

May, 2007

Retro-fitting Primary Keys

When an application doesn't use surrogate primary keys, you can add them easily with these functions

By Tamar E. Granor, Technical Editor

I've been working for the last couple of years on revisions to an existing application. This system was designed by someone with little database experience and as a result, the data structures are not properly normalized. Most of the primary keys are either meaningful data or require multiple fields. In addition, rather than pointing back to the original tables with foreign keys, much data is duplicated throughout the application.

Now that the most pressing problems in this application have been dealt with, we're starting to work on the database problems. I'm giving each table a surrogate primary key (using an auto-incrementing Integer field) and replacing the multiple fields that refer to another table with a single foreign key field that references the new primary key.

Of course, when we deploy this update, we need to preserve existing data. So as I modify the tables and the code that depends on them, I'm also developing code we can run after installation to update the customer's data to the new structure without data loss. Although I started out writing the necessary code for each specific table, I quickly found that a couple of generic functions could simplify my job considerably.

The problem

To make the problem clear, let's look at an example. Figure 1 shows a very simple database to represent task management for a company. There are five tables. Dept is just a list of departments, indicating the department manager. Employee is the list of employees, and indicates which department an employee is assigned to. JobType is a list of job categories. Task is a list of things to be done. Assigned is a many-to-many linking table between Employee and Task.

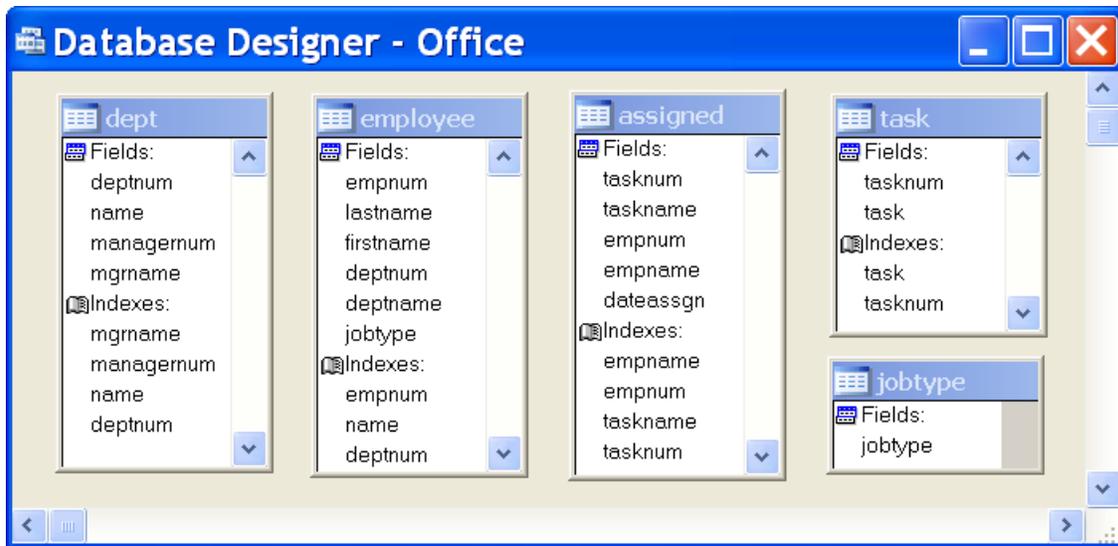


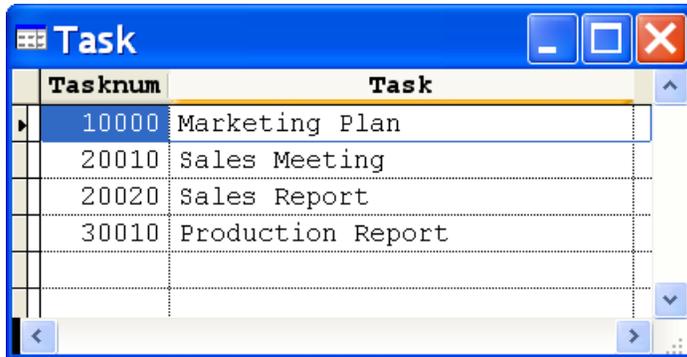
Figure 1. Poor database design—Some of the primary keys in this database are meaningful. In addition, data other than the primary key is duplicated between tables.

Figure 2 demonstrates one of the main problems. It shows the Employee table. Note that not only is the employee's department number included, but so is the name of the department. The problem is that if the department name should change, every record that refers to it must be changed as well. In addition, repeating the department name is a waste of space (though that's not a big issue given how cheap disk space is today). Since the department number uniquely identifies the department, there's no reason for the duplication.

Empnum	Lastname	Firstname	Deptnum	Deptname
10001	Smith	John	200	Sales
10002	Jones	Janice	100	Marketing
12394	Martin	William	100	Marketing
13847	Jackson	James	300	Production
21934	Anderson	Arthur	200	Sales
29381	Barrymore	John	300	Production
19384	Hartman	Mary	300	Production

Figure 2. Duplicated data—The DeptName field in the Employee table repeats the information in the Name field of the Dept table, wasting space and creating extra work if a department name should change.

Figure 3 shows the Task table and demonstrates the other main problem, the use of meaningful primary keys. Here, the number assigned to each task is based on the department number generating the task. While that may be a useful mechanism for the people working with the application, if a user changes the primary key to make it fit better, records in the Assigned table have to change. Good database design says that primary keys should never be shown to users; that way, they can't be changed. Meaningless keys that are hidden from the users are called "surrogate keys."



Tasknum	Task
10000	Marketing Plan
20010	Sales Meeting
20020	Sales Report
30010	Production Report

Figure 3. Meaningful primary keys—The Tasknum field is based on the department number.

The goal in updating this database is to add a surrogate key field to each table and to remove the repeated data, so that changes in one table don't require changes in another.

Adding Primary Keys

The first step for any table is adding a surrogate primary key. My preference is to use an Integer field with the AutoInc attribute set, so that I don't have to write any code to generate the keys or ensure that each new record gets a new key.

However, for an existing table, it's not enough to simply add such a field. AutoInc fields are read-only, offering no way to fill in the field for existing records. So a three-step process is necessary: first, add the Integer field; then, populate it for existing records; finally, change it to AutoInc starting with the next value. In addition, it's a good idea to create an index based on the new primary key.

Writing a generic function to do this is straightforward. Listing 1 shows AddPK.PRG. (This code assumes that you have control over the environment in which it runs and doesn't include much error-handling.)

Listing 1. Adding primary keys—Adding a surrogate primary key to existing data requires several steps.

```
*PROCEDURE AddPK
LPARAMETERS cTable, cField

IF FILE(FORCEEXT(cTable, "DBF"))
  SELECT 0
  USE (cTable) EXCLUSIVE ALIAS __AddPK
  IF TYPE(cField) <> "I"
    * Add it
    ALTER TABLE (cTable) ADD (cField) I

    * Populate it
    REPLACE ALL (cField) WITH RECNO()

    GO BOTTOM
    STORE EVALUATE(cField) TO nLastID

    * Make it auto-increment
    IF NOT EMPTY(CURSORGETPROP("Database", "__AddPK"))
      ALTER TABLE (cTable) ALTER (cField) I ;
      AUTOINC NEXTVALUE nLastID+1 PRIMARY KEY
    ELSE
      ALTER TABLE (cTable) alter (cField) I ;
      AUTOINC NEXTVALUE nLastID+1 UNIQUE
    ENDIF
  ENDIF

ENDIF

USE IN __AddPK
ENDIF

RETURN
```

To use this function, call it passing the name of the table and the name for the new primary key field. For example:

```
AddPK("Dept", "iID")
```

Replacing Meaningful Fields with Foreign Keys

Once a table has a surrogate key, the next step is to use that surrogate key to refer to that table in other tables. For example, once the Dept field has a surrogate key, the DeptNum and DeptName fields in Employee should be replaced with a single field iDeptID that points into the Dept table. The tricky part is making this change without losing the existing data.

To avoid confusion, let's refer to the table which has just acquired a surrogate as the *PK table* and the table that refers to the PK table as the *FK table* (FK for "foreign key"). As long there's a way to uniquely

identify a record in the PK table based on information already in the FK table, we can automate this process.

As with adding a primary key, a multi-step approach is called for:

1. Add the foreign key field to the FK table.
2. Populate the new foreign key field based on the data already in the FK table.
3. Index on the foreign key field.
4. Remove any index tags of the FK table that refer to the fields being removed (the repeated data from the PK table).
5. Remove the repeated data fields.

All the steps are straightforward except for the second. If the PK table has an index tag based on one or more of the repeated fields and that tag uniquely identifies a record in the PK table, we can set a relation between the two tables and issue REPLACE. That's the case for most of the relationships in the example Office database.

It's a little more complicated when there is no such index tag. In such situations, we need to use a function like LOOKUP() to find the matching record. In the example database, Employee includes the job type (matching the JobType field of JobType), but JobType isn't indexed on this field.

Since the application I'm working with has both kinds of cases, my function handles them both.

Here's the code for AddAndPopulateFK.PRG:

```
* Add FK to specified table and populate it,
* based on existing data
LPARAMETERS cFKTable, cFKField, cPKTable, cPKField, ;
             cPKDataTag, cFKRelExp, aDropFields

LOCAL cPKFieldAliased, cDropClause

IF FILE(FORCEEXT(cFKTable, "DBF"))
  SELECT 0
  USE (cFKTable) EXCLUSIVE ALIAS __FKTable
  IF TYPE(cFKField) <> "N"
    ALTER TABLE (cFKTable) ADD (cFKField) I

    IF NOT EMPTY(cPKDataTag)
      USE (cPKTable) ORDER (cPKDataTag) IN 0 ;
      ALIAS __PKTable
      SET RELATION TO EVALUATE(cFKRelExp) INTO __PKTable
      cPKFieldAliased = FORCEEXT("__PKTable", cPKField)
      REPLACE ALL (cFKField) ;
```

```

        WITH EVALUATE(cPKFieldAliased) IN __FKTable
    SET RELATION TO
ELSE
    * No index for desired tag.
    * Use specified expression instead
    USE (cPKTable) IN 0 ALIAS __PKTable
    * Replace aliases in expression
    cFindValue = STRTRAN(STRTRAN(cFKRelExp, cPKTable, ;
        "__PKTable"), cFKTable, "__FKTable")
    REPLACE ALL (cFKField) ;
        WITH EVALUATE(cFindValue) IN __FKTable
ENDIF
USE IN __PKTable

* Index on new FK
INDEX ON &cFKField TAG (cFKField)

* Remove extraneous fields, taking tags along
ATAGINFO(aTags)
cDropClause = ""
FOR nField = 1 TO ALEN(aDropFields,1)
    IF ASCAN(aTags,aDropFields[m.nField],-1,-1,1,7) > 0
        DELETE TAG (aDropFields[m.nField])
    ENDIF
    cDropClause = m.cDropClause + ;
        " DROP COLUMN " + ;
        aDropFields[m.nField]
ENDFOR

IF NOT EMPTY(m.cDropClause)
    ALTER TABLE (cFKTable) &cDropClause
ENDIF
ENDIF

USE IN __FKTable
ENDIF

RETURN

```

The function takes seven parameters:

- cFKTable is the name of the FK table.
- cFKField is the name of the field to add to the FK table (the foreign key).
- cPKTable is the name of the PK table.
- cPKField is the name of the primary key field in the PK table.
- cPKDataTag is the name of a tag in the PK table that can be used to uniquely identify the record referenced in the FK table. Leave this parameter empty when there's no appropriate tag.
- cFKRelExp is the expression to use either to set a relation from the FK table into the PK table, or to look up the appropriate

value in the PK table (when cPKDataTag is empty). Fields must be aliased.

- aDropFields is an array listing the fields to be removed from the FK table.

To use the function, set up the array and call it. For example, after adding the new primary key to Dept (as shown earlier in this article), we can use AddAndPopulateFK to update Employee, as follows:

```
LOCAL aDropFields[2]
aDropFields[1] = "DeptNum"
aDropFields[2] = "DeptName"

AddAndPopulateFK("Employee", "iDeptID", "Dept", "iID", ;
                 "DeptNum", "DeptNum", @aDropFields)
```

For cases where there's no matching key, you have to come up with a look-up expression. For example, here's the code to add a primary key to JobType and then create a foreign key to JobType in Employee:

```
AddPK("JobType", "iID")

LOCAL aDropFields[1], cLookupExpr
aDropFields[1] = "JobType"
cLookupExpr = "LOOKUP(JobType.iID, " + ;
              "UPPER(Employee.JobType), " + ;
              "JobType.JobType)"
AddAndPopulateFK("JobType", "iJobTypeID", ;
                 "JobType", "iID", ;
                 "", cLookupExpr, @aDropFields)
```

This month's Professional Resource CD contains UpdateOffice.PRG, a program that performs the complete transformation for the example Office database. Figure 4 shows the Office database after running UpdateOffice.PRG. Both the original and the transformed database are included (in separate folders) on the Professional Resource CD.

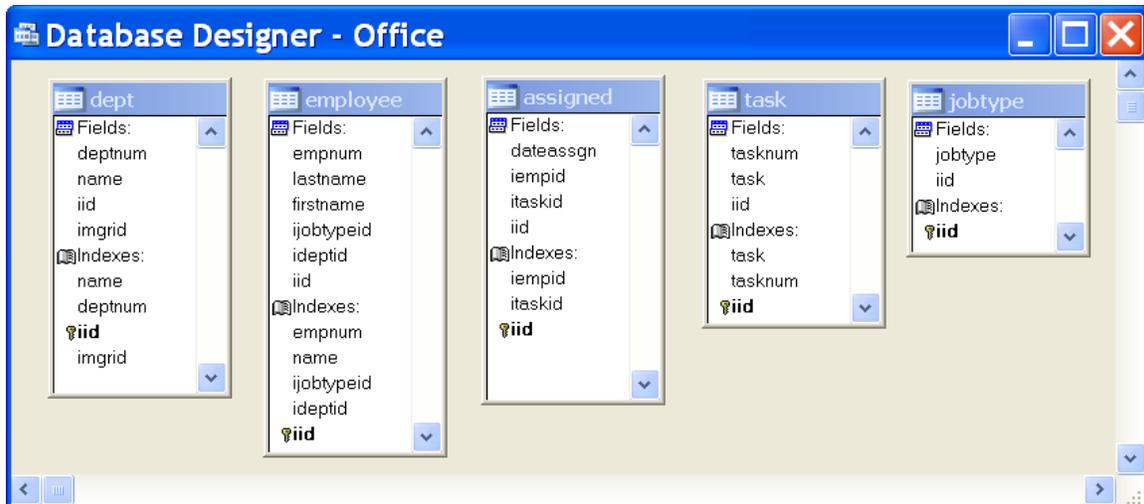


Figure 4. Transformed database—Every table has a primary and only foreign keys are used to link tables.

Final thoughts

The application I'm working on has over 100 tables (though some of the existing tables will disappear or be consolidated with others along the way). I'm making these changes in phases. The functions described in this article will save me hours of coding in each phase.

Sidebar: Why not use SDT?

The application I'm working on uses Stonefield Database Toolkit to manage the database and checks for updates to data structures each time it runs. However, SDT can't handle changes that require moving data from one table to another. Therefore, my code to update the existing data structures using `AddPK()` and `AddAndPopulateFK()` runs as part of a post-setup executable before the new version of the application ever runs.