The Fox Hunt

By Whil Hentzen

One important feature of Visual FoxPro is the inclusion of a data dictionary, and one fundamental capability of the data dictionary is the ability to specify a primary key for a table. If you're new to the term, a primary key is one or a combination of fields whose contents is unique for every record in the table. The purpose of a primary key is to be able to uniquely identify a record in order to relate it to other tables. Oftentimes, people use fields that contain real data as primary keys, such as employee numbers or customer Ids. However, it is all too easy to run into problems when ensuring uniqueness with fields that contain meaningful data.

Let's take Social Security Number as an example of a primary key for a table that includes employees and their dependents - a perfect choice, since everyone has a SSN, no one can have more than one, and they're never duplicated, right? Well, except for newborns, who may not have one. And non-citizens, who may not have one either. And, hard as it may be to believe, the Social Security Administration has been known to make a mistake and use a SSN more than once.

How about another example. You're developing a membership tracking application for a country club. In between Long Island Iced Teas out on the veranda, the membership chair promises that the four digit membership number will always be unique. So you develop the app using the membership number as the primary key for the Members table and all is running smoothly - until the club merges with another club that is also using a four digit membership number. Lo and behond - hundreds of duplicate Member ID numbers and the entire application is built around the concept of a unique number.

A solution to this potential catastrophe of a meaningful data item is to use a meaningless, internally generated ID whose exitence and use is hidden from the user. After all, to most users, the internal mechanisms are magic anyway. You can still use Member Numbers and Part IDs so that the user can uniquely identify the records, but by using an internal key, you won't get burned should the requirements of the system mean changing the rules of how the Part ID is handled.

At first glance, it looks like creating an internal ID would be easy - define a field, say "nIDPart" of type numeric and width of 6 with no decimals. You'd be able to store up to a million records - plenty for most systems, it would seem. However, depending on how often records are added and deleted, and how you handled deleted records, you may need to have more than a million IDs available. In other words, if you add and delete frequently, and you don't "reuse" deleted records, a table with 50-100,000 records may run out of unique IDs rather quickly. And just because you're working on small systems today doesn't preclude a multi-million record application hitting your desk tomorrow. So it's best to plan for the ability to handle tens or even hundreds of millions of records. However, now we're dealing with a rather long key - a billion values would require a ten character field, and over a large number of records, that's a lot of storage space used that doesn't need to be used.

An alternative method is to use a "binary key" which is made up of a range of value from ASCII(1) through ASCII(255). In other words, instead of the first key value being the number 1 (the numeric equivalent of 001), you'd use a character string made up of ASCII(1) + ASCII(1) +ASCII(1). The second key value would be ASCII(1) + ASCII(1) + ASCII(2), the third would be ASCII(1) + ASCII(1) + ASCII(3). The advantage is that you can store significantly more unique values in a fixed number of characters. Whereas a pure numeric key of length three could only stored 1000 unique values, a key using all 255 ASCII values in each position could contain 255 * 255 * 255, or 16,581,375 unique values. If you extend the size of the key to four or five, you can generate 4,228,250,625 or 1,078,203,909,375 unique values. One trillion unique keys should be sufficient for most applications, don't you think?

However, there's one caveat to this concept - your keys will look like the gibberish when you TYPE a binary file in DOS - full of happy faces, clovers, and line drawing characters. Remember that the upper ASCII characters (those above ASCII(127)) have different meanings in different languages. FoxPro attaches a "code page" to a table to indicate what the characters between ASCII(128) and ASCII(255) are. If you are going to be moving data from one code page to another, those values will get clobbered, and the mess that's been made of your nicely designed system of unique keys will make you wish cleaned tables at an all-night truck stop.

The answer to this is to use a restricted range of ASCII values. I use the range ASCII(65) to ASCII(122). This is the range starting with an upper case "A", includes some punctuation and the numbers, and then finishes off with the lower case letters. This range includes 58 unique values, and a five character key based on this subset will still generate 656 million unique values. If you need more, a six character key will generate 3.8 billion unique values. I use the five character key because I haven't run into a system where I've been that close to needing a 600 million unique values, and I've found the a five character key is a lot easier to read than a six character field.

Now that we've got a good mechansim for designing unqiue keys that is impervious to changing system specifications and the vagaries of changing system specifications, but is still efficient as far as storage space goes, we need a function that will automatically generate the next unique ID. There's more to this than meets the eye, so we'll spend the whole next FoxHunt investigating the solution.