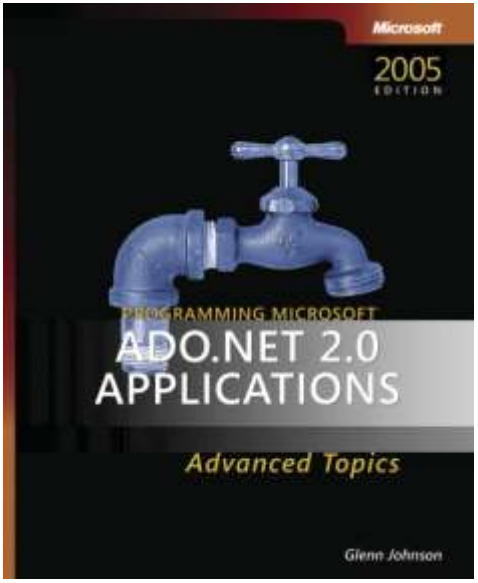

Programming Microsoft® ADO.NET 2.0 Applications
Advanced Topics

Glenn Johnson

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/7720.aspx

9780735621411
Publication Date: November 2005

Microsoft Press

Table of Contents

Chapter 5

Working with Disconnected Data

One challenge related to data access is that two or more users might need to access data in a given database simultaneously. For example, one user might be performing a task such as editing a customer record while another user is performing a different task, such as running a report of active customers. It is important to prevent two users from editing a record at the same time and to prevent reports from showing data that is half-complete.

Another data access issue is that users want the ability to access data from anywhere. We live in a mobile environment, where people want to access data on their laptops, cell phones, and PDAs. Many of these devices have limited communication capability, which adds to the challenge.

This chapter looks at the pros and cons of disconnected data access. It also explores how to implement unique row IDs as primary keys, such as Identity columns and Globally Unique Identifier (GUID) columns. These primary key types all have their advantages and problems, so we'll look at each type in detail.

Understanding Concurrency Issues

Concurrency is the ability to have multiple users accessing the database and seeing a consistent view of the data. You ensure a consistent view by locking rows, tables, and/or the database as necessary to keep users from accessing data that might be inconsistent. For example, let's say Mary is in the middle of a transaction that will transfer funds by debiting checking account A and crediting checking account B. At the same time, Joe might want to withdraw money from checking account A. What should Joe see as the current balance for account A if Mary is in the middle of her transaction? What should happen to Joe's transaction if Mary's transaction cannot be completed?

The database server addresses such situations by implementing database locking while a transaction is executing. If Mary is in the middle of a transaction that affects two checking accounts, both accounts should be locked until the transaction is complete. This means Joe's transaction should wait until Mary's transaction has been completed. The goal is to keep transactions as short as possible and to keep the lock wait time to a minimum.

Database locking allows concurrent access to the database in a connected environment where users are looking at and modifying live data, but what happens if you want to copy data to the client application, work on the data for a period of time, and then send all of the changes back

to the database? If you start a transaction that lasts until you save your changes back to the database, you might cause severe locking issues for other users who need to access the database. On the other hand, if you don't start a transaction, the data won't be locked and another user might modify the data before you have a chance to save your changes. This latter approach is the least obtrusive and more desirable approach, but you have to deal with conflicts that arise from multiple updates taking place without locking.

To deal with concurrency conflicts, the wizards in Microsoft Visual Studio implement concurrency checking by default when you create *DbDataAdapter* objects. To understand the default operation of the *DbDataAdapter*, consider the following scenario.

The database contains a table named TblBookList that contains columns for the ISBN number (primary key), BookName, and Quantity. Joe and Mary have retrieved a complete list of all rows from the table and are making changes while offline. The following sequence of events takes place.

1. Joe and Mary read data into a *DataTable*. The data is in the *CurrentVersion* and *OriginalVersion* of the *DataRow*. At this point, Joe and Mary have started with the same data.
2. Joe changes the BookName of the book whose ISBN is 123 to "Test Book 123". Mary changes the Quantity of the same book to 999.
3. Joe updates his changes to the database, which performs an update only if the original version of the *DataRow* matches the data that is currently in the database (column for column). There is a match, so the update succeeds.
4. Mary updates her changes to the database, which again performs an update only if the original version of the *DataRow* matches the data that is currently in the database (column for column). There is no match because the BookName that is currently in the database is "Test Book 123", which Joe saved, so the update fails with a *ConcurrencyException*.

Should a concurrency error be thrown in this case? After all, Joe changed one column and Mary changed a different column. This is the default behavior of the *DbDataAdapter* object's update command, which you can modify to suit your needs. If you examine the SQL update command, it looks like the following:

SQL Update Command

```
UPDATE [dbo].[TblBookList]
   SET [ISBN] = @ISBN,
       [BookName] = @BookName,
       [Quantity] = @Quantity
   WHERE (([ISBN] = @Original_ISBN)
     AND ([BookName] = @Original_BookName)
     AND ([Quantity] = @Original_Quantity))
```

The SQL update command's *WHERE* clause dictates that the update takes place only if all of the current database column values are equal to the original column values. This is probably

the safest and easiest generic approach to identifying concurrency conflicts. Also, notice that all of the columns will be set, regardless of the actual columns that have changed.

Resolving Concurrency Conflicts

When a concurrency conflict occurs, how should it be resolved? Should Joe's change override Mary's change? How you address concurrency conflicts is a business decision. Here are the main choices.

- *Prioritized on time; first update wins* Otherwise known as "first in wins." In this scenario, because Mary updated last, her changes are not persisted and Joe's changes are maintained. This approach is easy to implement because it is the default behavior of the *DataAdapter* Wizard.
- *Prioritized on time; last update wins* Otherwise known as "last in wins." In this scenario, because Mary updated last, her changes are persisted, which means Joe's changes are lost. This approach is easy to implement because it involves removing all of the extra conditions from the *WHERE* clause. In other words, the *WHERE* clause specifies only the primary key of the row to be updated.
- *Prioritized on role* Salespeople win over order entry people. If Joe is a salesperson and Mary is an order entry person, Joe has priority because it is assumed that salespeople are more knowledgeable about their customers. Thus, Joe's changes are kept and Mary's changes are rejected. This approach is a bit more difficult to implement because your application must know the role of each user. Also, if Joe and Mary have the same role, you still need to provide a fallback mechanism, such as prioritized on time.
- *Prioritized on location* Headquarters wins over branch offices. If Joe is in a branch office and Mary is at headquarters, Mary's changes are persisted and Joe's changes are overwritten. This approach is also a bit more difficult to implement because your application must know each user's location. Don't forget, if Joe and Mary are in the same location, you still need to provide a fallback mechanism, such as prioritized on time.
- *User resolves the conflict* When a conflict occurs, the user is presented with a conflict resolution screen with choices for how to resolve the conflict. Joe saves first, and at the time he saves there is no conflict. When Mary saves, a conflict is identified and she is presented with a conflict resolution screen that shows the original data that was retrieved from the database, the current data that is in the database (Joe's changes), and the current data that is in the application (Mary's changes). Mary can decide which data should be persisted.

How you should resolve concurrency conflicts depends entirely on the goals of your application and might also depend on the data you are working with. For example, with customer data the priority might be based on role, but with accounts receivable data the priority might be based on location. The automatic prioritization methods are rather straightforward; but

allowing the user to resolve the conflict can be challenging to implement. I will show an implementation of this approach later in the chapter.

Designing for Disconnected Data

Before writing your code, you must make some key design decisions that will affect your ability to work with disconnected data: how much data should be loaded, how the data will be updated when many related tables are involved, and (probably the most important decision) which type of primary key you will implement. We will look at each of these in detail, using the following scenario.

Joe is a traveling salesperson who needs to maintain a list of customers and orders for his territory. While he is on the road, he can modify the data and store it to disk. When he returns to the office, he sends his changes to the main database server and retrieves updated information as well. If any concurrency conflicts arise, Joe is prompted to select the correct data. Our example uses a simplified order entry database that contains five related tables, as shown in Figure 5-1.

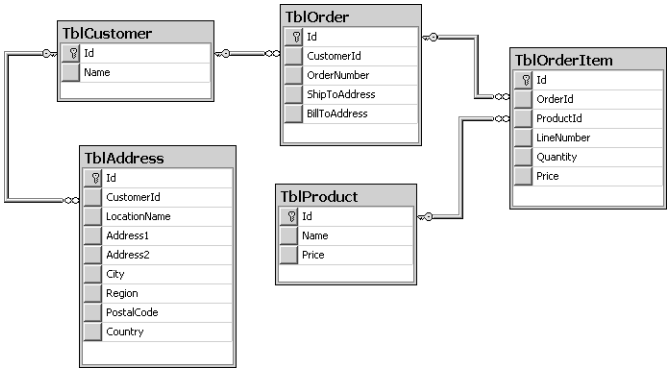

Figure 5-1 Simplified order entry database schema

What Data Should Be Loaded?

These factors affect the data that is to be loaded.

Data Selection

You should load only as much data as the user needs to work disconnected. If the user is a traveling salesperson, you might need to load his customer list and all of the information related to each customer. If you are building a Web site that uses disconnected data, you might only retrieve the data for the currently logged on customer. In almost all cases, you need to retrieve only a subset of the data in the database.

Data Size

The size of the data affects the load time, update time, and client memory requirements. Keep in mind that *DataSet* objects are memory based, so be careful about the amount of data you are retrieving. If you're not sure if you should retrieve certain data, don't retrieve it. This is a much better approach than arbitrarily retrieving data that might never be used.

Partitioning Data

It might be beneficial to break the data into multiple *DataSet* objects, based on what you think the *DataSet* object should represent. For example, you might think of a *DataSet* object as an object that contains all of the data for a single customer. In that case, the *DataSet* object might contain all five of the tables defined in Figure 5-1, but the TblCustomer and TblAddress tables contain only a single row with the information for a particular customer, and the TblOrder and TblOrderItem tables only have rows that relate to that customer.

What do you do with the TblProduct table? If the *DataSet* object is supposed to represent a complete snapshot of this customer at a point in time, you might want to include the TblProduct table but include only the product rows that relate to the TblOrderItem table. This would allow you to view all of the order information, including the products purchased. However, if you are going to allow products to be deleted from the TblProduct table, even if they have been used in orders, TblOrderItem should contain all of the data for the product being purchased, and the TblProduct table can be excluded from the customer *DataSet* object.

Similarly, if you need to be able to add more orders for different products, you must have the complete TblProduct table, so again you might want to place it into its own *DataSet* object. This would allow you to transfer product lists independently from the customer or customers. Remember that a foreign key constraint cannot be created between *DataTable* objects that are in different *DataSet* objects; however, in this case it's ok because you want to be able to delete obsolete products from the TblProduct table without being forced to delete references to them from the TblOrderItem table.

Therefore, the TblProduct table should be in its own *DataSet* object, and there should either be a *DataSet* object for each customer or one *DataSet* object that contains all of the customer data. In the scenario where Joe is a traveling salesperson, one *DataSet* object will contain the data for all of Joe's customers, and the second *DataSet* object will contain the products, as shown in Figure 5-2.

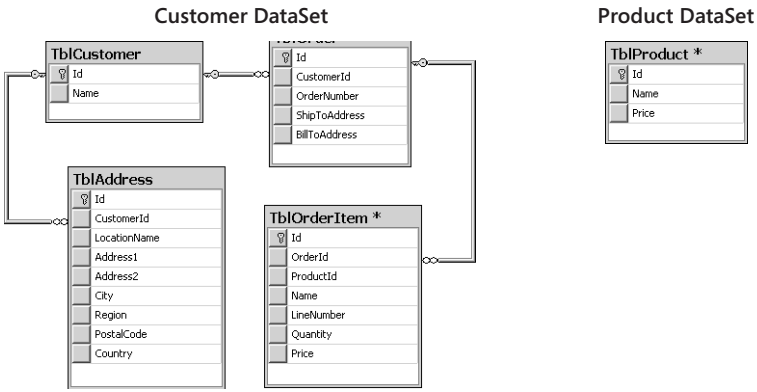

Figure 5-2 The customer data and product data should be in separate *DataSet* objects.

Choosing the Primary Key

The primary key is the column or combination of columns that uniquely defines a row. No column in the primary key can contain a null value. The primary key provides a means for the database engine to retrieve a specific row from a table in your database. The primary key is also used to enforce referential integrity. When you are working with disconnected data, you must eliminate any possibility of duplicate primary keys when multiple users are inserting data at the same time.

Intelligent, Natural Primary Keys vs. Surrogate Primary Keys

An intelligent key is a key that is based on the business data being represented. An example of an intelligent key is a Stock Keeping Unit (SKU) that is defined as a 10-character field (defined as CHAR(10) in the database). This SKU might be built as follows: the first four characters are the vendor code, the next three characters are a product type code, and the last three characters are a sequence number.

A natural key is a column or combination of columns that naturally exists in the business data and is chosen to uniquely identify records. For instance, an existing business process might define a social security number to identify a hospital patient.

Although intelligent and natural keys are different, they are both created from business-related columns that are normally viewed by the user, and the arguments for and against them are the same for the purposes of this discussion. I will refer to these keys collectively as *Intelligent-Keys*.

Surrogate primary keys are system-generated key values that have no relationship to the business data in the row, which makes them dumb keys. I'll refer to them generically as surrogate keys. One example of this kind of key is an auto-increment column where the value is set to 1, 2, 3, and so on as new rows are added. Auto-increment columns in Microsoft SQL Server are

referred to as *identity columns*. I will refer to this key as an *Identity-Key*. Another example of system-generated key values is the use of a globally unique identifier (GUID) that is set to a value that is guaranteed to be unique based on the algorithm that creates these values. I will refer to this as a *GUID-Key*.

Which type of primary key do the experts recommend? It depends on which expert you ask. Each approach has major advantages and disadvantages that you should understand before you make your choice.

Figure 5-3 shows an example of *Intelligent-Key* and surrogate primary key implementation. This example contains a table for authors and books and a join (many-to-many) table because an author can write multiple books and a book can be written by multiple authors.

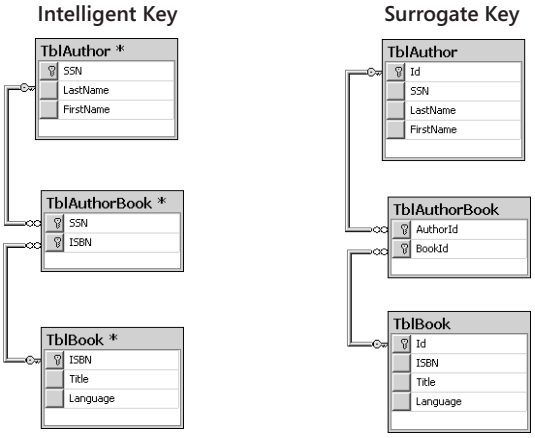

Figure 5-3 Example of intelligent and surrogate primary key implementations

Notice that the surrogate primary key implementation contains an extra Id column in the author and book tables because surrogate primary keys are system generated and have no relation to the row data. Surrogate primary keys should never be visible to the user. Here are the differences between the *Intelligent-Key* and surrogate key implementations.

Data Size Size itself is not too important, but the bandwidth that data consumes when it is transferred between the database and client is important. The surrogate implementation adds a column to each primary table. This can substantially increase data size. If the surrogate column is a GUID, the added column is 16 bytes per row. If the added column is an auto-increment column, the added column is based on the size of the numeric data type that you select (4 bytes for *int*, 8 bytes for *long*). It's common to see join tables that contain a very large quantity of rows, so be sure to consider the size difference between intelligent and surrogate keys when you analyze the overall database size difference. Also, the primary key enforces unique-

ness by creating a unique index, so be sure to consider this as well. Table 5-1 compares the sizes that result when choosing each key type.

Table 5-1 Example Primary Key Sizes

Description	*Intelligent-Key*	*Identity-Key (int)*	*GUID-Key*
1000 Authors	9 bytes/SSN = 9,000 bytes	4 bytes/int = 4,000 bytes	16 bytes/GUID = 16,000 bytes
3000 Books	10 bytes/ISBN = 30,000 bytes	4 bytes/int = 12,000 bytes	16 bytes/GUID = 48,000 bytes
10,000 AuthorBooks	19 bytes/key = 190,000 bytes	8 bytes/key = 80,000 bytes	32 bytes/key = 320,000 bytes
Subtotal	229,000 bytes	96,000 bytes	384,000 bytes
Index	229,000 bytes	96,000 bytes + 9000 SSN + 30,000 ISBN = 135,000	384,000 bytes + 9000 SSN + 30,000 ISBN = 423,000
Total Size	**458,000 bytes**	**231,000 bytes**	**807,000 bytes**

The apparent winner in this scenario is the *Identity-Key*, but remember that the maximum value of the *int* is $2^{31} - 1$ = 2,147,483,647. This should be good for most applications, but you might need to use a *long* data type for large row quantities. Notice the size calculations for the index category. The implementation of a surrogate primary key still requires a unique index on the SSN and ISBN columns to enforce uniqueness on these columns.

Key Visibility Surrogate keys are not intended to be seen by the user, whereas intelligent keys are seen and understood by the user. Your custom applications can hide surrogate keys, but database tools cannot. This means that people who use database tools must understand the use of surrogate keys. The *Intelligent-Key* is therefore the winner in this category.

Modifying Keys Primary keys are difficult to change because if you modify the key, the change must be propagated to the child tables. This is where the surrogate key shines and the intelligent key suffers. Why? Surrogate keys are not intended to be displayed to the user, so there is never a need to modify them. Intelligent keys consist of business data that is visible to the user, so you must always allow this data to change.

Surrogate *int* primary keys also need to change to ensure uniqueness (as described shortly), but surrogate GUID primary keys never need to change. This is the primary reason why I like to use surrogate GUID keys.

Quantity of Joins In some cases, you can reduce the number of joins with intelligent keys. For example, if you want to run a report showing the books by each author and containing just the author's SSN and the book's ISBN fields, you can simply query the TblAuthorBook join table when intelligent primary keys are implemented. When surrogate primary keys are implemented, you have to join the TblAuthor, TblAuthorBook, and TblBook tables to get this information. The intelligent primary keys win in this category, but consider how seldom you

need just these two columns without also needing additional information such as the author's name or book title.

SQL Complexity Intelligent primary keys are often implemented using multiple columns to achieve uniqueness. SQL queries can be much more complicated if such a compound intelligent key is involved. Although you might have more joins with surrogate keys, as described previously, surrogate keys can be easier to work with because they don't use compound keys (except possibly on join tables). If you compare the two surrogate key types, the *Identity-Key* implementation is easier to write queries for than the *GUID-Key*, but once you get familiar with some of the tricks of working with GUID data types (as described later), you'll find that the *GUID-Key* is only slightly more difficult to work with than the *Identity-Key*.

Ensuring Uniqueness When Disconnected It's essentially impossible to ensure uniqueness with intelligent keys when the user is entering data while disconnected. The problem is that someone else could enter matching information, resulting in a conflict when you attempt to send the added rows to the database server. One might argue that using surrogate keys can mask the problem, but don't forget that you still have the ability to create unique indexes on fields such as social security number or vehicle identification number, which can throw an exception if duplicate entries are added.

When surrogate *int* keys are used, the trick to managing the numbering on the primary key columns is to set the *AutoIncrement* property in the disconnected *DataSet* object to *true*, the *AutoIncrementStep* (increment) to *−1* (negative one), and the *AutoIncrementSeed* (starting value) to *−1*, which means that new rows will be added starting with a value of *−1* and will continue to increment by *−1*. The negative values are considered to be disconnected placeholders, and there is no chance of conflict with the server's identity column settings because the server assigns only positive numbers. The following SQL command shows the insertion followed immediately by querying for the inserted row. The information that is returned is used to update the placeholders (negative keys) with the value that the database created.

SQL Insert Command

```
INSERT INTO [TBLAUTHOR] ([SSN], [LastName], [FirstName])
   VALUES (@SSN, @LastName, @FirstName)
SELECT Id, SSN, LastName, FirstName FROM TblAuthor
   WHERE (Id = SCOPE_IDENTITY())
```

The *SCOPE_IDENTITY* function returns the value of the author's *Id* that was just inserted. Be careful not to use the *@@IDENTITY* function because this function returns an incorrect value if an insert trigger was fired and it inserted one or more rows into a table with an identity column.

Because the data must be retrieved from the server to update the placeholders in the disconnected data, you must consider the performance impact of updating the primary key values. What happens if you update the disconnected data key with the value that was created at the server? All of the child data must be updated to reflect the change as well; you can do this by

enabling cascading updates on the relationships. This creates another performance hit, especially for large *DataSet* objects.

When surrogate GUID primary keys are used, there is no need to change the key once it has been set. The main problem is setting the value. The following code snippet shows how to initialize the GUID.

Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
   For Each dt As DataTable In salesSurrogateGuidKeyDs.Tables
      If (not dt.Columns("Id") Is Nothing) Then
         AddHandler dt.TableNewRow, addressof InitializeGuid
      End If
   Next
End Sub
Private Sub InitializeGuid(ByVal sender As Object, _
      ByVal e As DataTableNewRowEventArgs)
   If (TypeOf e.Row("Id") Is DBNull) Then
      e.Row("Id") = Guid.NewGuid()
   End If
End Sub
```

C#

```
public Form1() //constructor
{
   InitializeComponent();
   foreach (DataTable dt in sales_SurrogateGuidKeyDs.Tables)
   {
      if(dt.Columns["Id"] != null)
         dt.TableNewRow += new DataTableNewRowEventHandler(InitializeGuid);
   }
}
private void InitializeGuid(object sender, DataTableNewRowEventArgs e)
{
   if(e.Row["Id"] is DBNull)
      e.Row["Id"] = Guid.NewGuid();
}
```

Because *TblAuthor* and *TblBook* have primary keys with the same name ("Id"), a method called *InitializeGuid* is coded for creating new GUIDs and the *TableNewRow* event of these tables is wired to call this method. It's usually a good idea to give all surrogate primary keys the same name, as I did here. There is no need to create cascading relationships with this implementation. This is why surrogate GUID primary keys win in this category.

Migrating Data to Other Databases Migrating data from one database to another when the *Identity-Key* is implemented requires some work. Imagine that your tables have keys with values 1 through *n* and these values are also placed in foreign keys throughout the database. You want to take that data and merge it into a database with data that uses the same numbers. To solve this problem, you need to renumber all of the Identity columns.

With the *GUID-Key* implementation, migrating data is a simple matter of copying the data from one database to the other, which means the *GUID-Key* wins in this category.

And the Winner Is...

I just finished a very large project where the *GUID-Key* was implemented, and I have also worked on large projects using *Intelligent-Key* and *Identity-Key* implementations. Table 5-2 summarizes how each approach rates in a number of categories that are important in terms of performance, size, and ease of use. Based on my experience with these approaches, I assigned scores on a weight of 0 to 100 percent, where a weight of 100 is most important. Next I assigned a first-place (1), second-place (0.5), and third-place (0) score to the key types and multiplied that score by the weight to obtain a weighted score for each item.

Table 5-2 Final Scores Based on Categories and Weights

	Weighted Scores		
Category and Weight	***Intelligent-Key***	***Identity-Key***	***GUID-Key***
Data Size = 25%	12.5%	25%	0%
Key Visibility = 5%	5%	2.5%	0%
Modifying Keys = 20%	0%	10%	20%
Quantity of Joins = 5%	5%	2.5%	2.5%
SQL Complexity = 5%	0%	5%	2.5%
Ensuring Uniqueness = 25%	0%	12.5%	25%
Migration = 15%	7.5%	0%	15%
Total = 100%	**30%**	**57.5%**	**65%**

Based on my weighting, the *GUID-Key* best satisfies the greatest number of the most important categories. If you feel differently about any of the items, try modifying the weights to see whether you get a different result. Note that none of these primary key implementations gets a perfect, 100 percent rating. My general feeling is that the *GUID-Key* implementation is the best approach for disconnected data applications. Is there a place for the *Intelligent-Key* implementation? Yes, it might be the best approach for data warehouse applications because data warehouse applications are typically designed to provide high performance read-only access with minimum joins. Since the data is read-only, there is little concern regarding key modification. With a bit of tweaking, Table 5-3 provides the scores based on different weights.

Table 5-3 Data Warehouse Scores Based on Categories and Weights

	Weighted Scores		
Category and Weight	***Intelligent-Key***	***Identity-Key***	***GUID-Key***
Data Size = 25%	12.5%	25%	0%
Key Visibility = 10%	10%	0%	0%
Modifying Keys = 5%	0%	2.5%	5%
Quantity of Joins = 25%	25%	0%	0%

Table 5-3 Data Warehouse Scores Based on Categories and Weights

	Weighted Scores		
Category and Weight	***Intelligent-Key***	***Identity-Key***	***GUID-Key***
SQL Complexity = 10%	10%	5%	5%
Ensuring Uniqueness = 5%	0%	0%	5%
Migration = 20%	10%	0%	20%
Total = 100%	**67.5%**	**32.5%**	**35%**

Use these tables as guidelines, and be sure to consider any additional categories that your project may have.

Who's Afraid of the Big, Bad GUID?

Many people find the GUID to be quite intimidating when they attempt to use the *GUID-Key*. GUIDs might be big, but they aren't so bad. A few tips can help.

Copying/Pasting GUIDs

When you are debugging, you can select code that contains a GUID and IntelliSense will show the GUID (Figure 5-4). You can select the GUID value and copy it to the clipboard. Paste this value into a query window, replacing the curly braces with single quotation marks, as shown in the following SQL statement.

SQL Query Using a GUID

```
SELECT Id, SSN, LastName, FirstName
FROM   TblAuthor
WHERE  (Id = 'cbc8c64c-6ba6-4bec-baef-4c0e50e8b251')
```

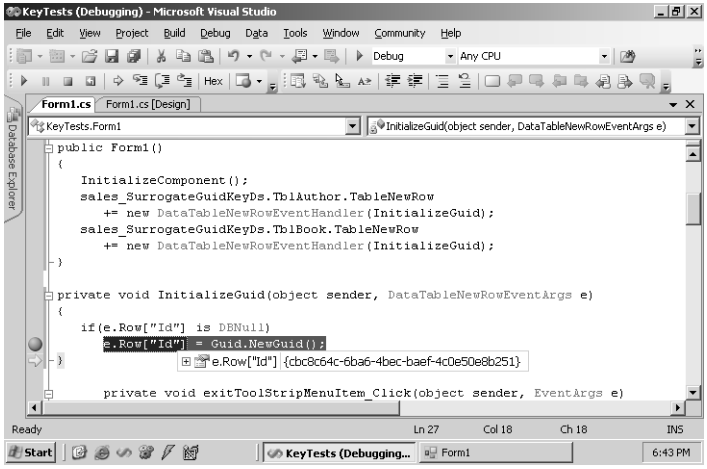

Figure 5-4 Using IntelliSense to copy a GUID for pasting into query tools

Using the Same Name for the Primary Key Column on Non-Join Tables

I strongly recommend that you use the same name (such as *Id*) for the primary key column of all of your non-join tables. This makes it easier to code stored procedures that work with GUIDs. Also, make it the first column in every table to help users understand the purpose of this field.

Finding a GUID in the Database

Depending on your database design, you might face situations where you are looking for a GUID in a foreign key column but have no idea where the data is for that GUID. An example of this is when you have an Exclusive-OR relationship, as shown in Figure 5-5. This type of relationship is typical in an object-oriented environment—you might have a book class with various child classes such as EBook, PaperBack, and HardCover. These classes will have fields that are not common to each other, so you must choose to either create a single table with lots of null column values, or create a separate table for each child class as shown in Figure 5-5.

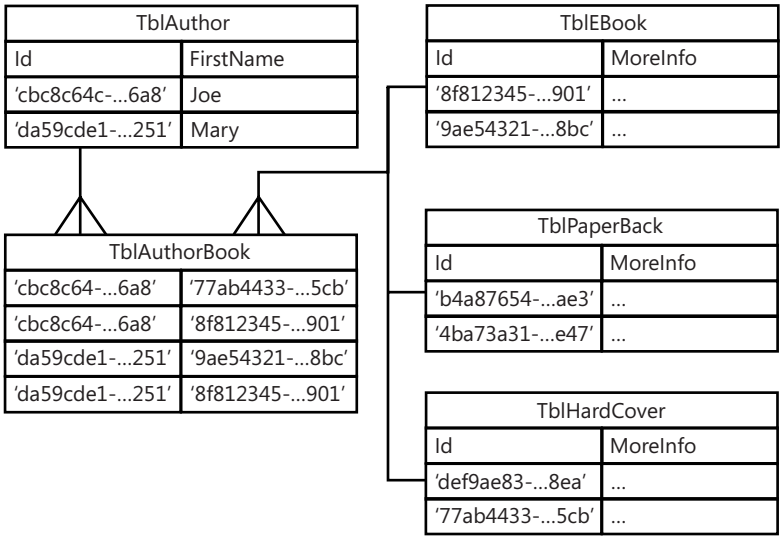

Figure 5-5 An example of an Exclusive-OR relationship

If you are trying to find out which table contains a specific GUID, you can use the following stored procedure to locate the table that has this GUID as a primary key value.

SQL uspGetDataForId

```
CREATE PROCEDURE dbo.uspGetDataForId
(
      @id uniqueidentifier
)
AS
```

```
SET NOCOUNT ON
--NOTE: This proc assumes that all user tables have 'Tbl' prefix
--Usage: in Query Analyser, type the following without the '--'
--exec uspGetDataForId '78257ec8-c8f9-4d35-a636-d58d8a67c3d4'
DECLARE @tbl varchar(2000)
DECLARE @sql varchar(2000)
IF OBJECT_ID('tempdb..#idTable') IS NOT NULL DROP TABLE #idTable
CREATE TABLE #idTable (
      Id uniqueidentifier,
      Count INT,
      TableName varchar(2000)
)
DECLARE tables_cursor CURSOR
   FOR SELECT TABLE_NAME FROM information_schema.Tables
      WHERE substring (TABLE_NAME,1,3)='Tbl'
OPEN tables_cursor
FETCH NEXT FROM tables_cursor INTO @tbl
WHILE @@FETCH_STATUS = 0
BEGIN
   IF EXISTS (SELECT * FROM information_schema.columns
              WHERE table_name=@tbl AND Column_Name='Id')
   BEGIN
      SET @sql = 'INSERT INTO #idTable SELECT id as ''Id'', '
         + 'count(*) as ''Count'',''' + @tbl +''' as ''TableName'' FROM '
         + @tbl + ' WHERE ID=''' + CONVERT(varchar(2000),@id)
         + ''' group by Id'
      EXEC(@sql)
   END
   FETCH NEXT FROM tables_cursor INTO @tbl
END
CLOSE tables_cursor
DEALLOCATE tables_cursor
SELECT Id, TableName FROM #idTable WHERE Count > 0
```

Note that this stored procedure relies on all primary key columns being named *Id* and the user tables having a *Tbl* prefix. This stored procedure does not attempt to find a GUID in any other column, but if you need to find all usages for a GUID, read on.

Finding All Usages of a GUID in the Database

You often want to find all usages of a GUID. The following SQL script enumerates all of the user tables with a *Tbl* prefix and queries all columns with a *uniqueidentifier* data type.

SQL uspGetUsagesForId

```
CREATE   PROCEDURE dbo.uspGetUsagesForId
(
      @id uniqueidentifier
)
AS
--NOTE: This proc assumes that all user tables have 'Tbl' prefix
--Usage: in Query Analyser, type the following without the '--'
--exec uspGetUsagesForId '78257ec8-c8f9-4d35-a636-d58d8a67c3d4'
SET NOCOUNT ON
```

```
DECLARE @tbl varchar(2000)
DECLARE @sql varchar(2000)
DECLARE @counter integer

IF OBJECT_ID('tempdb..#guidTable') IS NOT NULL DROP TABLE #guidTable
CREATE TABLE #guidTable (
      Id uniqueidentifier,
      Count INT,
      TableName varchar(2000)
)

DECLARE tables_cursor CURSOR
   FOR SELECT TABLE_NAME FROM information_schema.Tables
       WHERE SUBSTRING(TABLE_NAME,1,3)='Tbl'
OPEN tables_cursor
FETCH NEXT FROM tables_cursor INTO @tbl
WHILE @@FETCH_STATUS = 0
BEGIN
      SET @sql = 'INSERT INTO #guidTable SELECT id as ''Id'', '
            + 'count(*) as ''Count'',''' + @tbl +''' as ''TableName'' FROM '
            + @tbl + ' WHERE ' + dbo.fnGetGuidWhereClause(@tbl, @id)
            + ' GROUP BY ID'
      EXEC(@sql)
      SELECT @counter = COUNT(*) FROM #guidTable
      IF @counter > 0
      BEGIN
            SET @sql = 'SELECT ''' + @tbl + ''' as TABLE_NAME, * FROM ' + @tbl
                  + ' WHERE ' + dbo.fnGetGuidWhereClause(@tbl, @id)
            EXEC(@sql)
      END
      DELETE FROM #guidTable
FETCH NEXT FROM tables_cursor INTO @tbl
END
CLOSE tables_cursor
DEALLOCATE tables_cursor
```

If you think about the benefits of these stored procedure tricks, you will realize that you can't accomplish these tricks and get the same results with the other primary key implementations.

Building a Conflict Resolution Screen

The rest of this chapter focuses on building a conflict resolution screen that allows the user to resolve a conflict by selecting the current user value, the original database value, the current database value, or a typed-in value. To drive this conflict resolution screen, the application will display only the customer names and will allow the user to update the list. If a *DbConcurrency* exception occurs, the conflict resolution screen will be displayed.

Creating the Project

First we create the project for demonstrating conflict resolution when a *DbConcurrency* exception occurs after an update of the database server from disconnected data.

1. Create a Microsoft Windows application using the appropriate programming language.
2. Add a *MenuStrip* control to the form. Add the following menu items.

   MenuItem List

   ```
   &File
      &Sync With Database
      E&xit
   &Concurrency
      &Resolve Concurrency Errors
   ```

3. Add a status bar with a single status label called *status*.
4. Add a new database file to the application by right-clicking the project, choosing Add, choosing New Item, and then choosing SQL Database. Name the database *Customer.mdf,* and then click the Add button. This launches the Data Source Configuration Wizard. You haven't created any tables yet, so click Finished to add the empty customer *DataSet* to your project.
5. Add the tables as shown earlier in Figure 5-1 or, at a minimum, add a table called TblCustomer that has a column named Id that is a *uniqueidentifier* primary key and a column named Name that is a *varchar(50)* data type. Neither column should allow a null value. (You can edit the tables by right-clicking the database file and clicking Open.)
6. Open the DataSources window by choosing Data and then choosing Show Data Sources. The *CustomerDataSet* class is visible in this window, but it needs to be updated to show tables that you have added. Right-click the CustomerDataSet and choose Configure DataSet With Wizard, which will launch the Data Source Configuration Wizard. Select all the tables you have created in the database, as shown in Figure 5-6.

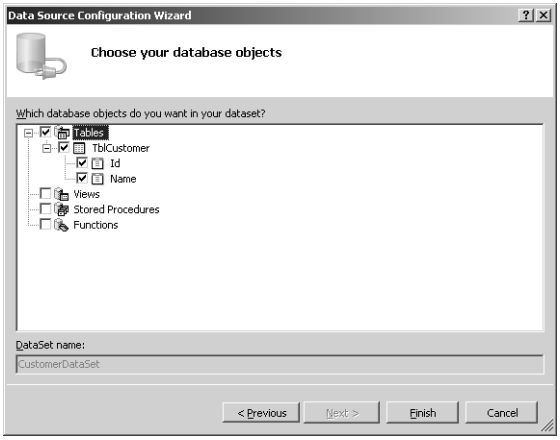

Figure 5-6 Adding tables to the *CustomerDataSet* using the Data Source Configuration Wizard

7. In the Data Sources window, drag and drop TblCustomer onto the form. This adds *TblCustomerDataGridView* to the form. Set the *TblCustomerDataGridView* object's *Dock* property to *Fill*. Set the *AutoSizeMode* of the Name column to *Fill*. Also note that Visual Studio automatically added *CustomerDataSet*, *TblCustomerBindingSource*, *TblCustomerTableAdapter*, and *TblCustomerBindingNavigator* to the designer tray (Figure 5-7).

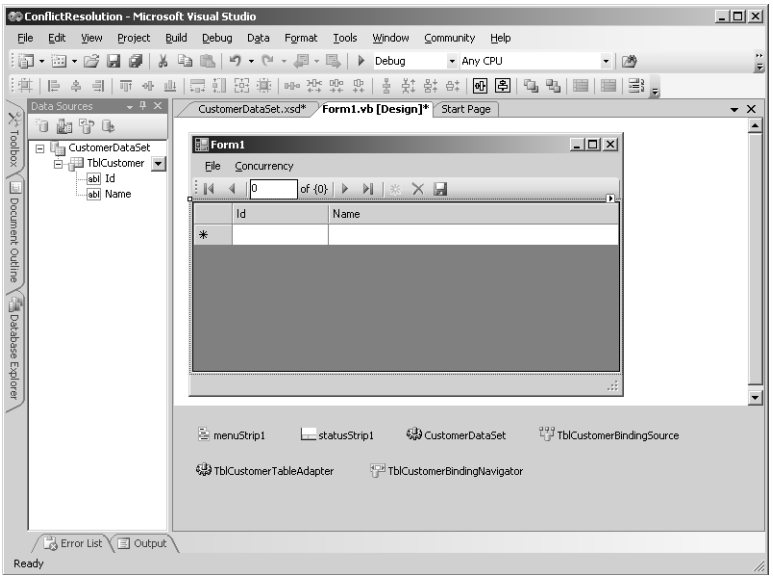

Figure 5-7 Adding *TblCustomer* to the form adds the necessary objects to the designer tray automatically.

Extending the Typed *DataSet* (*CustomerDataSet*) Class

We must extend this *CustomerDataSet* class so that there is one place to automatically generate new GUIDs for our primary key columns. Then we can forget about generating new GUIDs in the application.

1. Right-click the CustomerDataSet.xsd file and choose View Code. This adds a new file called CustomerDataSetExtension.vb or CustomerDataSetExtension.cs under the CustomerDataSet.xsd node.

2. Add the following code to this file.

Visual Basic

```
Imports System.Data
Partial Public Class CustomerDataSet
   Private createDefaultGuidForNewRows As Boolean = False
   Public Sub CreateDefaultGuids()
      If (createDefaultGuidForNewRows) Then Exit Sub
      createDefaultGuidForNewRows = True
      For Each dt As DataTable In Me.Tables
         If (Not dt.Columns("Id") Is Nothing) Then
```

```
            AddHandler dt.TableNewRow, AddressOf TableNewRow
         End If
      Next
   End Sub
   Private Sub TableNewRow(ByVal sender As Object, _
         ByVal e As DataTableNewRowEventArgs)
      If (TypeOf e.Row("Id") Is DBNull) Then
         e.Row("Id") = Guid.NewGuid()
      End If
   End Sub
End Class
```

C#

```
using System;
using System.Data;
namespace ConflictResolution
{
   public partial class CustomerDataSet
   {
      private bool createDefaultGuids = false;
      public void CreateDefaultGuids()
      {
         if (createDefaultGuids) return;
         createDefaultGuids = true;
         foreach (DataTable dt in this.Tables)
         {
            if (dt.Columns["Id"] != null)
               dt.TableNewRow += new DataTableNewRowEventHandler(TableNewRow);
         }
      }
      private void TableNewRow(object sender, DataTableNewRowEventArgs e)
      {
         if (e.Row["Id"] is DBNull)
            e.Row["Id"] = Guid.NewGuid();
      }
   }
}
```

3. This code needs to be called once from the application, so add the following code into the form object's *Load* method. You can then run the application and add new customers to the table.

Visual Basic

```
CustomerDataSet.CreateDefaultGuids()
```

C#

```
customerDataSet.CreateDefaultGuids();
```

Extending the *TableAdapter* (*TblCustomerTableAdapter*) Class to Expose the *ContinueUpdateOnError* Property

The *DataAdapter* object contains a property called *ContinueUpdateOnError*. If this property is set to *true*, multi-row updates continue when concurrency errors occur. The *TableAdapter*

objects expose only the minimally required properties, and this application requires the *ContinueUpdateOnError* property of the underlying *TableAdapter* object's *DataAdapter* to be set to *true*. This means the *TblCustomerTableAdapter* must be extended.

1. In the file called CustomerDataSetExtension.vb or CustomerDataSetExtension.cs that you just added when you extended the *CustomerDataSet* class, add the following code to expose the *ContinueUpdateOnError* property. (Unfortunately, this code cannot easily be added in a single place for all *TableAdapter* objects. If you have additional *TableAdapter* objects, add code to extend each of them in a similar fashion.)

   Visual Basic

   ```
   Namespace CustomerDataSetTableAdapters
      Partial Public Class TblCustomerTableAdapter
         Public Property ContinueUpdateOnError() As Boolean
            Get
               Return Adapter.ContinueUpdateOnError
            End Get
            Set(ByVal value As Boolean)
               Adapter.ContinueUpdateOnError = Value
            End Set
         End Property
      End Class
   End Namespace
   ```

   C#

   ```
   namespace ConflictResolution.CustomerDataSetTableAdapters
   {
      public partial class TblCustomerTableAdapter
      {
         public bool ContinueUpdateOnError
         {
            get { return Adapter.ContinueUpdateOnError; }
            set { Adapter.ContinueUpdateOnError = value; }
         }
      }
   }
   ```

2. Add the following code to the form's *Load* method to set the *ContinueUpdateOnError* property of each *TableAdapter* object to *true*.

   Visual Basic

   ```
   TblCustomerTableAdapter.ContinueUpdateOnError=True
   ```

   C#

   ```
   tblCustomerTableAdapter.ContinueUpdateOnError = true;
   ```

Synchronizing the Disconnected *DataSet* with the Database Server

The application can now be run, but without any synchronization of the disconnected *CustomerDataSet* with the database server. The following step adds synchronization to your code.

- Place the following code into the Synchronize With Database menu item. This code attempts to send all changes back to the database server. If it is successful, the customer table is refilled from the database server to capture any changes that other users have made.

Visual Basic

```
Private Sub syncwithDatabaseToolStripMenuItem_Click( _
      ByVal sender As System.Object, _
      ByVal e As System.EventArgs) _
      Handles syncwithDatabaseToolStripMenuItem.Click
   TblCustomerTableAdapter.Update(CustomerDataSet.TblCustomer)
   If (CustomerDataSet.HasErrors) Then
      status.Text = "Partial synchronization with concurrency errors."
   Else
      'get current database data
      tblCustomerTableAdapter.Fill(CustomerDataSet.TblCustomer)
      status.Text = "Database synchronized."
   End If
   TblCustomerDataGridView.Refresh()
End Sub
```

C#

```
private void syncwithDatabaseToolStripMenuItem_Click(
      object sender, EventArgs e)
{
   tblCustomerTableAdapter.Update(customerDataSet.TblCustomer);
   if (customerDataSet.HasErrors)
   {
      status.Text = "Partial synchronization with concurrency errors.";
   }
   else
   {
      //get current database data
      this.tblCustomerTableAdapter.Fill(customerDataSet.TblCustomer);
      status.Text = "Database synchronized.";
   }
   tblCustomerDataGridView.Refresh();
}
```

Creating the Conflict Resolution Screen

The primary purpose of the conflict resolution screen is to allow you to select the final data values when a concurrency error occurs. It displays your current values, the original database values, and the current database values. The screen is somewhat generic, and you will probably want to tweak it to suit your needs. For reference, the form that you create is shown in Figure 5-8.

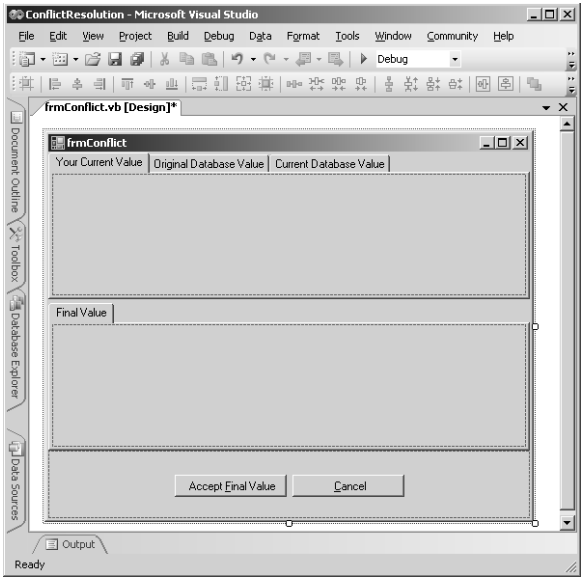

Figure 5-8 The completed conflict resolution screen

1. Add a new form to your project called frmConflict.vb or frmConflict.cs.
2. Add a *Panel* control to the form, and set its *Dock* property to *Bottom*.
3. Add two *Button* controls to the *Panel* control. Assign the *Name* properties, *btnAccept* and *btnCancel*, and set their *Text* property to *"Accept Final Value"* and *"Cancel"*. Set the *Anchor* property on both buttons to *Top*.
4. Add a *SplitContainer* control to the form. Set its *Dock* property to *Fill*, and set its *Orientation* property to *Horizontal*.
5. Add a *TabControl* to the top of *the SplitContainer*. Create three tab pages, with the *Name* property set to *tabCurrent*, *tabOriginal*, and *tabCurrentDb*. Set the *Text* property to *Your Current Value*, *Original Database Value*, and *Current Database Value*.
6. Add a *TabControl* to the bottom of the *SplitContainer*. Create a single tab page, with the *Name* property set to *tabFinal*. Set the *Text* property to *Final Value*.
7. The conflict resolution screen is created by passing two *DataRow* objects into a new constructor. The first *DataRow* object contains your current data values in the *Current DataRowVersion* and the original database data values in the *Original DataRowVersion*. The second *DataRow* object initially contains the current database data value in the *Original* and *Current DataRowVersion*. The *Current DataRowVersion* will contain the final result of your conflict resolution. These *DataRow* objects are stored in properties. Here is the code to accomplish this.

Visual Basic

```
Public Sub New(ByVal currentDataRow As DataRow, _
      ByVal currentDatabaseDataRow As DataRow)
   MyBase.New()
```

```
      InitializeComponent()
      Me.CurrentDataRow = currentDataRow
      FinalDatabaseDataRow = currentDatabaseDataRow
   End Sub
   Public Property FinalDatabaseDataRow() As DataRow
      Get
         Return m_finalDatabaseDataRow
      End Get
      Set(ByVal value As DataRow)
         m_finalDatabaseDataRow = value
      End Set
   End Property
   Private m_finalDatabaseDataRow As DataRow = Nothing
   Public Property CurrentDataRow() As DataRow
      Get
         Return m_currentDataRow
      End Get
      Set(ByVal value As DataRow)
         m_currentDataRow = value
      End Set
   End Property
   Private m_currentDataRow As DataRow = Nothing
```

C#

```
public frmConflict(DataRow currentDataRow, DataRow currentDatabaseDataRow)
{
   InitializeComponent();
   CurrentDataRow = currentDataRow;
   FinalDatabaseDataRow = currentDatabaseDataRow;
}
public DataRow FinalDatabaseDataRow
{
   get { return m_finalDatabaseDataRow; }
   set { m_finalDatabaseDataRow = value; }
}
private DataRow m_finalDatabaseDataRow = null;
public DataRow CurrentDataRow
{
   get { return m_currentDataRow; }
   set { m_currentDataRow = value; }
}
private DataRow m_currentDataRow = null;
```

8. Create the *PopulateTab* and *CopyToFinal* methods. For each column in the *DataRow* object, the *PopulateTab* populates a tab page with a *Label* that shows the column name, a *TextBox* that shows the value of the column based on the *DataRowVersion*, and a *Button* that lets you copy the selected value to the final value by executing the *CopyToFinal* method. Here is the code for these two methods.

Visual Basic

```
Public Sub PopulateTab(ByVal tab As TabPage, ByVal dataRow As DataRow, _
      ByVal dataRowVersion As DataRowVersion, ByVal m_ReadOnly As Boolean)
   Const verticalSpacing As Integer = 30
   Const labelWidth As Integer = 50
```

```
   Const horizontalSpacing As Integer = 10
   Const buttonWidth As Integer = 100
   Const buttonHeight As Integer = 20
   For col As Integer = 0 To dataRow.ItemArray.Length - 1
      Dim val As Object = dataRow(col, dataRowVersion)
      Dim label As New Label()
      tab.Controls.Add(label)
      label.Text = dataRow.Table.Columns(col).ColumnName
      label.Top = (col + 1) * verticalSpacing
      label.Left = horizontalSpacing
      label.Width = labelWidth
      label.Visible = True
      Dim textBox As New TextBox()
      tab.Controls.Add(textBox)
      textBox.Text = val.ToString()
      textBox.Top = (col + 1) * verticalSpacing
      textBox.Left = (horizontalSpacing * 2) + labelWidth
      textBox.Width = tab.Width - textBox.Left _
         - buttonWidth - (horizontalSpacing * 2)
      textBox.Name = tab.Name + label.Text
      textBox.ReadOnly = m_ReadOnly
      textBox.Visible = True
      textBox.Anchor = AnchorStyles.Left _
         Or AnchorStyles.Top Or AnchorStyles.Right
      If (tab.Name = "tabFinal") Then Continue For
      Dim btn As New Button()
      tab.Controls.Add(btn)
      btn.Text = "Copy to Final"
      btn.Left = textBox.Left + textBox.Width _
         + horizontalSpacing
      btn.Top = (col + 1) * verticalSpacing
      btn.Height = buttonHeight
      btn.Visible = True
   btn.Anchor = AnchorStyles.Top OR AnchorStyles.Right
      AddHandler btn.Click, AddressOf CopyToFinal
      Dim propertyBag As New ArrayList()
      propertyBag.Add(dataRow.Table.Columns(col))
      propertyBag.Add(textBox)
      btn.Tag = propertyBag
   Next
End Sub
Private Sub CopyToFinal(ByVal sender As Object, ByVal e As EventArgs)
   Dim btn As Button = CType(sender, Button)
   Dim propertyBag As ArrayList = CType(btn.Tag, ArrayList)
   Dim dc As DataColumn = CType(propertyBag(0), DataColumn)
   Dim textBox As TextBox = CType(propertyBag(1), TextBox)
   tabFinal.Controls(tabFinal.Name + dc.ColumnName).Text = textBox.Text
End Sub
```

C#

```
public void PopulateTab(TabPage tab, DataRow dataRow,
      DataRowVersion dataRowVersion, bool readOnly)
{
   const int verticalSpacing = 30;
   const int labelWidth = 50;
```

```
      const int horizontalSpacing = 10;
      const int buttonWidth = 100;
      const int buttonHeight = 20;
      for (int col = 0; col < dataRow.ItemArray.Length; col++)
      {
         object val = dataRow[col, dataRowVersion];
         Label label = new Label();
         tab.Controls.Add(label);
         label.Text = dataRow.Table.Columns[col].ColumnName;
         label.Top = (col + 1) * verticalSpacing;
         label.Left = horizontalSpacing;
         label.Width = labelWidth;
         label.Visible = true;
         TextBox textBox = new TextBox();
         tab.Controls.Add(textBox);
         textBox.Text = val.ToString();
         textBox.Top = (col + 1) * verticalSpacing;
         textBox.Left = (horizontalSpacing * 2) + labelWidth;
         textBox.Width = tab.Width - textBox.Left
            - buttonWidth - (horizontalSpacing * 2);
         textBox.Name = tab.Name + label.Text;
         textBox.ReadOnly = readOnly;
         textBox.Visible = true;
         textBox.Anchor = AnchorStyles.Left | AnchorStyles.Top
            | AnchorStyles.Right;
         if (tab.Name == "tabFinal") continue;
         Button btn = new Button();
         tab.Controls.Add(btn);
         btn.Text = "Copy to Final";
         btn.Left = textBox.Left + textBox.Width + horizontalSpacing;
         btn.Top = (col + 1) * verticalSpacing;
         btn.Height = buttonHeight;
         btn.Visible = true;
         btn.Anchor = AnchorStyles.Top | AnchorStyles.Right;
         btn.Click += new EventHandler(CopyToFinal);
         ArrayList propertyBag = new ArrayList();
         propertyBag.Add(dataRow.Table.Columns[col]);
         propertyBag.Add(textBox);
         btn.Tag = propertyBag;
      }
   }
```

9. Add code to call the *PopulateTab* method for each tab page when the form is loaded. The code is as follows.

Visual Basic

```
Private Sub frmConflict_Load( ByVal sender As System.Object,  _
      ByVal e As System.EventArgs) Handles MyBase.Load
   PopulateTab(tabCurrent, CurrentDataRow, _
      DataRowVersion.Current, true)
   PopulateTab(tabOriginal, CurrentDataRow, _
      DataRowVersion.Original, true)
   PopulateTab(tabCurrentDb, FinalDatabaseDataRow, _
      DataRowVersion.Original, true)
   PopulateTab(tabFinal, FinalDatabaseDataRow, _
```

```
      DataRowVersion.Current, false)
End Sub
```

C#

```
private void frmConflict_Load(object sender, EventArgs e)
{
   PopulateTab(tabCurrent, CurrentDataRow,
      DataRowVersion.Current, true);
   PopulateTab(tabOriginal, CurrentDataRow,
      DataRowVersion.Original, true);
   PopulateTab(tabCurrentDb, FinalDatabaseDataRow,
      DataRowVersion.Original, true);
   PopulateTab(tabFinal, FinalDatabaseDataRow,
      DataRowVersion.Current, false);
}
```

10. Add code to collect the final data values from the *TextBox* objects in *tabFinal* and place the values into the *FinalDatabaseRow*, which will be retrieved by the calling form. This code will be placed into the *Click* event handler of *btnAccept* as follows.

Visual Basic

```
Private Sub btnAccept_Click( ByVal sender As System.Object,  _
      ByVal e As System.EventArgs) Handles btnAccept.Click
   for each dc as DataColumn  in FinalDatabaseDataRow.Table.Columns
      FinalDatabaseDataRow(dc) = _
         tabFinal.Controls(tabFinal.Name + dc.ColumnName).Text
   next
End Sub
```

C#

```
private void btnAccept_Click(object sender, EventArgs e)
{
   foreach (DataColumn dc in FinalDatabaseDataRow.Table.Columns)
   {
      FinalDatabaseDataRow[dc] =
         tabFinal.Controls[tabFinal.Name + dc.ColumnName].Text;
   }
}
```

11. The final step is to configure the buttons. Set the *DialogResult* of *btnAccept* to *OK*, and set the *DialogResult* of *btnCancel* to *Cancel*. On the form's property screen, set the *AcceptButton* property to *btnAccept* and set the *CancelButton* property to *btnCancel*.

Calling the Conflict Resolution Screen

The conflict resolution screen deals with conflicts only on a *DataRow*-by-*DataRow* basis. This means you must loop through your concurrency errors, displaying the conflict resolution screen for each conflict.

- In the Resolve Concurrency Errors menu item, add the following code to retrieve the latest data from the database and call the conflict resolution screen for each conflict that has occurred since the last time you synchronized with the database.

Visual Basic

```
Private Sub resolveConcurrencyErrorsToolStripMenuItem_Click( _
      ByVal sender As System.Object,  ByVal e As System.EventArgs) _
      Handles resolveConcurrencyErrorsToolStripMenuItem.Click
   if (customerDataSet.TblCustomer.HasErrors) then
      'get data refresh with most current
      Dim refreshCustomer as new CustomerDataSet()
      tblCustomerTableAdapter.Fill(refreshCustomer.TblCustomer)
      'loop through the errors
      for each dr as DataRow in customerDataSet.TblCustomer.GetErrors()
         Dim currentDb as DataRow = _
            refreshCustomer.TblCustomer.Rows.Find(dr("Id"))
         using conflict as new frmConflict(dr, currentDb)
            if (conflict.ShowDialog(Me) = DialogResult.OK) then
               dr.ClearErrors()
               tblCustomerTableAdapter.Update(conflict.FinalDatabaseDataRow)
               customerDataSet.TblCustomer.LoadDataRow( _
                  conflict.FinalDatabaseDataRow.ItemArray, _
                  LoadOption.OverwriteChanges)
               status.Text = "Single row updated."
            else
               status.Text = "Single row update cancelled."
            end if
         end using
      next
      tblCustomerDataGridView.Refresh()
   end if
End Sub
```

C#

```
private void resolveConcurrencyErrorsToolStripMenuItem_Click(
   object sender, EventArgs e)
{
   if (customerDataSet.TblCustomer.HasErrors)
   {
      //get data refresh with most current
      CustomerDataSet refreshCustomer = new CustomerDataSet();
      this.tblCustomerTableAdapter.Fill(refreshCustomer.TblCustomer);
      //loop through the errors
      foreach (DataRow dr in customerDataSet.TblCustomer.GetErrors())
      {
         DataRow currentDb = refreshCustomer.TblCustomer.Rows.Find(dr["Id"]);
         using (frmConflict conflict = new frmConflict(dr, currentDb))
         {
            if (conflict.ShowDialog(this) == DialogResult.OK)
            {
               dr.ClearErrors();
               tblCustomerTableAdapter.Update(conflict.FinalDatabaseDataRow);
               customerDataSet.TblCustomer.LoadDataRow(
                  conflict.FinalDatabaseDataRow.ItemArray,
                  LoadOption.OverwriteChanges);
               status.Text = "Single row updated.";
            }
            else
            {
```

```
                    status.Text = "Single row update cancelled.";
                }
            }
        }
        tblCustomerDataGridView.Refresh();
    }
}
```

Correcting Concurrency Errors with the Conflict Resolution Screen

When it's time to test the conflict resolution screen, you must have two instances of the application running. When each instance starts, it loads a copy of the data from the database. If changes are made in both instances, the first instance that synchronizes with the database will succeed and the second instance will fail for each *DataRow* object that was also updated by the first instance.

Follow these steps to create and resolve concurrency errors.

1. Start two instances of the application. (Navigate to the ConflictResolution.exe file and double-click it twice, or right-click the project in the Solution Explorer window and choose Debug | Start New Instance twice.) You will see the contents of the *TblCustomer* table.
2. The first time the application is executed there is no data, so add data to both instances, as shown in Figure 5-9.

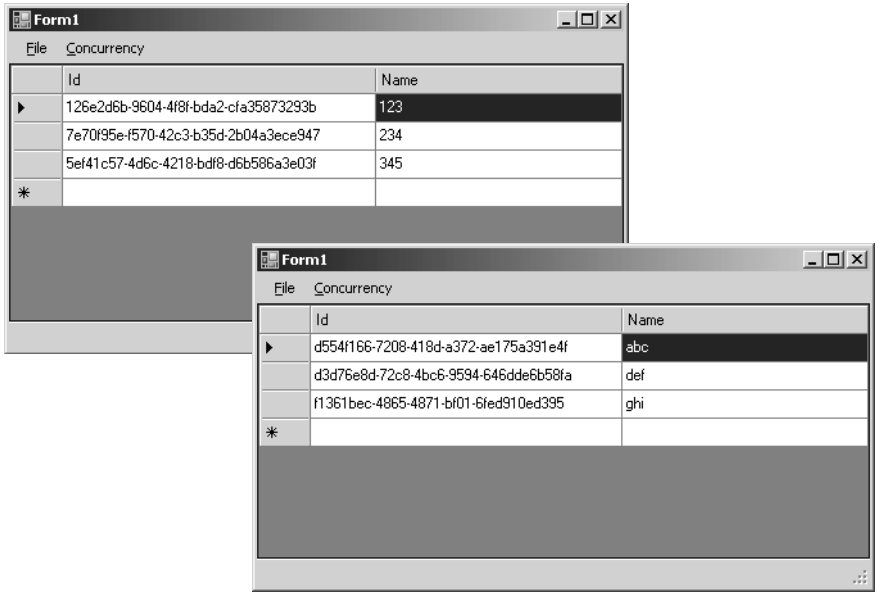

Figure 5-9 Different data has been entered into each instance of the application.

3. Synchronize with the database. When the first instance is synchronized, the only noticeable change is that the status bar displays a message indicating that the data is synchronized. When the second instance is synchronized, the same message is displayed, but the grid displays the data that was input into the first instance. Notice that both instances can synchronize with the new data because there are no conflicts. If the first instance is synchronized again, it will contain the data from the second instance, as shown in Figure 5-10.

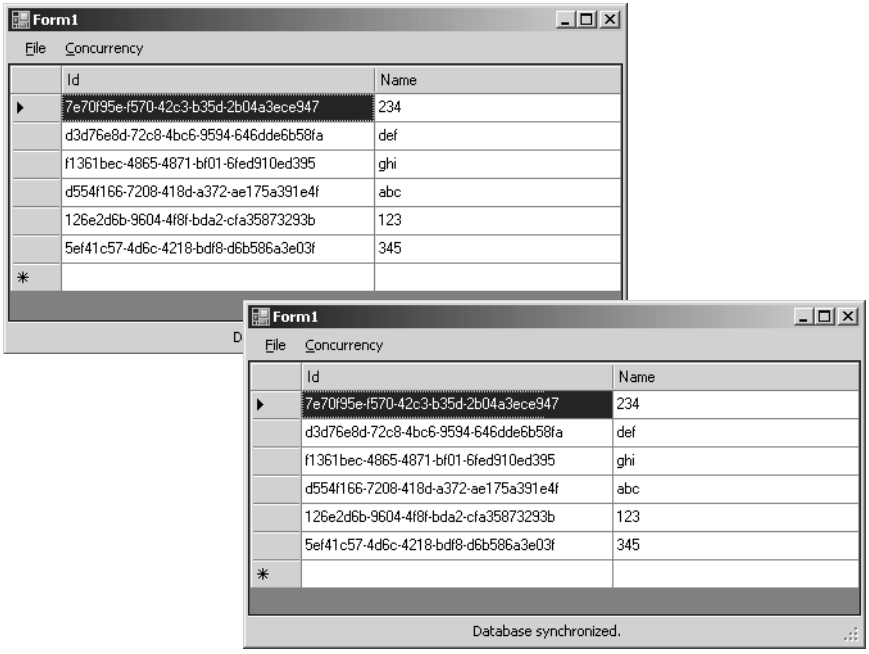

Figure 5-10 The two instances are synchronized with the database.

4. Make changes to the same name in both instances. For example, you can change the name *234* to *Joe* in one instance and to *Mary* in the other instance. Synchronize the instance with *Joe* as the name, and the synchronization will succeed. Next, synchronize the instance with *Mary* as the name; a concurrency error occurs because the data in the database was changed between the time this instance was previously synchronized and this attempt to synchronize. The grid displays the error, as shown in Figure 5-11.

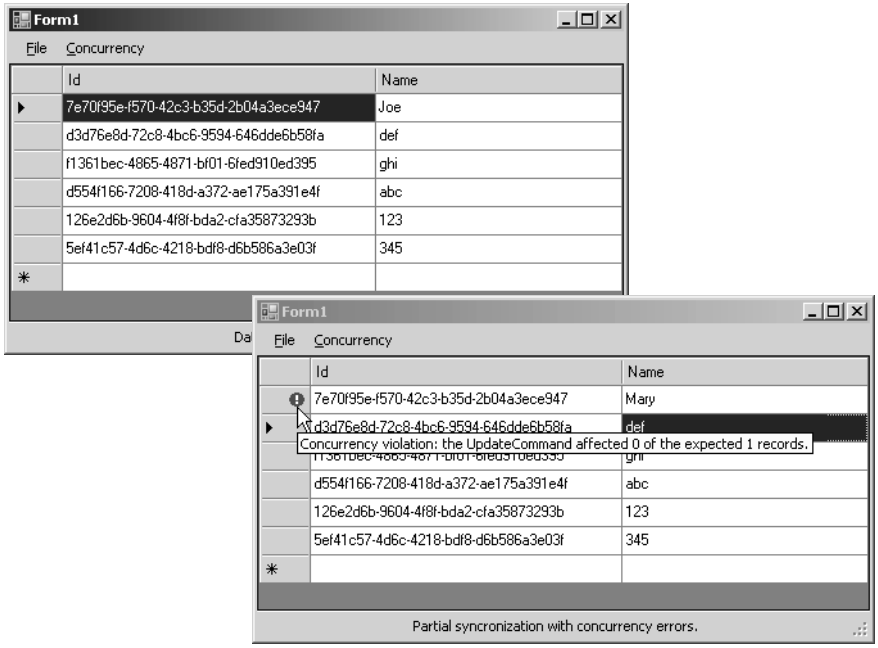

Figure 5-11 The concurrency error is displayed in the grid because the underlying data changed since we read the data.

5. Finally, test the conflict resolution screen. Click the Resolve Concurrency Errors menu item to launch the conflict resolution screen, as shown in Figure 5-12. The tab pages are populated with data. Move between the tab pages to see the current disconnected data values, the original database values, and the current database values. Note that the final data values are displayed at the bottom, and notice the button beside each *TextBox* that allows you to copy the value in the *TextBox* into the final data value.

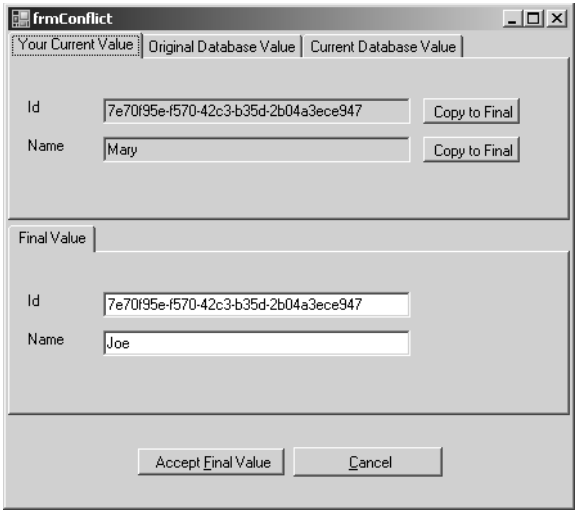

Figure 5-12 The conflict resolution screen allows you to resolve a concurrency error by selecting the current disconnected data values, the original database values, or the current database values.

Building a Better Conflict Resolution Screen

The conflict resolution screen was designed to let you generically resolve conflicts from concurrency errors. Most of the code can be reused in your application, but you might want to make some changes to make the screen more appealing to your users and to further encapsulate your code. Here are some changes to consider.

- *Pass two* DataSet *objects to the conflict resolution form instead of two* DataRow *objects.* The first *DataSet* object contains all of your data, along with the concurrency errors. The second *DataSet* object contains the data that is currently in the database. You can program the conflict resolution screen to loop through the errors so that users don't have to examine each error individually.
- *Hide surrogate keys.* Surrogate keys have no relationship with business data, so there is no need to show them. If you use *GUID-Keys*, you can simply hide all columns that are the GUID data type.
- *Provide Copy All buttons.* On each tab page, add a Copy All button that copies all of the data on the tab page to the final data values.

Summary

You must consider two important issues when you work with disconnected data: primary key implementation and concurrency error (conflict) resolution. The primary key is the column or combination of columns that uniquely defines a row. Intelligent and natural keys both contain business-related columns that are normally viewed by the user, but surrogate primary keys are system-generated key values that have no relationship to the business data in the row. The *GUID-Key* implementation is considered the best approach for disconnected data applications.

Concurrency is the ability for multiple users to access the database and be presented with a consistent view of the data. On connected database applications, locking rows and tables enforces concurrency, but disconnected database applications allow concurrency errors, so the application must be written to resolve conflicting data. How you resolve conflicts depends entirely on your business needs. You can choose to implement these strategies: *first update wins*, *last update wins*, *prioritized on role*, *prioritized on location*, or *user resolves the conflict.*