The Right Keys are Primary

Both the content and the way of creating primary keys have changed over the years. With VFP 8 and later, setting up surrogate primary keys is a breeze.

Tamar E. Granor, Ph.D.

FoxPro is a relational database, that is, data is stored in multiple tables with fields that establish relationships between those tables. In order to establish those relationships, there must be a way to uniquely identify each record in a table. The field or fields that link one table to another are called keys. Over the years, the best practices for creating those keys have changed, as have the tools for doing so in VFP.

When I started working with FoxBase+, it wasn't unusual to use keys that spanned several fields. For example, a customer might be linked to its orders using the company name and phone number, as in Listing 1.

Listing 1. In the early days of Xbase, it wasn't unusual to link two tables together using multiple fields.

```
USE Customers IN 0
USE Orders IN 0 ORDER Customer
SELECT Customers
SET RELATION TO UPPER(Company + PhoneNum) ;
  INTO Orders
```

However, working with this type of relationship is cumbersome, and even before VFP was introduced, using a single field to link two tables emerged as a best practice. When VFP 3 introduced the ability to designate a field as a primary key, with its uniqueness enforced by the database engine, the move to single field keys accelerated.

In the example above, rather than using company name plus phone number to identify a customer, you add a customer id field of some sort and propagate that to the child tables. Using this strategy, the code to open and relate the tables looks more like Listing 2.

Listing 2. Using a single field rather than multiple fields to relate two tables is a best practice.

```
USE Customers IN 0
USE Orders IN 0 ORDER CustomerID
SELECT Customers
SET RELATION TO CustomerID INTO Orders
```

Few would argue with the idea of using a single identifying field as a *primary key*, the principal identifier of a record, in a table like Customers or Products. We're accustomed to the idea of assigning an ID to real-world objects like customers and products. (Consider, for example, the UPC codes that adorn millions of products or the social security number that ostensibly uniquely identifies each American taxpayer.)

However, even tables that don't represent real-world objects with obvious identifiers should have primary keys as well. So, in the example above, not only should the Customers and Orders table have an ID field, so should the LineItems table that lists the individual items on each order.

You might wonder why you can't just identify each line item by its order number and line number, or by the order number plus the product number. The second example is easier to dispense with. If a detail line is identified by order number plus product number, each product can be included on only one line of an order. While that might be the norm, enforcing such a rule might impose a hardship on an application's users.

But why not identify a detail line by the order number and line number? This leads us to the question of surrogate keys.

Using surrogate keys

Although the topic is somewhat controversial, most VFP experts recommend using what are known as *surrogate keys*. That is, give the table a field whose only purpose is to uniquely identify the record. The field has no other meaning, and normally, is never seen by users. VFP 8 made using surrogate keys easy, with its addition of an auto-incrementing integer data type.

There are many pros, as well as a few cons, to working with surrogate keys. The biggest strength of surrogate keys is that they never need to change. When you use a meaningful data item as a primary key (sometimes called an *intelligent key*), you run the risk that the data might change later. For example, consider a Customer table using the name of the customer (company) as a primary key. If the customer company changes its name, not only must the Customer table be updated, but all the records for that customer in other tables (such as Order) must be modified as well. With a surrogate primary key, only the actual customer record needs to change.

Another big win for surrogate keys comes in ensuring uniqueness. Since surrogate keys are assigned internally and never seen by the user, it's easy to make sure to assign unique values. With intelligent keys, there's a chance that the data designated as the primary key won't be unique. Obviously, given names and company names aren't unique. Even supposedly unique identifiers like social security numbers turn out to be duplicated occasionally, whether through error or fraud. Using a made-up "company code" type field is better, but then you're relying on the user to create unique identifiers. Even if you establish a rule for creating the identifiers (say, the first 10 characters of the company name plus the zip code), sooner or later, you're likely to run into a repeated value and have to find another way to generate the code.

Related to ensuring uniqueness is handling gaps. With surrogate keys, gaps in the sequence don't matter. No one ever sees them. If you use a meaningful field like order number, some users will want to make sure every value in the sequence is used. In multi-user applications, that turns out to be surprisingly difficult.

Because surrogate keys are always just one field, not multiple fields, writing joins that involve them is simple.

Finally on the positive side, surrogate keys are often smaller than intelligent keys. While storage isn't a big issue any more, anything that saves space without giving up something in return is still a plus.

There are two real downsides to using surrogate keys. The first is that they can actually make queries more complex. If a child table contains only a foreign key to the parent table, rather than actual data from the parent, every query that needs data from both must perform a join.

In addition, using surrogate keys makes it difficult to re-link records when data has been damaged in some way. Along these lines, it's also more difficult to simply look at a table and understand what it contains.

Overall, though, the positives of surrogate keys far outweigh the negatives, and using surrogate primary keys is a best practice for VFP.

Generating surrogate keys

There are several ways to generate surrogate primary keys in VFP, but for VFP 8 and later, the auto-incrementing integer field type is the easiest way.

Prior to VFP 8, there was no fully automatic way to generate surrogate keys. You had to write some code. In FoxPro 2.x and earlier, the process was completely manual. You had to be sure to call the right code to populate the primary key field.

When VFP 3 introduced default values for tables in a database, the standard approach was to set the default value for the primary key field to call the appropriate routine.

As for the code itself, there are several approaches to generating the key value.

The simplest way to do so was widely used, but is unsafe for multi-user applications. That is to find the highest key in use and add one to it, with code like Listing 3.

Listing 3. This function to generate a surrogate key doesn't work for multi-user applications or those that allow multiple copies of a data entry form to be used simultaneously.

```
PROCEDURE GetID
CALCULATE MAX(iID) TO nLastID
nNewID = m.nLastID + 1
RETURN m.nNewID
```

The problem with this code is that two users working in the same table can generate the same key value. Even if the record is saved quickly after generating the key, there's still a chance for another user to call this code in the time between the first user's call and the new record being saved. Except in very special circumstances, calculating the maximum is not the right choice.

The TasTrade database that comes with VFP demonstrates a safer way to generate a surrogate key. This technique uses a table to track the key values for each table. The primary key field of some of the tables in the database (for example, Orders) has a default value set to NewID(). The stored procedures for the database contain the NewID function, which is shown in Listing 4. Until VFP 8, many, many developers used some version of this approach.

Listing 4. The NewID method from TasTrade looks up the next surrogate key value in a table, then updates the table.

```
FUNCTION NewID(tcAlias)
  LOCAL lcAlias, ;
        lcID, ;
        lcOldReprocess, ;
        lnOldArea

  lnOldArea = SELECT()

  IF PARAMETERS() < 1
    lcAlias = UPPER(ALIAS())
  ELSE
    lcAlias = UPPER(tcAlias)
  ENDIF

  lcID = ""
  lcOldReprocess = SET('REPROCESS')

  *-- Lock until user presses Esc
  SET REPROCESS TO AUTOMATIC

  IF !USED("SETUP")
    USE tastrade!setup IN 0
  ENDIF
  SELECT setup
```

```
  IF SEEK(lcAlias, "setup", "key_name")
    IF RLOCK()
      lcID = setup.value
      REPLACE setup.value WITH ;
              STR(VAL(ALLT(lcID)) + 1, ;
              LEN(setup.value))
      UNLOCK
    ENDIF
  ENDIF

  SELECT (lnOldArea)
  SET REPROCESS TO lcOldReprocess

  RETURN lcID
ENDFUNC
```

Some people prefer to use GUIDs as surrogate keys. The term "GUID" stands for "Globally Unique ID" and refers to an identifier that is extremely unlikely to ever be duplicated anywhere. While that may be overkill for many applications, creating GUIDs is easy. For distributed applications that need to allow users in disconnected locations to create records, GUIDs can be a good choice. The code in Listing 5, copied from the VFP Wiki (at http://fox.wikis.com/wc.dll?Wiki~GUID~VFP) generates and returns a GUID.

Listing 5. To generate GUIDs, you call an API function. Note that the return value isn't particularly readable since it uses the complete ASCII character set.

```
Function makeid
  Local encodedid
  Declare Integer CoCreateGuid In OLE32.Dll
String @encodedid
  encodedid = Space(16)
  If CoCreateGuid(@encodedid)<>0
    Error "Cannot Create ID"
  Endif
  Return m.encodedid
```

Letting VFP generate surrogate keys

VFP 8 introduced the autoincrementing integer data type. Using this type, you can set a field up so that each new record is assigned the next value. There's no need to specify a default value or write any code. Autoincrementing integers can be used in free tables, as well as in database tables.

To set a field to use this type, choose "Integer (AutoInc)" from the Type dropdown in the Table Designer. You can also create these fields programmatically using CREATE TABLE, as in Listing 6.

Listing 6. Use the AUTOINC keyword to make an integer field autoincrementing.

```
CREATE TABLE MyTable ;
   (iID I AUTOINC, cOther C(10))
```

Adding an autoincrementing integer field to an existing table is a little trickier. These fields are read-only, so you have to add the field, populate it, and then make it autoincrement, as in Listing 7.

Listing 7. To add an autoincrementing integer field to an existing table, add an integer field, populate it, and then change it to autoincrement.

```
ALTER TABLE MyTable ADD iID I
REPLACE ALL iID WITH RECNO() IN MyTable
nNextVal = RECCOUNT("MyTable") + 1
ALTER TABLE MyTable ;
   ALTER COLUMN iID I AUTOINC ;
   NEXTVALUE m.nNextVal
```

Because autoincrementing integer fields are read-only, you may have to write some processing code differently. For example, SQL INSERT commands must include a list of fields, so that you're not attempting to put data in the autoincrementing field. Similarly, other commands that attempt to update all fields of a record have to be used with caution.

Propagating autoincrementing keys

When working with parent-child tables, you need to know the primary key of the parent in order to insert records into the child table. This can be a particular problem when working with views. VFP 9 adds the GetAutoIncValue() function, which returns the last autoincrement value generated in the current (or specified) data session. (Note that you can't specify which table it applies to, so you do need to exercise caution with this function.) You can use the function to retrieve the primary key for a record after you save it, so you can use that key as a foreign key in the corresponding child records.

Listing 8 shows an example. It's based on a database containing two tables, Person and Phone. Each person record can have one or more phone records. The code shows the addition of a person with two phones. While code exactly like this is unlikely to appear in an application, it shows the basic structure that might be used in a data entry form. The database, tables and code are included in this month's downloads.

Listing 8. GetAutoIncValue() lets you retrieve the most recent auto-increment value within a data session. It lets you find the primary key of a parent record, so it can be inserted into a child record.

```
* Add a person and some phone numbers,
* using views.

LOCAL iPersonID

SET MULTILOCKS ON

OPEN DATABASE People

USE v_Person IN 0 NODATA
USE v_Phone IN 0 NODATA

CURSORSETPROP("Buffering", 5, "v_Person")
CURSORSETPROP("Buffering", 5, "v_Phone")

INSERT INTO v_Person (cFirst, cLast) ;
   VALUES ("John", "Smith")
```

```
INSERT INTO v_Phone (cPhone) ;
   VALUES ("215-555-1234")
INSERT INTO v_Phone (cPhone) ;
   VALUES ("800-555-9832")

BEGIN TRANSACTION
IF TABLEUPDATE(.f., .f., "v_Person")
   iPersonID = GETAUTOINCVALUE()
   REPLACE ALL iPersonID ;
       WITH m.iPersonID IN v_Phone
   IF TABLEUPDATE(.t., .f., "v_Phone")
      END TRANSACTION
   ELSE
      ROLLBACK
   ENDIF
ELSE
   ROLLBACK
ENDIF

RETURN
```

Choose the right keys

When designing new applications, use surrogate keys. If all users will be connected (that is, working on the same network), VFP's autoincrementing integers provide the easiest way to set up surrogate keys. With the addition of GetAutoIncVal() in VFP 9, the biggest problem with these keys has been resolved. For applications that run in multiple locations and need to create a common database, GUIDs provide a good solution.

In older applications, you may find no primary keys, or only meaningful primary keys. If you're making significant changes in such an application, adding surrogate primary keys is likely to be your best path.

Whichever approach you choose, using surrogate primary keys will make your applications more stable and easier to maintain.

What should I cover?

As I mentioned in my last column, I'd love to hear from readers about what topics to cover in this column. Please email your suggestions to tamar@thegranors.com.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of ten books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2, coming out this year. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com