column that generally isn't part of the natural key.

Since all non-key columns should be wholly dependent on the primary key, a transitive dependency exists when a column's functional dependency routes through another column or set of columns on its way to the primary key. This type of relationship indicates that there are at least two subject-facts in a table, but the dependency is not a column in the natural key.

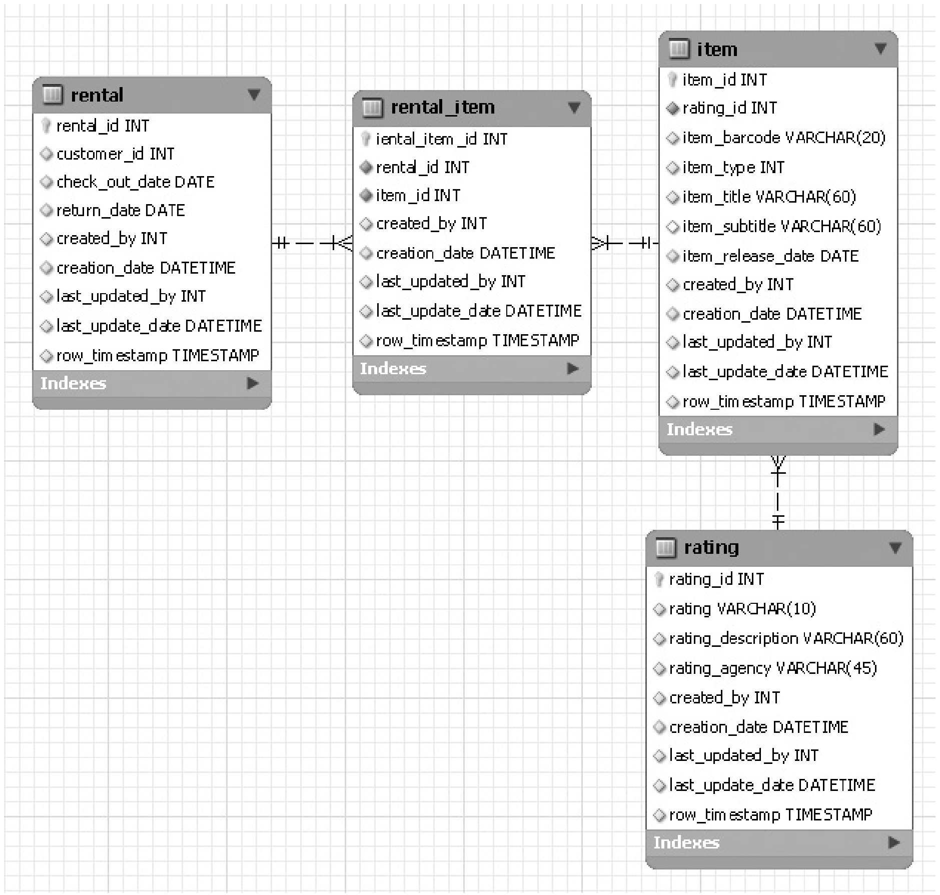
Our example table for this section is the ITEM table from the previous example, as shown:



The ITEM_RATING column has a transitive dependency through the ITEM_RATING_AGENCY column. From a business perspective, a PG rating depends on a Motion Picture of America Association (MPAA) rating agency and an E10+ rating depends on the Entertainment Software Rating Board (ESRB).

You fix a transitive dependency by applying the same technique used to fix a partial dependency. You remove the two columns from the ITEM table and put them in their own table or a COMMON_LOOKUP table.

The transitive dependency in this instance is a one-to-one relationship between the subject-fact table and nested subject-fact table, which means you add a foreign key column that points back to the new RATING table. The solution is shown in the following illustration:



You should generally make the column or set of columns that acts as the intermediary column the natural primary key in a new table. The column or columns that had a transitive dependency on the intermediary column should also move to the new table. A RATING_ID foreign key column in the new table replaces the original columns, but allows access through joins to them. The following ER Model shows the fix to the problem.

Breaking the two subject-fact tables into one subject-fact table successfully promotes a 2NF table to a 3NF table. The foreign key links in the original table links the RATING and RATING_AGENCY column values to the base table.

This section has shown you how to migrate from 0NF to 1NF, 1NF to 2NF, and 2NF to 3NF. The key to the changes are ensuring that all tables contain a single subject-fact.

Summary

This chapter explained the basic of data modeling theory and systems, as well as normalization practices. You should be positioned to design database models.

Mastery Check

The mastery check is a series of true or false and multiple choice questions that let you confirm how well you understand the material in the chapter. You may check Appendix A for answers to these questions.

True or False:

1. ___External identifiers hold primary key columns.
2. ___Internal identifiers hold natural key columns.
3. ___External references hold foreign key columns.
4. ___Non-key data includes surrogate key columns.
5. ___Both surrogate and primary keys are candidate keys.
6. ___UNF is the same as HNF when a table is in 0NF.
7. ___HNF is only 3NF or higher.
8. ___A table is in 1NF when it has a partial dependency.
9. ___A table is in 1NF when it has a transitive dependency.
10. ___There is nothing higher than 3NF.

Multiple Choice:

11. MySQL Workbench uses which data modeling system?

    A. Chen

    B. Chen-Martin

    C. Information engineering

    D. UML

    E. None of the above

12. Which relationship line models an ID-dependent relationship?

- **A.** Aggregation line
- **B.** Composition line
- **C.** Inheritance line
- **D.** Relationship line
- **E.** None of the above

13. What type of relationship models a primary-to-foreign key association?

- **A.** Unary
- **B.** Binary
- **C.** N-ary
- **D.** Inheritance
- **E.** Aggregation

14. What type of variable holds only one thing? (Multiple answers are possible.)

- **A.** Compound variable
- **B.** Composite variable
- **C.** Scalar variable
- **D.** Primitive variable
- **E.** None of the above

15. Most relationships between tables are?

- **A.** Independent
- **B.** Dependent
- **C.** ID-dependent
- **D.** Non-ID–dependent
- **E.** None of the above