LESSON 4: Understanding Data Storage

You work for a large warehouse distribution company that provides outdoor camping equipment to over 90 stores in North America. You have just received an office memo stating that the company has purchased a second store in the state of Washington. Your boss has asked you to add a new table to the database for this store, saying that the table should be populated with the same equipment that is sold in the first Washington store. She wants to ensure both that database integrity is maintained and that the new table remains normalized to the third normal form.

In this lesson, you will learn about database normalization, the five most common levels of normalization, and the purpose of normalization as it relates to database integrity. You will also learn why foreign, primary, and composite keys play an integral role in referential integrity.

Normalizing a Database

THE BOTTOM LINE

As a database administrator, you must understand the reasons for normalization, the five most common levels of normalization, and how to normalize a database to the third normal form.

The main reason for using normalization techniques in data storage arose in the days when storage cost a great deal more than it does today. Indeed, ***normalization***, in a nutshell, is the elimination of ***redundant data*** to save space.
Now that you understand the general definition of normalization, let's look more closely at this concept and its application in database design.

TAKE NOTE*

Normalization is the elimination of redundant data to save space.

Understanding Normalization

Normalization is based entirely on the data design and organization processes that are derived from the rules used when building and designing relational databases. Therefore, understanding what relational databases are and the importance of good design methodologies is extremely important.

CERTIFICATION READY

What are the first three normalization forms, and how do they differ from one another?

4.1

Normalization, by definition, is the process of organizing data in order to reduce redundancy by dividing a database into two or more tables and then defining table relationships. The objective of this operation is to isolate data so that additions, deletions, and modifications occurring in each field can be made inside one table and then propagated throughout the rest of the database using these defined relationships. There are five ***normalization forms (NFs)***, of which we'll focus on the first three:

- **First normalization form (1NF):** Eliminate repeating groups
- **Second normalization form (2NF):** Eliminate redundant data
- **Third normalization form (3NF):** Eliminate columns not dependent on key
- **Fourth normalization form (4NF):** Isolate independent multiple relationships
- **Fifth normalization form (5NF):** Isolate semantically related multiple relationships

Let's now look at each one of these forms in a little more depth. Taking a detailed look at the normal forms moves this lesson into a more formal study of relational database design. Contrary to popular opinion, the forms are not a progressive methodology, but they do represent a progressive level of compliance. Technically, you can't be in 2NF until you have met 1NF; therefore, don't plan on designing an entity and moving it through the first normal form to the second normal form, and so on, because each normal form is simply a different type of data integrity with different requirements that must be fulfilled.

Understanding the First Normal Form

The ***first normalized form (1NF)*** means the data is in an entity format, which basically means that the following three conditions must be met:

- The table must have no duplicate records. Once you have defined a primary key for the table, y
- The table also must not have multivalued attributes, meaning that you can't combine in a si valid for a column. For an example of the first normal form in action, consider the listing o Adventures database. Table 4-1 shows the base camp data in a model that violates the first 1, Tour 2, and Tour 3) is not unique. In other words, there are three values assigned for Tou
- The entries in the column or attribute must be of the same data type.

Table 4-1: Base camp data (before): Violating the first normal form

TOUR ENTITY		BASECAMP ENTITY	
BASECAMPID(FK)	**TOUR**	**BASECAMPID (PK)**	**NAME**
1	Appalachian Trail	1	Asheville
1	Blue Ridge Parkway Hike	2	Cape Hatteras
2	Outer Banks Lighthouses	3	Freeport
3	Bahamas Dive	4	Ft. Lauderdale
4	Amazon Trek	5	
5	Gauley River Rafting		

Understanding the Second Normal Form

The ***second normal form (2NF)*** ensures that each attribute does in fact describe the entity. This form is entirely based on dependency: specifically, the attributes of the entity in question, which is not part of a candidate key, must be functionally dependent upon the entire primary key. What ends up happening on occasion is that combined primary keys run into trouble with the second normal form if the attributes aren't dependent on every attribute in the primary key. If an attribute depends on one of the primary key attributes but not the others, then it becomes a partial dependency, which violates the second normal form. To better understand violations of the second normal form, Table 4-3 shows an example of the same base camp data when it is not in 2NF, and Table 4-4 shows the same data after it has been conformed.

Table 4-3: Base camp data (before): Violating the second normal form

PK-BASECAMP	PK-TOUR	BASE CAMP PHONENUMBER
Asheville	Appalachian Trail	828-555-1212
Asheville	Blue Ridge Parkway Hike	828-555-1212
Cape Hatteras	Outer Banks Lighthouses	828-555-1213
Freeport	Bahamas Dive	828-555-1214
Ft. Lauderdale	Amazon Trek	828-555-1215
West Virginia	Gauley River Rafting	828-555-1216

Table 4-4: Base camp data (after): Conforming to the second normal form

TOUR ENTITY		BASE CAMP ENTITY	
PK-BASE CAMP	**PK-TOUR**	**BK-BASE CAMP**	**PHONENUMBER**
Asheville	Appalachian Trail	Asheville	828-555-1212
Asheville	Blue Ridge Parkway Hike	Cape Hatteras	828-555-1213
Cape Hatteras	Outer Banks Lighthouses	Freeport	828-555-1214
Freeport	Bahamas Dive	Ft. Lauderdale	828-555-1215
Ft. Lauderdale	Amazon Trek	West Virginia	828-555-1216
West Virginia	Gauley River Rafting		

Understanding the Third Normal Form

The ***third normal form (3NF)*** checks for transitive dependencies. A transitive dependency is similar to a partial dependency in that both refer to attributes that are not fully dependent on a primary key. A dependency is considered transient when attribute1 is dependent on attribute2, which is then dependent on the primary key.

When looking at whether there is a violation in either the second or third normal form, remember that each attribute is directly or indirectly tied to the primary key. Therefore, the second normal form is violated when an attribute depends on only part of the key, and the third normal form is violated when the attribute depends on the key but also on another nonkey attribute. The central phrase to remember in describing the third normal form is that every attribute must "provide a fact about the key, the whole key, and nothing but the key." Just as with the second normal form, the third normal form is resolved by moving the nondependent attribute to a new entity.

To better understand violations of the third normal form, Table 4-5 shows an example of the same base camp data when it is not in 3NF, and Table 4-6 shows the data after it has been conformed.

Table 4-5: Base camp data (before): Violating the third normal form

BASE CAMP ENTITY			
BASECAMPPK	**BASECAMPPHONENUMBER**	**LEADGUIDE**	**DATEOFHIRE**
Asheville	1-828-555-1212	Jeff Davis	5/1/99
Cape Hatteras	1-828-555-1213	Ken Frank	4/15/97
Freeport	1-828-555-1215	Dab Smith	7/7/2001
Ft. Lauderdale	1-828-555-1215	Sam Wilson	1/1/2002
West Virginia	1-828-555-1216	Lauren Jones	6/1/2000

Table 4-6: Base camp data (after): Conforming to the third normal form

TOUR ENTITY		LEADGUIDE ENTITY	
BASECAMPPK	**LEADGUIDE**	**LEADGUIDEPK**	**DATEOFHIRE**
Asheville	Jeff Davis	Jeff Davis	5/1/99
Cape Hatteras	Ken Frank	Ken Frank	4/15/97
Freeport	Dab Smith	Dab Smith	7/7/2001
West Virginia	Lauren Jones	Lauren Jones	6/1/2000

HOW TO NORMALIZE A DATABASE TO THE THIRD NORMAL FORM

There are two basic requirements for a database to be in third normal form:

- The database must already meet the requirements of both 1NF and 2NF.
- The database must not contain any columns that aren't fully dependent upon the primary ke

In order to understand how a database can be put into the third normal form, let's imagine that we have a table of widget orders that contains the following attributes:

- Order Number (primary key)
- Customer Number
- Unit Price
- Quantity
- Total

Remember, our first requirement is that the table must satisfy the requirements of both 1NF and 2NF. Are there any duplicative columns? No. Do we have a primary key? Yes, the order number. Therefore, we satisfy the requirements of 1NF. Are there any subsets of data that apply to multiple rows? No, so we also satisfy the requirements of 2NF.

Now, are all the columns fully dependent upon the primary key? The customer number varies with the order number, and it doesn't appear to depend on any of the other fields. What about the unit price? This field could be dependent on the customer number in a situation where we charged each customer a set price. However, from the information provided in the table fields, we can sometimes charge the same customer different prices. Therefore, the unit price is fully dependent on the order number. The quantity of items also varies from order to order, so we're okay there.

What about the total? It looks as if we might be in trouble here. The total can be derived by multiplying the unit price by the quantity, and therefore it's not fully dependent on the primary key. We must remove it from the table to comply with the third normal form. Perhaps we could replace our original attributes with the following attributes:

- Order Number
- Customer Number
- Unit Price
- Quantity

Now our table is in 3NF. But, you might ask, what about the total? This is a derived field, and it's best not to store it in the database at all. We can simply compute it "on the fly" when performing database queries. For example, we might have previously used this query to retrieve order numbers and totals:

```
SELECT OrderNumber, Total

FROM WidgetOrders
```

We can now use the following query in order to achieve the same results, without violating normalization rules:

```
SELECT OrderNumber, UnitPrice * Quantity AS Total

FROM WidgetOrders
```

Understanding the Fourth Normal Form

The ***fourth normal form (4NF)*** involves two independent attributes brought together to form a primary key along with a third attribute. But, if the two attributes don't really uniquely identify the entity without the third attribute, then the design violates the fourth normal form.

Understanding the Fifth Normal Form

The ***fifth normal form (5NF)*** provides a method for designing complex relationships involving multiple (usually three or more) entities.

Typically, database administrators feel that satisfying the requirements of the first, second, and third normal forms is enough. The fourth and fifth normal forms may be complex, but violating them can cause severe problems.

It is important to look at database design as a whole and not just as designing something to fulfill half of the needs of your users, employers, and so forth. It's not necessarily whether a number of entities are used or not used; rather, it's a matter of properly aligning the attributes and keys. Any violation of the normal forms can cause a cascading effect with multiple violations and inefficient databases.

Normalization reduces locking contention and improves multiple-user performance. Locks are essential mechanisms that are used to prevent simultaneous changes to the database, such as two different users making changes to the same record. Without locks, a change made by one transaction could be overwritten by another transaction that executes at the same time. Last, normalization has these three advantages:

- **Development costs:** Although it may take longer to design a normalized database, such databases are easier to work with and reduce development costs.
- **Usability:** Placing columns in the correct table makes it easier to understand a database and write correct queries. This helps reduce design time and cost.

- **Extensibility:** A non-normalized database is often more complex and therefore more difficult to modify. This leads to delays in rolling out new databases and increases in development costs

> **TAKE NOTE***
> Normalization also reduces locking contention and improves multi-user performance.

Understanding Primary, Foreign, and Composite Keys

> **THE BOTTOM LINE**
> In this section, you'll learn about the reasons for using keys in a database. You'll also explore how to choose appropriate primary keys, select appropriate data types for keys, select appropriate fields for composite keys, and understand the relationship between foreign and primary keys.

Three different types of constraints available within SQL Server ensure that you are able to maintain database integrity: primary keys, foreign keys, and composite (unique) keys. A ***unique key constraint*** will allow you to enforce the uniqueness property of columns, in addition to a primary key within a table. A unique constraint acts similarly to a primary key but with two important differences:

1. Columns containing a unique key constraint may contain only one row with a NULL value. You cannot have two rows containing a NULL value in the same option, as that would violate the unique constraint's duplicate value error.

   > **CERTIFICATION READY**
   > What are the differences between a primary key and a foreign key?
   > 4.2

2. A table may have multiple unique constraints'

CREATE A UNIQUE CONSTRAINT

GET READY. Before you begin this exercise, be sure to launch the SQL Server Management Studio application and connect to the database you wish to work with. Then, follow these steps:

1. Using SQL Server Management Studio, open the table in which you want to create the constraint in Design View. You can do so by right-clicking the table and selecting Design from the menu that appears, as shown in Figure 4-1.

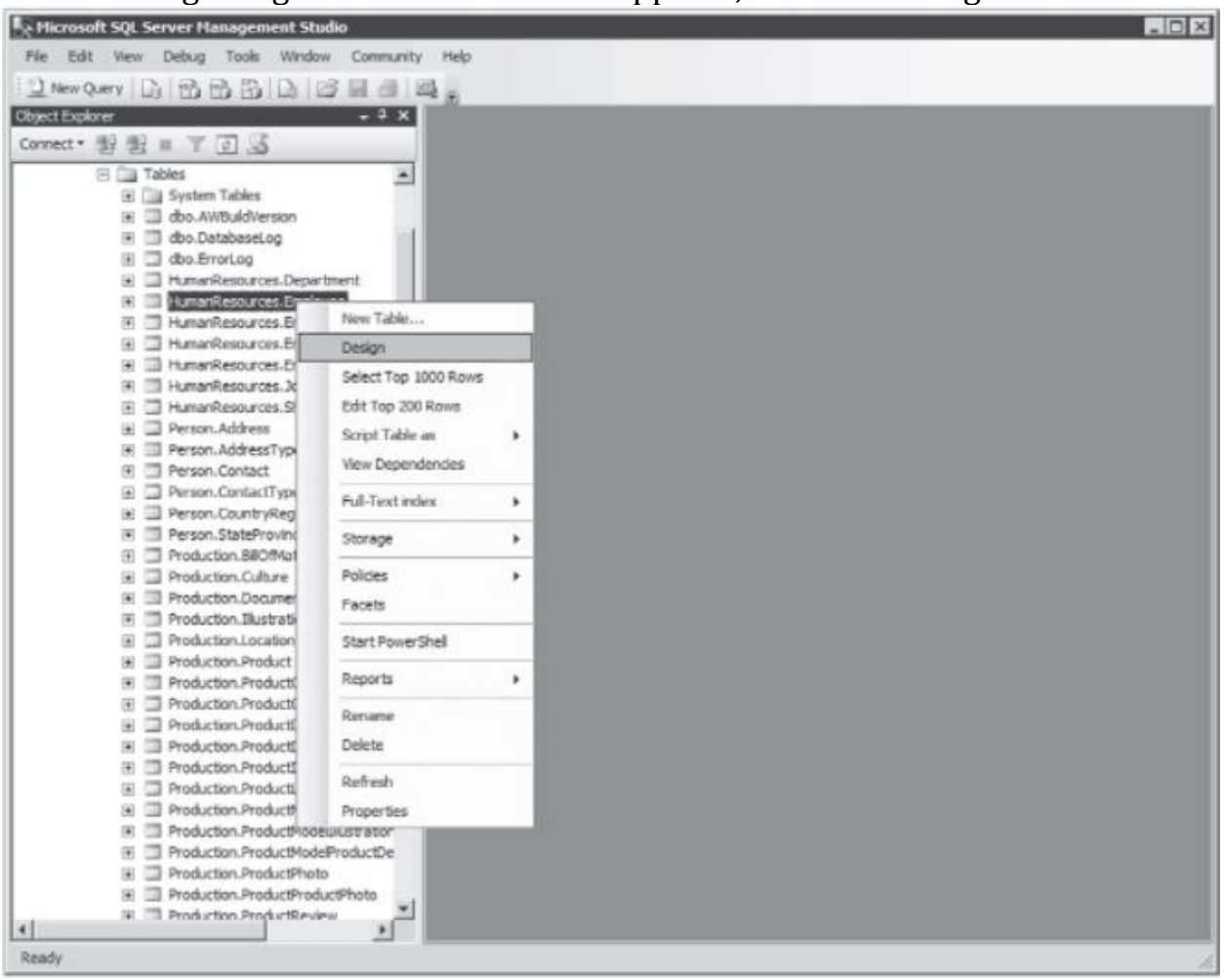

Figure 4-1 Design view

2. From the Table Designer drop-down menu at the top of the toolbar, select Indexes/ Keys, as shown in Figure 4-2. This opens the Indexes/Keys window. You will notice that the table already has a primary key constraint identified, as shown in Figure 4-3.

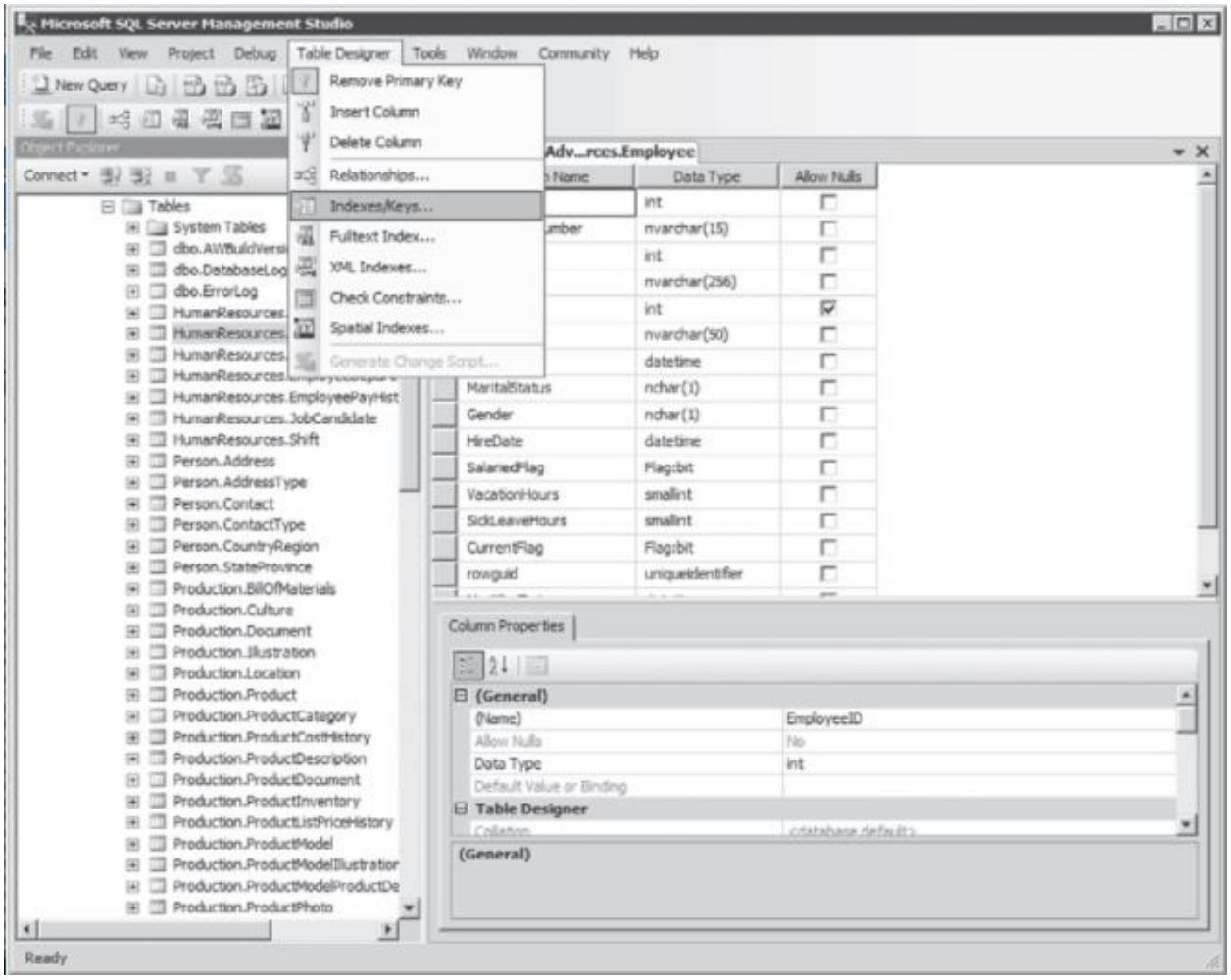

Figure 4-2 Indexes/Keys

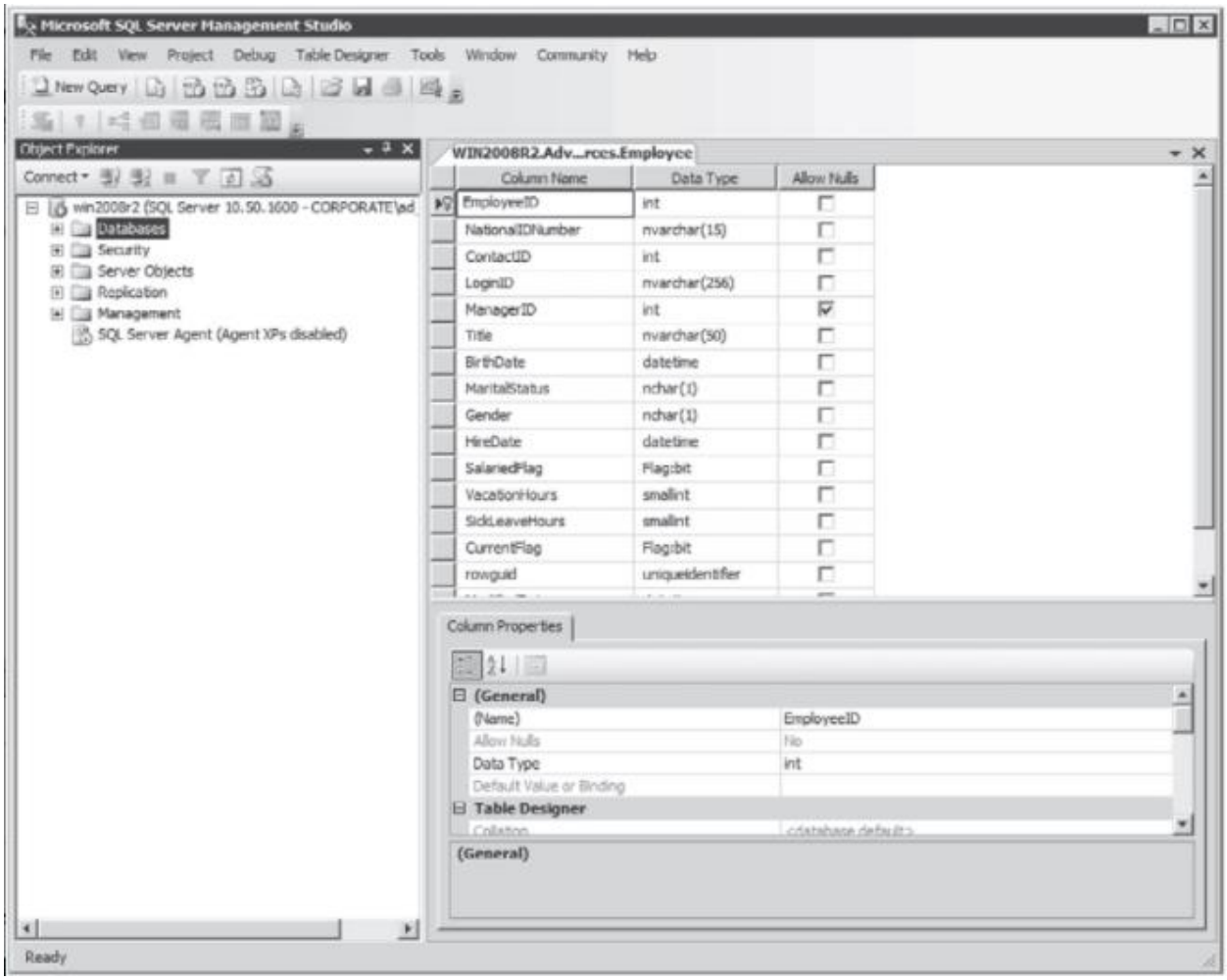

Figure 4-3 Connecting to a database

3. Click on the Add button to create a new key.
4. Click the Type property in the right side of the property box and change it from the default of Index to Unique Key, as shown in Figure 4-4.

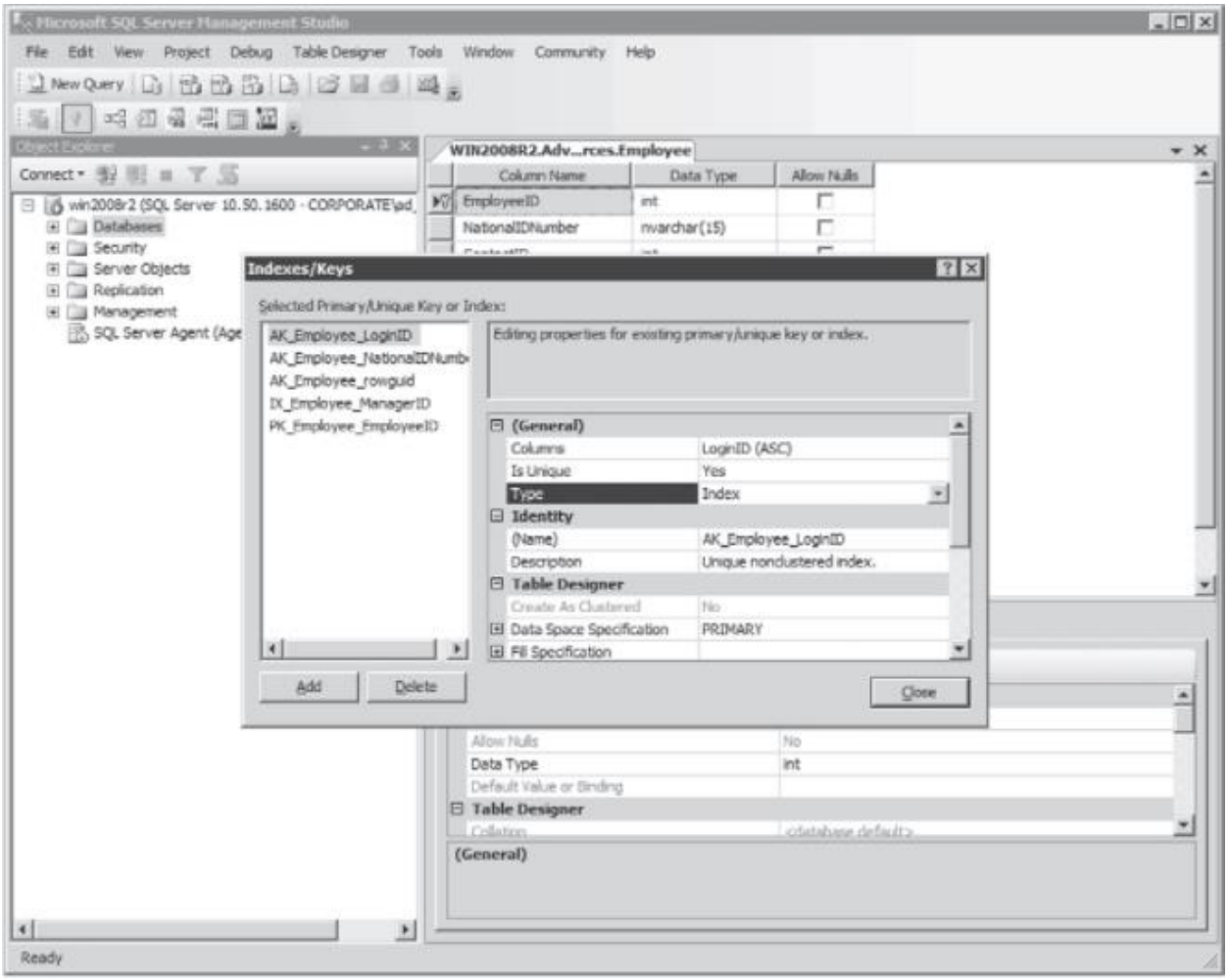

Figure 4-4 Type property box

5. Click the ellipsis (. . .), which is found beside the Columns property section. See Figure 4-5. You can now select the columns you wish to include in your unique constraint.

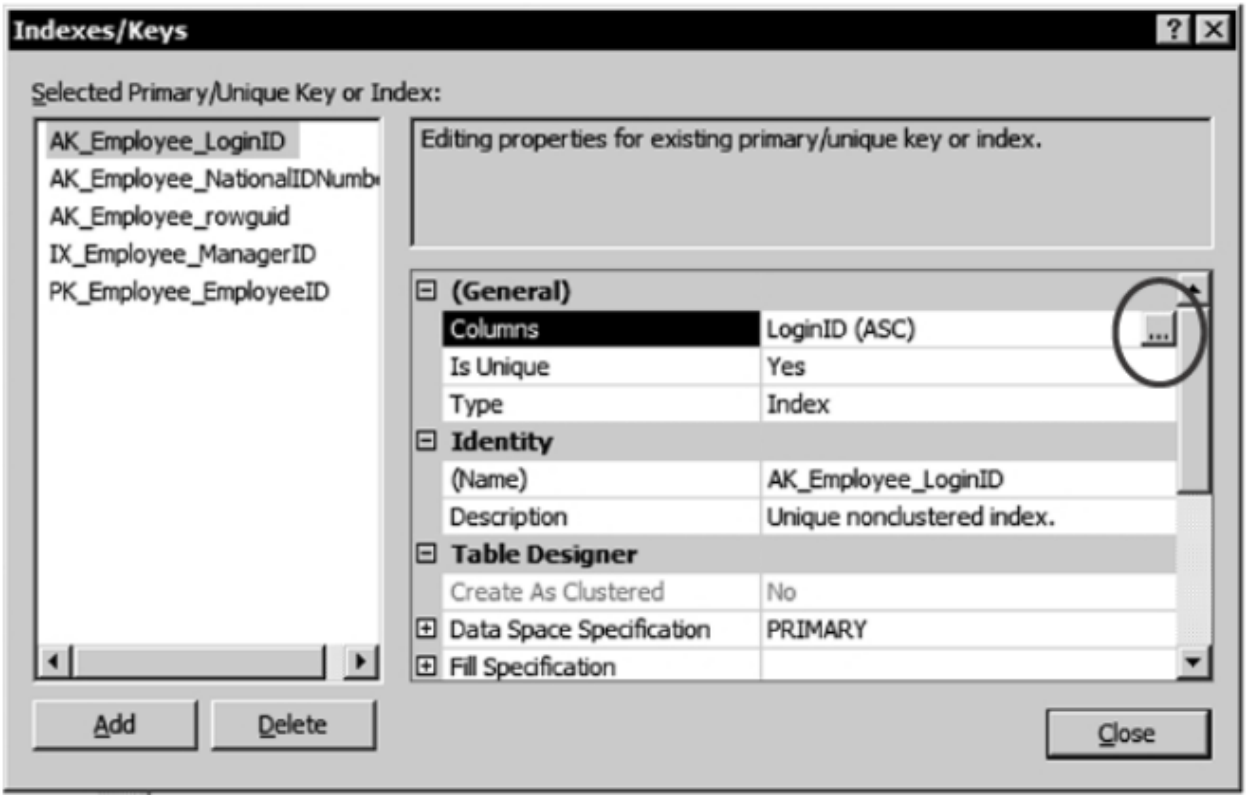

Figure 4-5 Selecting columns to add

6. Click the Close button.
7. Save your newly created constraint by selecting Save all from the File menu, as shown in Figure 4-6.

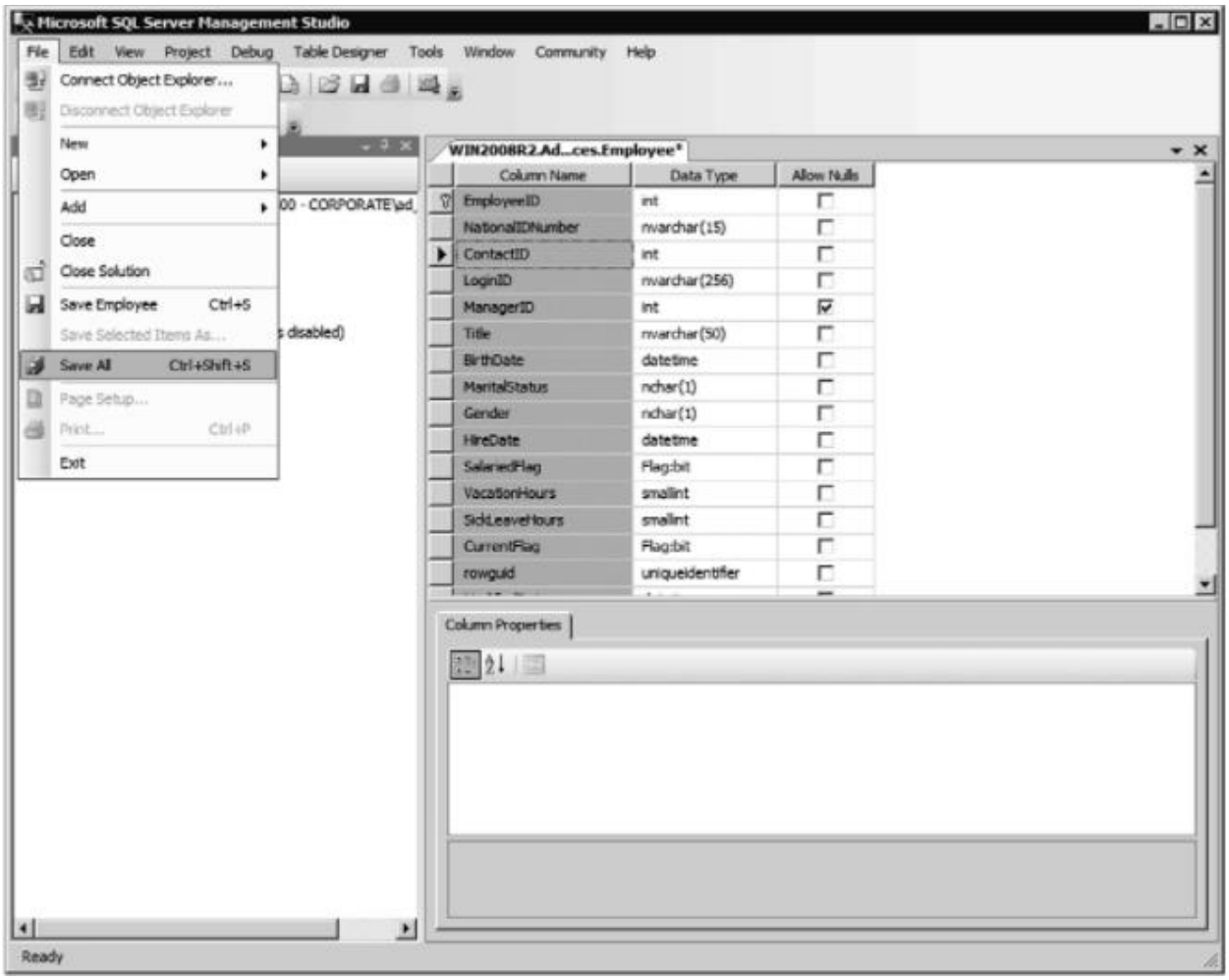

Figure 4-6 Saving a new constraint

You have now created your own unique constraint.

PAUSE. Leave the SSMS interface open for the next exercise.

Understanding Primary Keys

Perhaps the most important concept of designing any database table is ensuring that it has a ***primary key***—in other words, an attribute or set of attributes that can be used to uniquely identify each row. Every table must have a primary key; without a primary key, it's not a valid table. By definition, a primary key must be unique and must have a value that is not null.

In some tables, there might be multiple possible primary keys to choose from, such as employee number, driver's license number, or another government-issued number such as a Social Security number (SSN). In this case, all the potential primary keys are known as candidate keys. Candidate keys that are not selected as the primary key are then known as alternate keys.

Remember, in the initial database diagramming phase, a primary key might be readily visible—for instance, it could be an employee number or a manufacturer name; however, more often than not, there is no clearly recognizable, uniquely identifying value for each item in most real-world scenarios.

Understanding Foreign Keys

Throughout the lessons in this book, you have been inundated with the terminology of relational databases. This terminology also carries forward into the use of index keys, such as ***foreign keys***. When two tables relate to each other, one of them will act as the primary table and the other will act as the secondary table. In order to connect the two tables, the primary key is replicated from the primary to the secondary table, and all the key attributes duplicated from the primary table become known as the foreign key. Although this may at times be referred to as a parent-child relationship, enforcing the foreign key attribute is actually referred to as referential integrity (refer to the discussion in the previous lesson of referential integrity).

To get a better visual idea of this type of relationship, look at Figure 4-7, which shows an order's primary key duplicated in the order detail table, thus providing the link between the two tables.

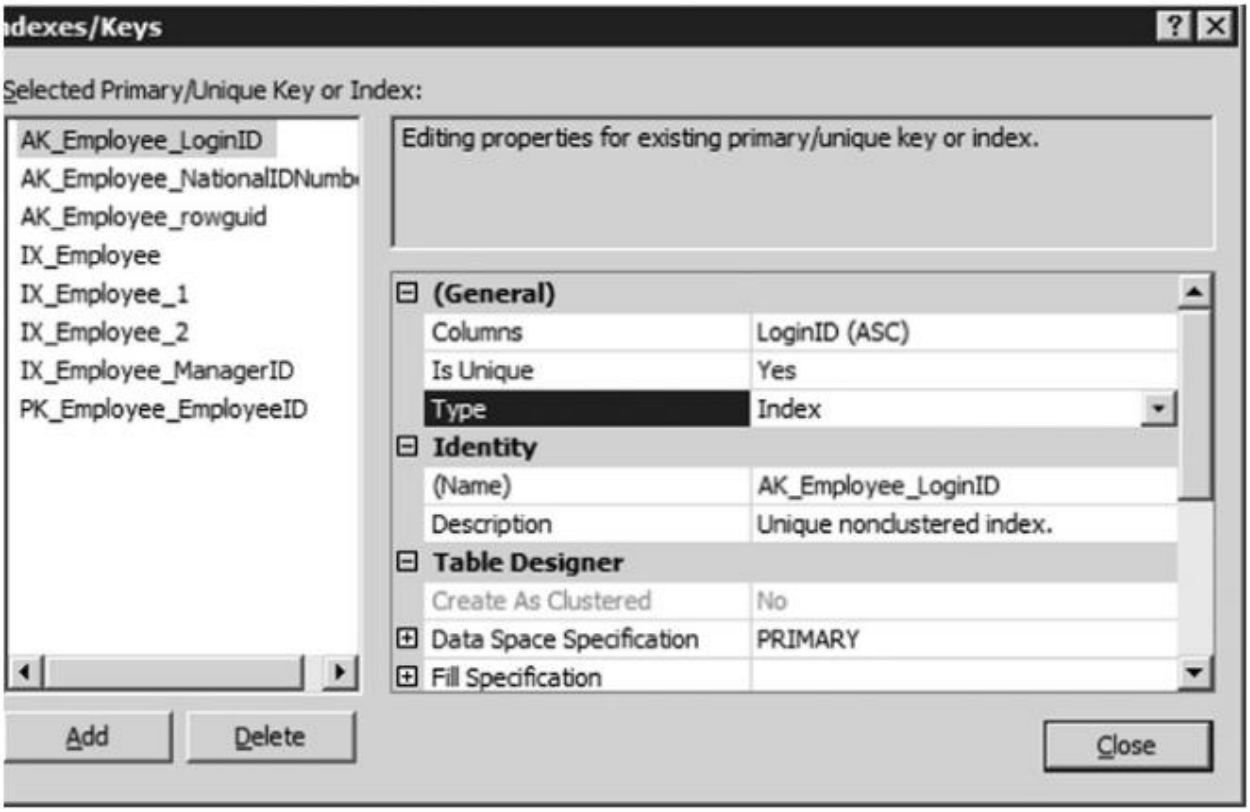

Figure 4-7 Primary key duplication

When discussing foreign keys and primary keys, the terminology will almost always include the constraint as part of the description. As an example of the role of the ***foreign key constraint***, let's consider a retail store database. Each store has a table containing information about that store, such as employee information, products sold, inventory on hand, and more than likely information about customers. One way to reference the table data logically would be to create a Unit_Number field within the Employees table that would then contain the Unit_Number (the primary key of the Stores table) of the employee's store, thus creating a link between the two tables.

Is this a logical setup or not? Ask yourself what would happen if one of the stores closed in the future. In this situation, it's likely that all the employees associated with the store would be "orphaned," as they would be associated with a Unit_Number field that no longer exists. There is also the potential for human error—for instance, someone may inadvertently type an incorrect Unit_Number while entering an employee into the database, thus creating a number for a store that doesn't exist. This could prove a problem for payroll or other human resources actions.

These types of problems are referred to as relational integrity issues, as you learned in the previous lesson. Thankfully, SQL Server provides the necessary foreign key constraint to prevent this type of error. A foreign key then creates a relationship between

two tables by linking the foreign key in one of the tables to the primary key of the referenced table.

TAKE NOTE*
Every table must have a primary key; without a primary key, it's not a valid table. By definition, a primary key must be unique and must have a value that is not null.

CREATE A FOREIGN KEY USING SQL SERVER MANAGEMENT STUDIO

GET READY. To create a foreign key constraint using the SSMS interface, follow these steps. Before you begin, be sure to launch the SQL Server Management Studio application and connect to the database you want to work with.

1. In SSMS, open the table in which you wish to create a foreign key. Right-click the table and select Design view, as shown in Figure 4-8.

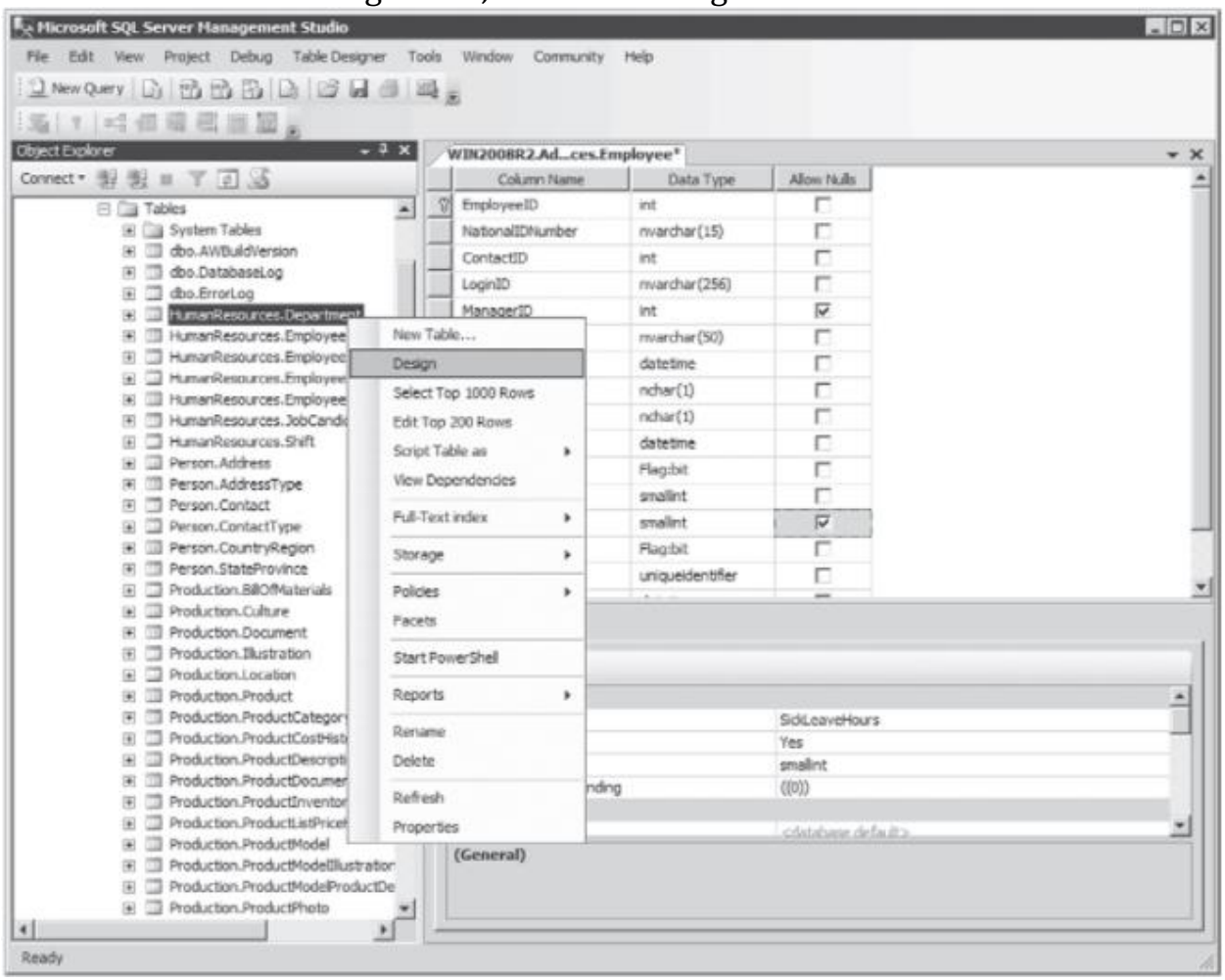

Figure 4-8 Design view

2. Select Relationships from the Table Designer drop-down menu, as shown in Figure 4-9.

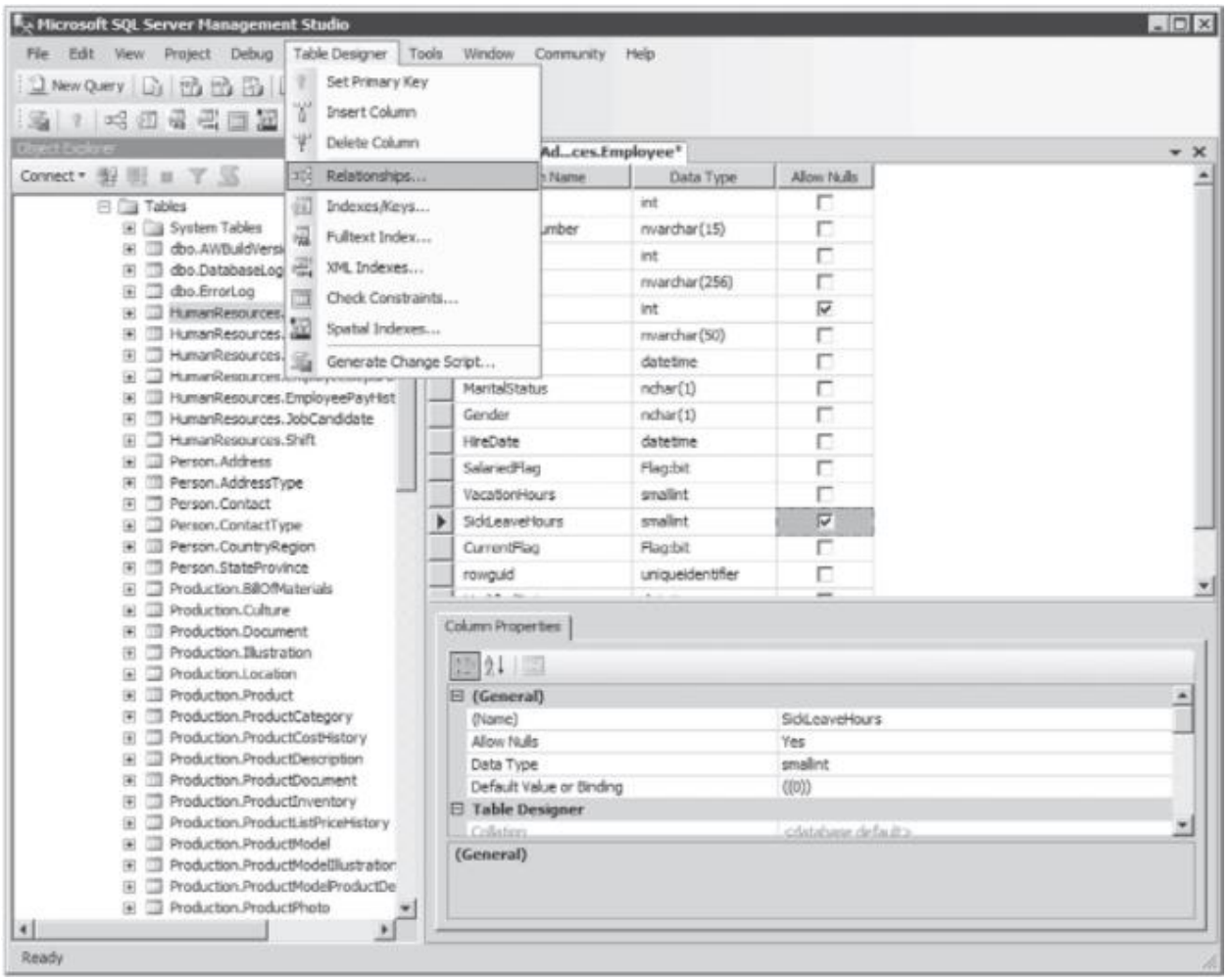

Figure 4-9 Selecting Relationships

3. Select the table to which you wish to add the foreign key constraint.
4. Click the ellipsis (. . .) beside the Tables and Columns Specification property dialog box, as shown in Figure 4-10.

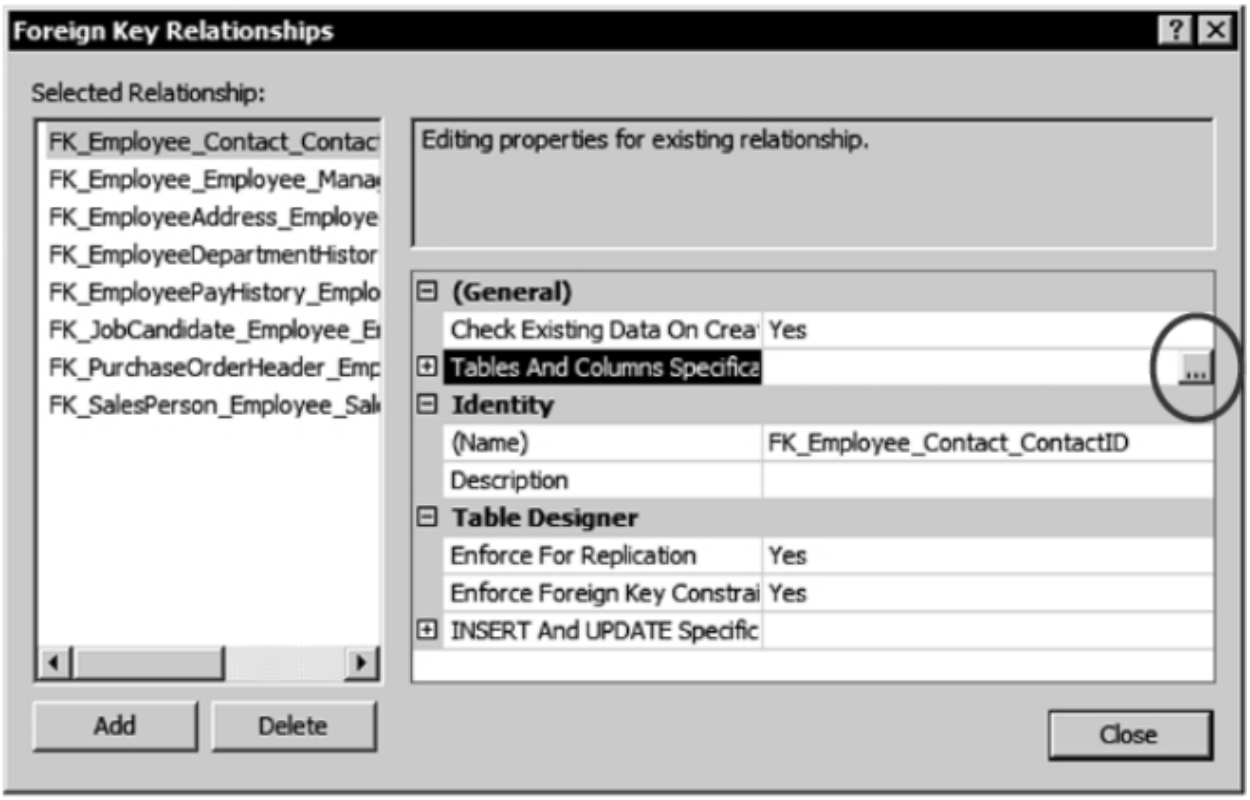

Figure 4-10 Tables and Columns Specification Box

5. Select the table that your foreign key refers to in the primary key table drop-down list, as shown in Figure 4-11.

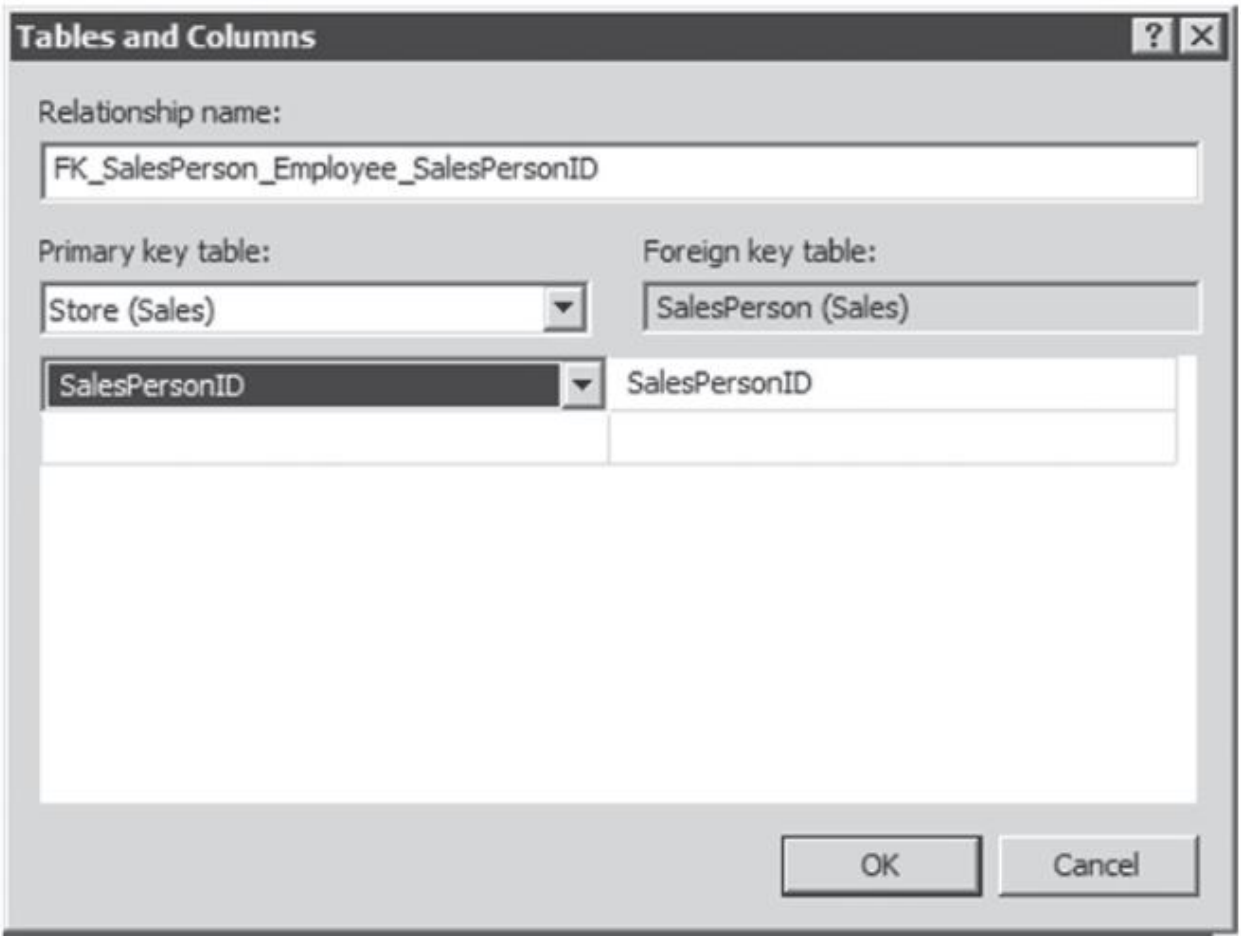

Figure 4-11 Selecting the table your foreign key refers to

6. Once you have completed adding the foreign key table information, click OK to close the dialog box.
7. Click the Close button.
8. Save your newly created constraint by selecting Save all from the File menu.

PAUSE. Leave the SSMS interface open for the next exercise

Now that this foreign key relationship has been created between two tables, SQL Server will require all associated values with the constraint in the foreign key table to have corresponding values in the primary key table. This does not require that the opposite happen (i.e., that the primary key have corresponding values in the foreign key table.) Remember, multiple values in the foreign key table can reference the same record in the primary key table.

Understanding Composite Primary Keys

One of the most confusing issues regarding primary keys is the definition of a composite primary key. A ***composite primary key*** occurs when you define more than one column as your primary key. Although many database administrators do not use composite primary keys and are not aware of them, they play an integral part in designing a good, solid data model.

As a simplified example, imagine that you take the tables in a database and categorize them based on two data types:

- Tables that *define* entities
- Tables that *relate* entities

The tables that define entities are the tables defining such things as customers, salespersons, and transactions related to sales. You can choose any column in these tables as a primary key, because in this discussion of composite primary keys, tables that define entities are not an important topic.

It is in the tables that relate entities that the composite primary key plays an important role. Using our previous example, suppose you have a system in place that tracks customers, and this system allows you to assign multiple products to multiple customers in order to indicate what they can or cannot order. You are thus looking at a "many-to-many" relationship between the Customer and Products tables. You already have a table for Customers and one for Products, and you also have a primary key selected for the ProductID column of the Products table and the CustomerID column of the Customers table. Next, you would need to look at how to define your CustomerProducts table.

The CustomerProducts table relates the customers to products, so the purpose of this table is to relate the two entities that have already been defined within the database. Oftentimes, when a database administrator is designing a table for the first time, much thought goes into ensuring that data integrity is at the forefront of the design guidelines; yet by ensuring the table's primary key is identified, a database administrator does ensure that data integrity is maintained. The bottom line is that in order to maintain data integrity, the primary key must form part of the design requirements for each table.

Many times a table is not designed to take into account the possible duplication of data inputs, and although many think that the UI (unique identifier) can handle any data duplication, an update to the table data can always occur, such as when a system is upgraded and data must be moved over, or some transaction must be restored from a backup. Hence, there is a necessity to ensure that data integrity is maintained.

If you begin to look at table design with an understanding of data integrity and defining a table's primary key, you will see that using a unique constraint will ensure that

integrity is maintained. Remember, a primary key is a set of columns in a table that uniquely identifies each row of data.

Understanding Clustered and Non-Clustered Indexes

THE BOTTOM LINE

In this section, you'll learn about clustered and non-clustered indexes and their purpose in a database.

As a database administrator, you should understand what the two types of indexes (clustered and non-clustered) do and what the role of these indexes is within a database environment. You're probably already familiar with the index in a textbook, which contains entries for particular subjects, words, and ideas. Whenever you want to quickly find information in the book, you can simply turn to this index. Indexing with databases, in the larger scheme of things, is exactly the same thing.

CERTIFICATION READY

What is the difference between a clustered and a non-clustered index?
4.3

In SQL Server, to retrieve data from a database, SQL server checks each row to look for the query on which you were trying to find information. Does this sound like an incredible amount of time spent inefficiently? If you answered yes, you would be correct! Thus, what SQL Server does (with the help of database administrators) is build and maintain a variety of indexes in order to locate and return commonly used fields quickly.

The only real drawbacks to indexing are the time it takes to build the actual indexes and the storage space the indexes require. One important decision in using indexes is figuring out what indexes are appropriate for your database, based on the types of queries you will perform. Remember, SQL Server allows you to create your indexes on either single or multiple columns, but the real speed gain will be on those indexes based on the column(s) inside the index.

Understanding Clustered Indexes

When you begin looking at implementing indexes, it is important to remember that each table can have only one ***clustered index*** that defines how SQL Server will sort the data stored inside the table. After all, because that data can only be sorted in one way, it simply is not possible to have two clustered indexes on the same table. It should also be mentioned that a clustered index is a physical construct, unlike most indexes, which are logical or software-based.

TAKE NOTE*

Only one clustered index is allowed for each table.

One important feature of SQL Server is its automatic creation of a clustered index when the primary key is defined for a table. A primary key makes it simple for you, as database administrator, to look at creating non-clustered indexes based on the columns in a table.

So far, we have given a simplistic overview of what indexes are and why they are created. It is important now to look at the basics of an index. An index is an on-disk (or stored) structure associated entirely with a table or a view that increases the speed of data retrieval. In order to create an index, a series of keys is built from one or more columns in each row within a table or a view. These keys are then stored in a structure called a B-tree that enables SQL Server to find the row(s) associated with those defined values much more quickly and efficiently. Figure 4-12 shows an example of a B-tree structure.

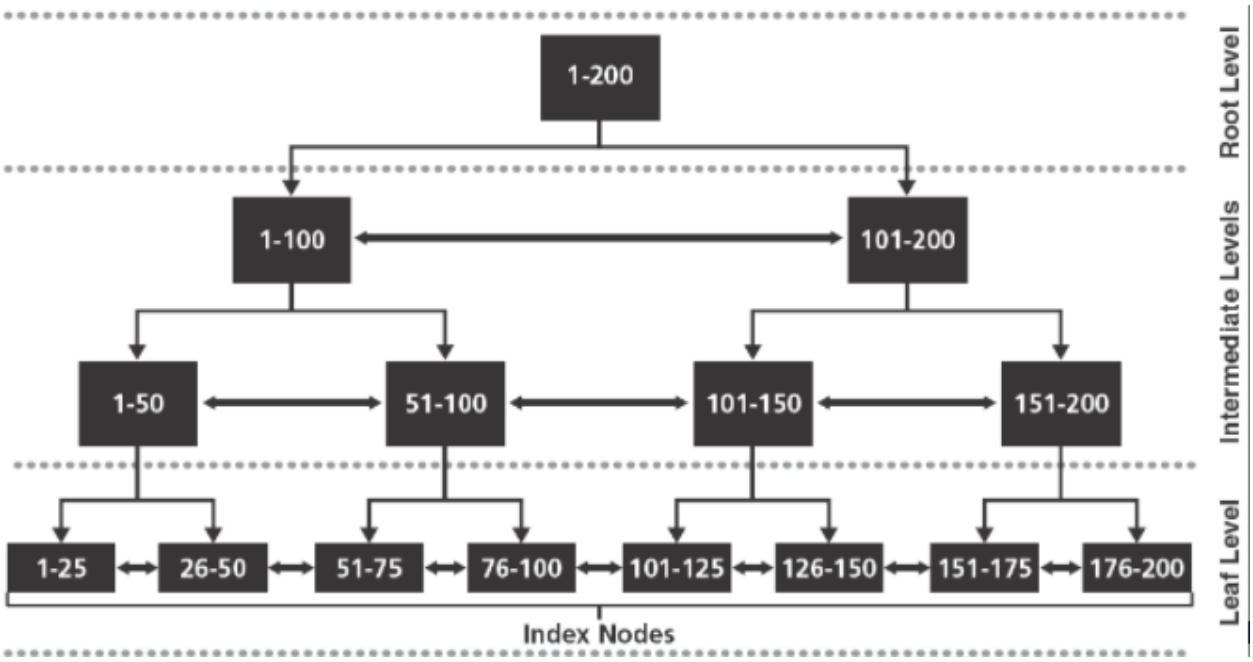

Figure 4-12 B-tree structure

In a clustered index, the data is sorted and stored in the table or view that is based on their respective key values. These columns are included within the index definition, and because the data in the rows themselves are sorted in only one order, this is why, as mentioned above, you can only have one clustered index per table.

A table with a clustered index is considered a clustered table; when a table has no clustered index, the data rows are then stored in an unordered structure called a heap. This brings us to the definition of a non-clustered index.

Understanding Non-Clustered Indexes

You have the freedom to create your own non-clustered indexes because they have a structure different from the clustered index structure. This is because a ***non-clustered index*** contains the non-clustered index key values, and each of those keys has a pointer to a data row that contains the key value. This pointer is referred to as a row locator, and the locator's structure depends on whether the data pages are stored in a heap or as a clustered table. This is an important part of a non-clustered index's function: if it points to a heap, the row locator is a pointer to the row, but in a clustered table, the row locator is then the clustered index key.

Creating a Non-Clustered Index on a Table

There are two ways to create a non-clustered index on a table. One uses Transact-SQL script statements, and the other uses the visual interface of SQL Server Management Studio. As a database administrator, you should know how to create indexes either way.

CREATE A NON-CLUSTERED INDEX USING SQL SERVER MANAGEMENT STUDIO

GET READY. Before you begin, be sure to launch the SQL Server Management Studio application and connect to the database you want to work with. Then, follow these steps.

1. Click the plus (+) icon to the left of the Databases folder in order to expand the folder. You should now see instances of many subfolders in your main Database folder, as shown in Figure 4-13.

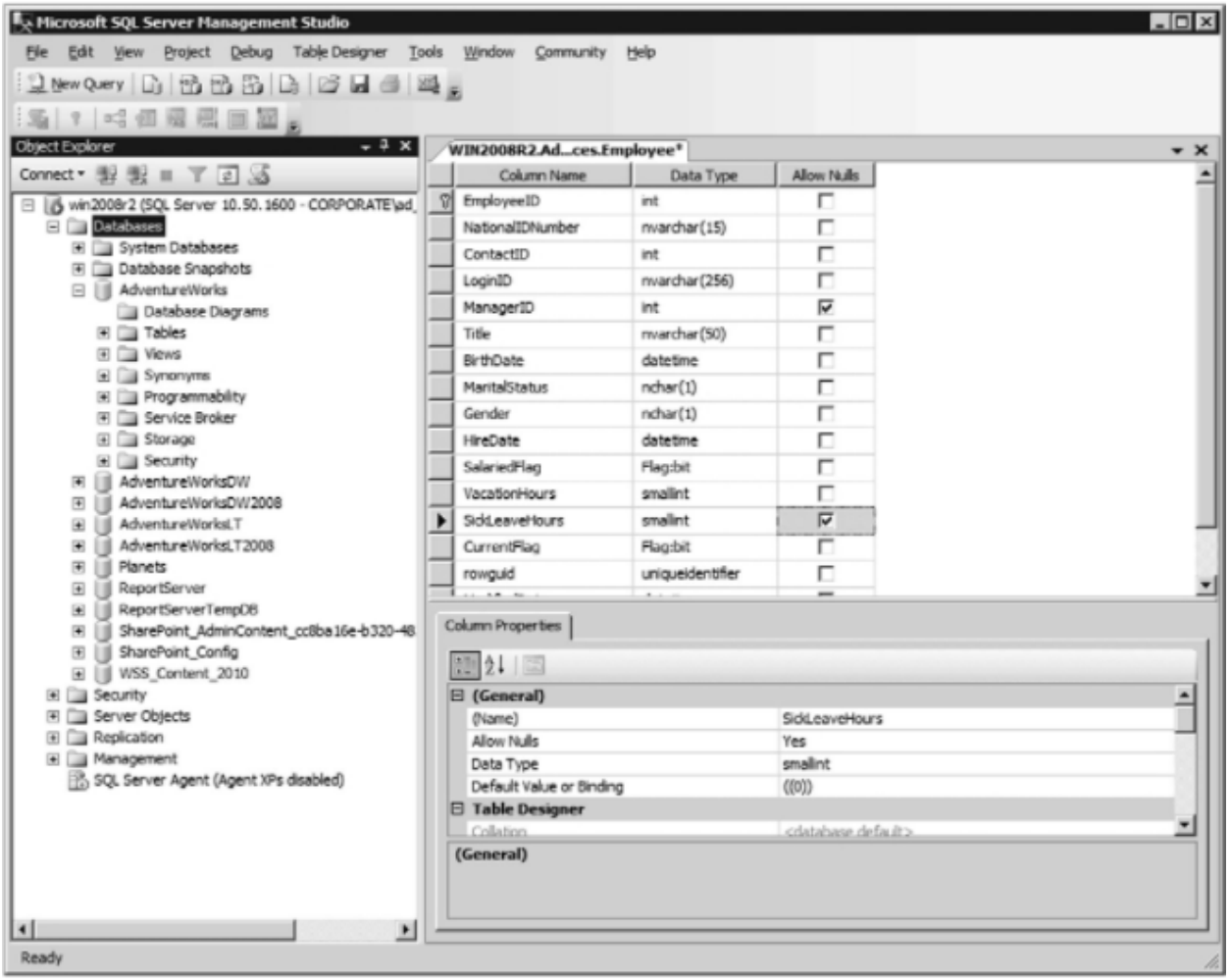

Figure 4-13 Database folder

2. Click the plus (+) icon beside the database on which you would like to create an index, as shown in Figure 4-14. You should now see many different subfolders.

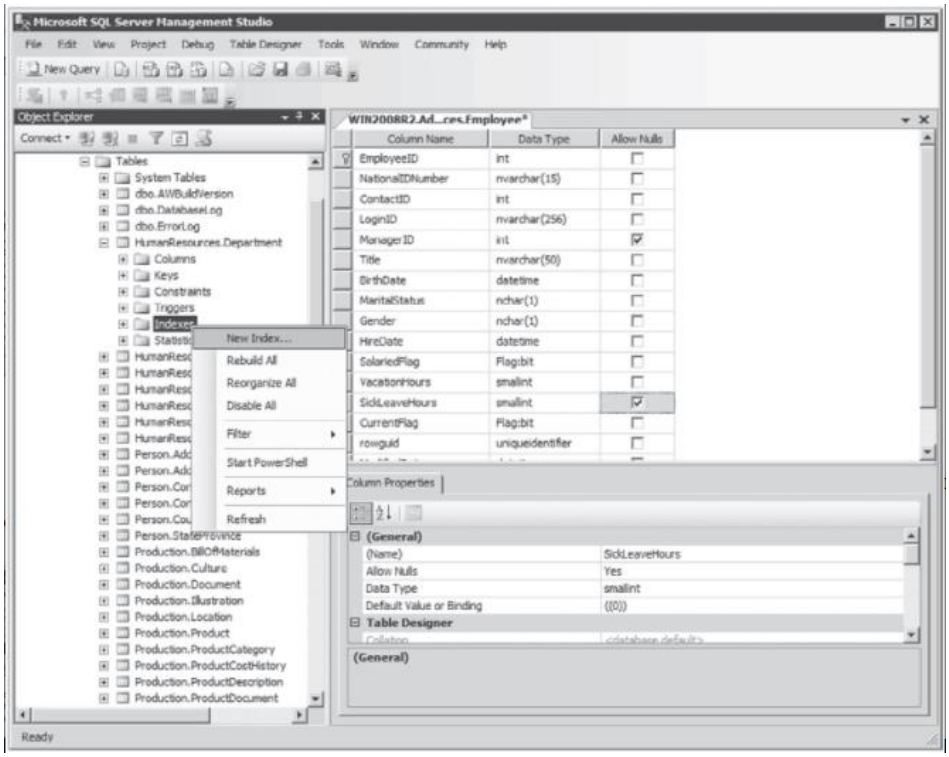

Figure 4-14 Creating an index

3. Click the plus (+) icon to the left of the Tables folder in order to expand it, as shown in Figure 4-15. You should now see a number of tables under the Tables folder.

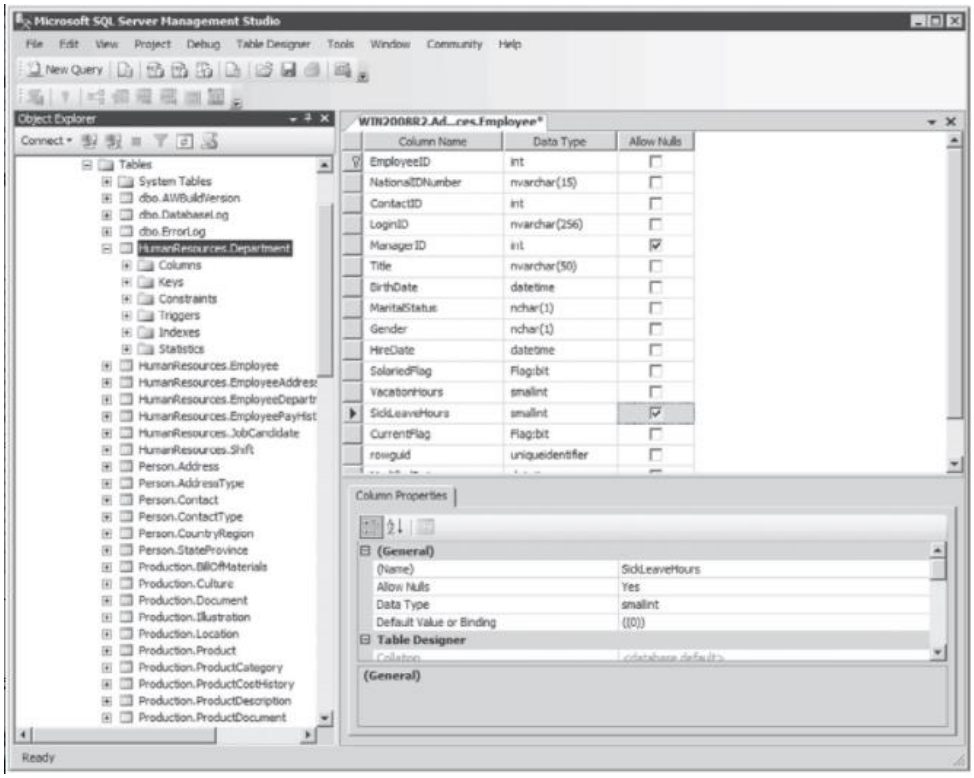

Figure 4-15 Viewing the Tables folder

4. Right-click the Indexes subfolder and select New index from the pop-up menu that appears, as shown in Figure 4-16.

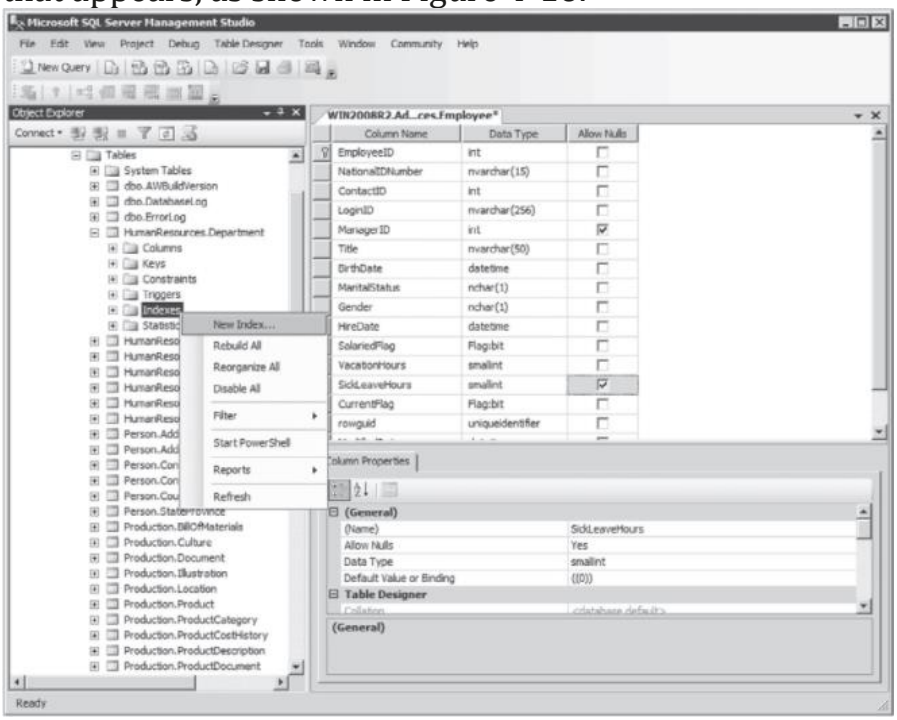

Figure 4-16 New Index menu

5. You will now see a new dialog box appear, the New Index properties box, for which you can input the desired inputs. This is where you would select whether the index you are creating is to be clustered or non-clustered (as shown in Figure 4-17).

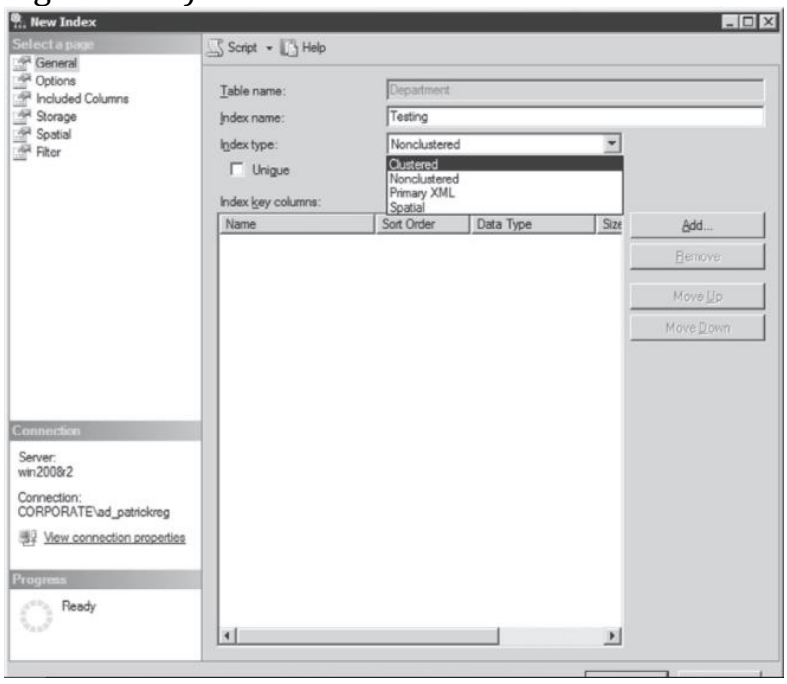

Figure 4-17 New Index property box

PAUSE. Leave the SSMS interface open for the next exercise.

For an idea of what a clustered index's Properties dialog box may look like, see Figure 4-18.

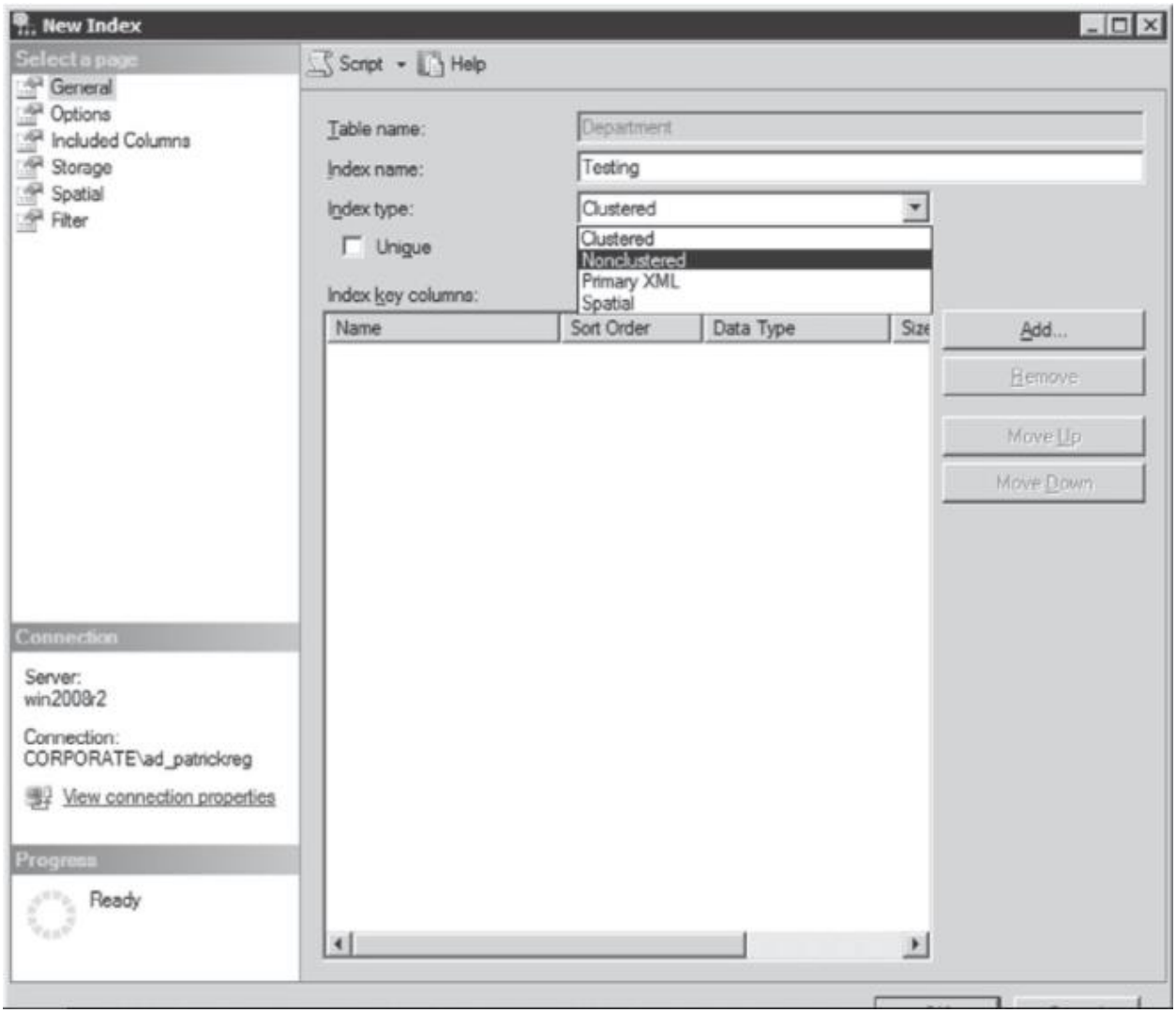

Figure 4-18 Clustered index property box

When looking at the example clustered index, note that you cannot add another index. Herein lies, as mentioned previously, the importance of ensuring you pick the right key to act as your clustered index key: This is your primary sorting index for the table, and you cannot have two clustered indexes per table.

However, on a table with a non-clustered index, you can add multiple table columns to the index key, as shown in Figure 4-19.

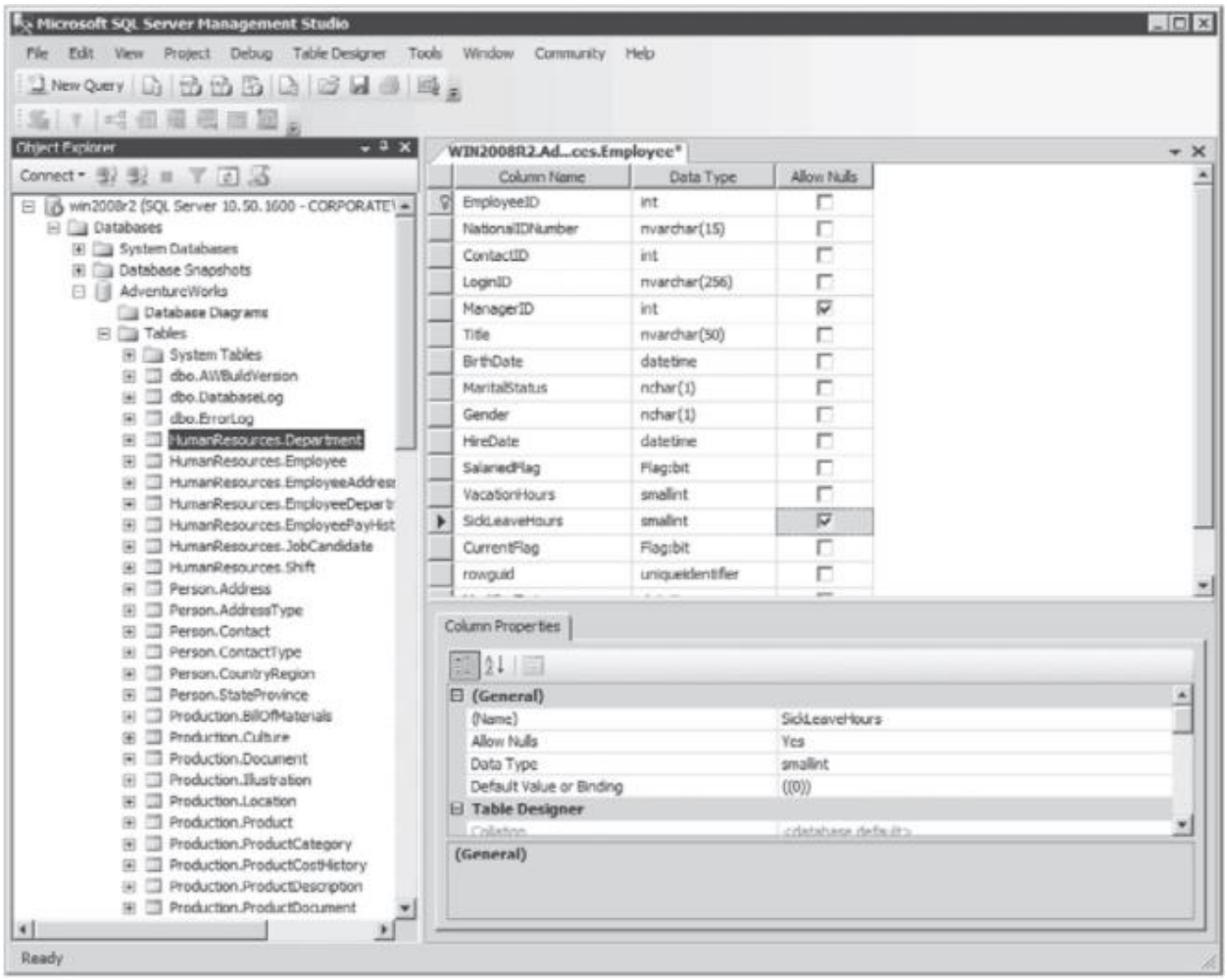

Figure 4-19 Non-clustered index property box

SKILL SUMMARY

IN THIS LESSON, YOU LEARNED THE FOLLOWING:

- Normalization, in a nutshell, is the elimination of redundant data to save space
- In the first normalized form (1NF), the data is in an entity format, which basically means that the following three conditions must be met: the table must have no duplicate records, the table must not have multi-valued attributes, and the entries in the column or attribute must be of the same data type.
- The second normal form (2NF) ensures that each attribute does in fact describe the entity.
- The third normal form (3NF) checks for transitive dependencies. A transitive dependency is similar to a partial dependency in that they both refer to attributes that are not fully dependent on a primary key.
- The fourth normal form (4NF) involves two independent attributes brought together to form a primary key along with a third attribute.
- The fifth normal form (5NF) provides the method for designing complex relationships involving multiple (usually three or more) entities.
- Three different types of constraints available within SQL Server can help you maintain database integrity: primary keys, foreign keys, and composite (unique) keys.
- A unique key constraint allows you to enforce the uniqueness property of columns, in addition to a primary key within a table.
- Perhaps the most important concept in designing any database table is that it has a primary key—an attribute or set of attributes that can be used to uniquely identify each row.
- Every table must have a primary key; without a primary key, it's not a valid table. By definition, a primary key must be unique and must have a value that is not null.
- In order to connect two tables, the primary key is replicated from the primary to secondary table, and all the key attributes duplicated from the primary table are known as the foreign key.
- A composite primary key occurs when you define more than one column as your primary key.
- The primary drawbacks to using indexes are the time it takes to build the indexes and the storage space the indexes require.
- When you begin implementing indexes, it is important to remember that each table can have only one clustered index that defines how SQL Server will sort the data stored inside it, because that data can be sorted in only one way.
- A non-clustered index contains the non-clustered index key values, and each of those keys has a pointer to a data row that contains the key value.