Issue Date: FoxTalk January 1996

Save Space in Memo Fields

Bob Grommes

Here's an overview of how block sizes affect memo field size and fragmentation in both Visual FoxPro and FoxPro 2.x, plus the scoop on a little-known Visual FoxPro enhancement to memo fields and some coding tips to reduce memo field "bloat."

Memo fields (variable-length fields in the parlance of many languages) have been with us for a long time in the Xbase world. Old-timers will recall that they weren't very useful back in the pre-FoxPro days. One of the most exciting things about FoxPro 1.0 was that memo fields could handle binary data of arbitrary size, and not just text. Not only that, but we finally got some useful tools and commands for manipulating memo field data.

But there are some interesting benefits to understanding how FoxPro implements memo fields. Applying your knowledge of what goes on "behind the scenes" can result in significant, sometimes dramatic, disk space and efficiency savings, as I'll demonstrate.

Memo field data is stored in an .FPT memo file, which is completely separate from the .DBF table file. In the .DBF, a pointer is stored to the physical location of the memo data in the memo file. At that location in the memo file, FoxPro sees eight bytes of data that indicate the number of blocks the memo data occupies (almost always one block), and the length of the first memo block. So far, so good.

But FoxPro introduces a wrinkle. When a table is created, the .FPT file is stamped with a minimum memo field allocation for each value stored. This "memo block size" is determined by the SET BLOCKSIZE command in effect at the time the table is created.

The SET BLOCKSIZE command is an odd beast, because like many parts of the FoxPro language, it's saddled with backward compatibility issues. In dBASE III Plus, you could SET BLOCKSIZE to any value between 1 and 32. If you SET BLOCKSIZE TO <n>, dBASE would use a block size of <n> times 512 bytes.

This meant that even if you SET BLOCKSIZE TO 1, the minimum value, a minimum of 512 bytes was always allocated in the memo file for the contents of any memo field -- an extravagance in the days of 20M hard disks!

FoxPro 1.0 extended SET BLOCKSIZE so that if you used a value of 33 or greater, it was interpreted in bytes instead of 0.5K units. This effectively gave us block sizes ranging from 33 bytes and upwards. The default value for SET BLOCKSIZE in FoxPro is, and always has been, 64.

A hidden treasure

You may not have noticed the new ability to set the memo field block size to "zero" in Visual FoxPro, using the command SET BLOCKSIZE TO 0. When you SET BLOCKSIZE TO 0, Visual FoxPro effectively uses a block size of 1 byte -- in other words, the contents of memo fields can now take up only as much space in the .FPT file as the data demands. More accurately, a table created while SET BLOCKSIZE TO 0 in effect stores each memo field in a single block of exactly LEN(<memo field>) + 8 in size. In effect, there is no minimum allocation for a block of memo data.

For example, consider an .FPT with the following two memo field values stored in it:

```
This is a test.
X
```

Ignoring the 512-byte header that appears once at the very beginning of any .FPT, the .FPT storing these two strings takes up 32 bytes if you SET BLOCKSIZE TO 0 before creating the table, but takes 82 bytes if you SET BLOCKSIZE TO 33. With the default SET BLOCKSIZE TO 64, it takes 128 bytes. Clearly, with thousands of records, smaller block sizes can add up to big space savings.

Incidentally, regardless of the block size setting, there is one more improvement in Visual FoxPro memo fields. The memo field pointer in a FoxPro 2.x .DBF used to take 10 bytes per record, whether or not the memo field contained data; in Visual FoxPro, it takes only four bytes per record. That's a savings of six bytes per record, or about 1K times the number of memo fields for every 170 records in a table.

What's the best size?

Should you always SET BLOCKSIZE TO 0 in Visual FoxPro (or to 33 in FoxPro 2.x)? Probably not, but these are generally the best defaults to use. The trade-offs in memo field sizes have to do with FPT fragmentation.

I've described the "ideal" situation that exists when a table contains nothing but newly added, unedited records. The trouble begins when you start editing memo fields. Any time you replace the existing contents of a memo field with a longer string

than the original, FoxPro abandons the original block and rewrites the entire string at a new location in the .FPT. The more often this happens, the more wasted space in the .FPT, and it's reclaimable only via PACK, PACK MEMO, or by copying the table. On the other hand, regardless of memo block size, once a block of memo data is written to the .FPT, values that are the same length or shorter will reside in the same area and the only wasted space will be the difference between the lengths of the original and the new values.

It's important to realize that block sizes have to do only with minimum allocations for memo field data. For example, if you SET BLOCKSIZE TO 64, that doesn't mean that a very long memo field value will be broken up into dozens of 64-byte blocks that are chained together through a pointer list; a large memo will be physically stored in a single block. SET BLOCKSIZE TO 64 just means that 64 - LEN(<memo field>) bytes will be "wasted" if the memo field value is less than 64 bytes long. But it also means that you can REPLACE any values you want into the memo field, as long as the values are 64 bytes or less in length, and the memo data won't be "moved" to a new block, thereby "orphaning" the entire original 64-byte allocation.

How do you determine the best memo block size for any given table in your application?

Smaller block sizes are ideal for tables where the memo field contents aren't changed frequently, are generally short in length, or for smaller tables that can be packed frequently. For example, Visual FoxPro uses a block size of 1 byte when creating forms (SCXs) and class libraries (VCXs), regardless of what you currently have the memo field block size set to. These small tables are frequently rewritten and can be packed quickly, and the memo fields usually contain short strings. Furthermore, SCXs and VCXs are often embedded in .APP or .EXE files, where space is at a premium. I suspect this latter consideration may have had a lot to do with inspiring the new SET BLOCKSIZE TO 0 option and the new, more compact memo pointer in the .DBF. (Incidentally, unlike SCXs and VCXs, database containers (DBCs) are created with whatever block size is in effect at creation time, just like any other table, while report form tables (FRXs) are always created with a block size of 33).

Consider larger block sizes when you know that most of your memo field data will consist of longer strings, or will be changed frequently with data of varying widths.

Analyzing application data

In general, if a table has a single memo field, and you have some idea of what the average length of the data going into that field will be (ignoring records with no data in the field), then that's probably your ideal memo block size for that table. The smaller the percentage of records that actually have memo field data, the closer the block size can afford to approach the maximum length that the field will ever store.

Often, it's best to let a new application run in the field for awhile and accumulate a significant amount of data. You can then analyze the contents of the memo fields and adjust the block size by using SET BLOCKSIZE followed by COPY to create a new table with a more efficient FPT. If the table isn't too big, you might consider making several copies of it with different memo block sizes and using the setting that produces the smallest FPT.

However, for an accurate comparison, first do a PACK MEMO on the original table to remove any fragmentation. Not only will your comparisons be more accurate, but the degree of fragmentation will be instructive. Even if a particular block size initially yields the best efficiency, a larger block size may help prevent fragmentation as the file is used use over time.

Things get more complicated if there are multiple memo fields in the table with varying requirements. Regrettably, the memo field block size is an attribute of the table, not of individual memo fields. In such situations you'll have to weigh the requirements of each memo field individually, weight them according to frequency of usage, and come up with a good overall block size for the entire table.

Block size-related commands

SET BLOCKSIZE doesn't alter the block size of existing tables; it simply governs the block size of any tables that you create. Any time your application is about to create a table (CREATE TABLE, COPY, and similar commands), consider what you expect to see in the table's memo fields and consider issuing an appropriate SET BLOCKSIZE command first.

You can determine the memo block size (in bytes) of existing tables using the SYS(2012) function against the table once it's open (SYS(2012) returns zero if the table has no memo fields). You can use SET(`BLOCKSIZE') to determine what your current SET BLOCKSIZE setting is (in bytes).

A likely problem and its solution

Here's one caveat to observe: the SET(`BLOCKSIZE') function always returns the current block size setting in bytes. If, for example, you SET BLOCKSIZE TO 1, SET('BLOCKSIZE') will return 512. If you're changing the block size in a routine and want to preserve and restore the caller's block size (a technique often used in application startup and shutdown code as well), you run into a problem. Consider the following code:

```
nOldBlockSize = SET('BLOCKSIZE')
SET BLOCKSIZE TO 64
*  more code here, then reset the caller's block size
SET BLOCKSIZE TO (nOldBlockSize)
```

If the caller had previously SET BLOCKSIZE TO 0, then SET('BLOCKSIZE') will return a value of 1, since SET BLOCKSIZE TO 0 produces one-byte blocks. But when you "restore" the caller's block size, you'll issue SET BLOCKSIZE TO 1, which will create 512-byte blocks, just like in the good old days of dBASE III Plus! To get around this problem, you need code like the following instead:

```
nOldBlockSize = SET('BLOCKSIZE')
IF nOldBlockSize = 1
  nOldBlockSize = 0
```

```
ENDIF
SET BLOCKSIZE TO 64
*  more code here, then reset the caller's block size
SET BLOCKSIZE TO (nOldBlockSize)
```

REPLACE and fragmentation

A technique you can use to minimize .FPT fragmentation regardless of memo block size is to avoid repeated REPLACEs on a memo field whenever possible. For example, suppose that you have code that appends transaction log information to a memo field named TranLog. Perhaps you need to add several lines to the field. You might have code similar to the following:

```
FOR n = 1 TO nMessages
  REPLACE TranLog WITH CHR(13) + cTransMsg[n] ADDITIVE
ENDFOR
```

This code extends the memo field on each iteration of the loop, potentially orphaning .FTP file space each time. It's much better, and probably faster, to build the additional data in memory and do a single REPLACE:

```
cNewMessages = ''
FOR n = 1 TO nMessages
  cNewMessages = cNewMessages + CHR(13) + cTransMsg[n]
ENDFOR
IF NOT EMPTY(cNewMessages)
  REPLACE TranLog WITH cNewMessages ADDITIVE
ENDIF
```

I once had a text management application that would read in text files, do on-the-fly translations, and write the text files a line at a time to a memo field. I noticed that the memo file was bloating by a factor of several hundred K per day, even though no new records were being added to the table. This coding trick got rid of most of the "bloat."

Conclusion

Fiddling with memo block sizes isn't one of the most pressing issues in FoxPro database development, but as you deal with larger and larger databases, some attention to detail in this area can have a significant impact on disk storage efficiency. And, as you saw in the code examples I just shared, paying attention to the subtleties of syntax can produce big savings, too.

Finally, I suspect that Visual FoxPro is taking us in directions that are likely to cause us to provide database administration services to clients in addition to application design and development services. Knowing how to tune memo fields is one skill that will differentiate the kind of administration you can provide from what the client can do in-house.