Performance and COBOL

Bruce Toback
OPT, Inc.
2205 Fulton Road
La Verne, CA 91750 USA
(714)593-1681, Telex 532678

Performance and COBOL

The first standard COBOL was proposed in 1960, when computers were very different from today's mini- and microcomputers. They were single-user, multi-priest behemoths with architectures designed to fit the electronic components available at the time. Fast memory was prohibitively expensive; slow memory was merely expensive. The processor itself required a large corporation and a large staff to purchase and maintain it.

The original COBOL standard was designed with these facts in mind, and many of the programming practices and accumulated wit and wisdom of the COBOL programmer come from those times. Presented here is a selection of that wit and wisdom, and how it relates to COBOL/3000 (and indeed most modern computers).

1. Indexing is faster than subscripting.

Remember registers? (If you ever programmed in assembly language, you certainly do. If you've only programmed in a high-level language, you probably don't.) General purpose computers usually had a limited number of these high-speed memory locations, generally eight or sixteen. The compiler, when generating code for a COBOL source program used several of these for itself, but the remainder were available to the programmer for USAGE IS INDEX items. Indexing was much faster than subscripting because in order to access an array element, a subscript would have to be brought into a register, and possibly converted to a different data type. Indexing, by definition, meant using an item which was already in a register, so bypassing the conversion and data movement steps. Worse still, some computers had addresses that could be operated on only by special instructions (e.g., the Burroughs B200/300/500 series). This is the reason that indices can only be added to or subtracted from.

On the HP3000, though, there is only one index register, andit is shared by all arrays. Therefore, as long as the subscript you are using is USAGE COMP PIC S9(4), there is no difference between indexing and subscripting - either in the generated code or the speed of the resulting program. Shops which try to speed up programs by converting them to use indexing would be better off devoting the time to programmers' vacations: system performance would then at least be improved because of a smaller program development load! However, performance can be improved by changing your subscripts from COMP-3 to COMP: the compiler emits code necessary to do the required conversion, but this is relatively expensive in run time. (But see 5.)

2. COBOL sorts take longer than external sorts.

This is very application dependent, and again has strong roots in history and folklore. In the Dark Times, the COBOL SORT verb generated in-line code to call some routines the compiler folks wrote. The compiler folks generally had better things to do than write sorts, so the sorts were not necessarily very good. In addition, these sorts were not very adaptive to circumstances, and had a limited performance range. They might involve additional overlays that had to be read in from (heaven forbid!) cards. Besides, in a batch-oriented, job-step environment there was not much point to an internal sort.

On the HP3000, though, all sorting is done by the SORT/3000 subsystem, regardless of whether you use SORT.PUB.SYS or the COBOL SORT verb: the HP3000 COBOL compilers simply emit code to call SORT/3000 intrinsics on behalf of your program. The result of this is that sorts done with the same sets of keys and in the same kinds of environments will take the same amount of time, regardless of whether you invoke them through COBOL or through MPE.

The key here is in the same kinds of environments. SORT/3000 needs memory to work: the more, the better. If your COBOL program requires 20k words of memory for itself when you execute the SORT verb, SORT/3000 will get only about 9k. Experiments show that when sorting 80-byte records, SORT/3000 needs about 8k words to produce acceptable performance, and works best with at least 16k words. So, you'll have better performance with an external sort...

Sometimes. To determine whether to use an internal sort or an external one, you should look at your entire application. Many batch-oriented systems converted to run on the HP3000 have jobsteps that include a sort to a temporary file, a report on the temp file, a different sort to a temp file, a report on the new temp file, and so on. This means that each record in your master file is being handled three times: once as input to the sort, once as output to the temp file, and once again as input from the temp file. Replacing the external sort with a SORT verb with an OUTPUT PROCEDURE reduces this to only one: records are read once by SORT/3000 on behalf of your program, and then passed to your program via the output procedure. (Of course, SORT/3000 handles records several times during its execution, but this number is largely irreducible except by providing more memory for the sort.) The results of this kind of redesign can be dramatic: reductions of 2:1 or 3:1 in runtime are possible.

3. COMPUTE is faster/slower than the ADD/SUBTRACT/MULTIPLY/DIVIDE verbs.

Like SORT, this depends on how you use the various computational verbs available to you in COBOL. If the object is to perform a complex series of arithemetic operations, using COMPUTE will generally be faster (by a few microseconds). If you are simply adding values to an accumulator, ADD and COMPUTE will generate exactly the same code, and so there will be no performance difference.

4. COBOL is inefficient at arithmetic.

As the previous point shows, how "efficient" a language is at a particular task depends mostly on how you use it. (This generalization applies only to general-purpose languages.) By following a few simple rules, COBOL is as efficient as, say, FORTRAN at arithmetic. In particular, avoid either implicit or explicit type conversions: don't mix COMP-3 and COMP items, and try not to mix COMP PIC S9(1-4) with COMP PIC S9(5-9). And in addition, avoid arithmetic with USAGE DISPLAY items, since these always require type conversions. (The HP3000 has instructions to operate directly on COMP and COMP-3 items, but lacks arithmetic instructions to operate on DISPLAY types.) For an extended discussion of this, see Jim May's paper Programming for Performance , published in the Proceedings of the 1982 HP3000 IUG Conference, Edinburgh.

5. Searching: IMAGE and internal techniques.

earches, or table lookups, are very common in data processing operations. You probably do them without thinking in most cases, since every IMAGE master lookup (DBGET mode 7, or DBFIND) is really a search operation. If you routinely use IMAGE to do your table lookups for you, you might be surprised at how much time can be saved by using your own code to perform the same operation.

The examples used for this article are derived from a real-life applicaiton: printing a purchase order report. Purchase order records were contained in a detail data set containing four items: a part number, a vendor number, a purchase date, and a quantity. (Other items were left out for the purpose of this example.) Each part number's corresponding manual master record contains a description for it, and each vendor number's corresponding manual master record contains the vendor's name and address. To test the methods outlined here, I wrote a small COBOL-II program to generate a simple purchase order report, and then modified it to try different search methods.

In the first example, the item being looked up was the vendor's name. The program in Figure 1 is representative of most such programs: a serial read (or perhaps, a sort output procedure) gets each detail record, and then a lookup is performed on the associated manual master to get descriptive information. The lookup is entirely contained in the paragraph **GET-VENDOR,** so that various lookup techniques can be tried without making significant changes in the rest of the program. (If your programs are written like this, you should be able to simply lift code from the examples, place it in your system, and enjoy the kudos.)

All of the examples, and the performance information derived from them, comes from a 1/2-megabyte Series 30 running T-delta-1 (MPE-V/T) and a single 7925 disc drive. You should keep this in mind when you examine the performance data. If you are running a faster system (you couldn't *possibly* be running a slower one), you should divide the "CPU-seconds" figures by 2 if you are on a Series III or Series 37; by 5 on a Series 4x CPU; or by 12 (!) if you are on a Series 6x machine. Clock times will not scale by as much, since non-cached I/O rates for a single disc drive are almost identical on all CPU's. In addition, all timing was done with disc caching turned off. Caching will in most cases improve wall-time performance, but leave CPU time almost unchanged. (Disc caching resulted in a slight performance degradation on this small-memory Series 30.)

The first test was run using the "standard" program in Figure 1. To produce this 9,000-line report took just under thirteen minutes, and used 475 CPU seconds. (If you are scaling these numbers for your Series 68 CPU, you should come out with about 5:45 clock time, and 40 CPU seconds.) Is any improvement possible? Since by now you have looked at the graph in Figure 5, you know the answer is "yes." The program in Figure 2 replaces the DBGET with a SEARCH verb, which performs a serial search of a table in memory. Of course, some preparation is required for this technique, and this is done in the paragraph **INIT-VENDOR-SEARCH. INIT-VENDOR-SEARCH** does a simple serial read of VENDOR-MASTER, and saves the vendor number and vendor name in VENDOR-TABLE. The result of running Program 2 is almost a 2:1 reduction in run-time, and savings of about one-third in CPU time. Clearly the extra effort of creating a table in memory was well-spent.

COBOL, though, supports an even faster search verb: SEARCH ALL. Using SEARCH ALL requires that the table used for the search be sorted into ascending or descending order. This, of course, requires some additional preparation, shown in Figure 3. SORT-VENDOR-TABLE is a generalized Shell sort, and you should be able to use this in any of your programs by changing only the identifiers used for the table. For sorting tables that fit entirely in memory, it is *much* faster than the SORT verb. (The sort would execute faster still if it were programmed in SPL, but the difference for small tables is not worth the extra effort.)

The result of using SEARCH ALL is a further improvement, although not as dramatic a change as eliminating IMAGE from the picture. The ratio will improve noticeably, though, with larger tables, as you will see shortly.

There is a search method even faster than binary search, however, for most commercial data. The "80-20" rule is a generalization about most things in business: twenty percent of the customers generate eighty percent of the revenue (or orders, or complaints); twenty percent of your vendors are responsible for eighty percent of your shortages, and so on. (As practical examples, think about the number of transactions in your accounting system that use the "accounts payable" or "inventory" accounts.) Because your computer records will (should) reflect the parameters in your business, you will probably discover that eigty percent of the records in a detail are linked to only twenty percent of your master entries. (Of course, for pure "control" information such as invoice numbers, this is not true.) This will probably apply to parts in inventory, to purchase order line items, to sales order items, and other details requiring "descriptive" information from associated masters.

You can use this property of business data to come up with a new search rule for your memory array: Whenever an item is found in the table, it is moved up one entry. Repeated application of this rule rapidly and automatically organizes your table by frequency of use. The resulting program is shown in Figure 4; the very simple change is in paragraph **GET-VENDOR.** About 20 percent less clock time is required now, and about about 15 percent less CPU time as well. (The test data used for these timings was designed to have approximately an 80-20 distribution.)

To summarize the information gathered so far, look at Figure 5. (You've probably already done this, but look again.) Clearly, the "standard" method is the worst of the lot. Given the small effort involved in adding *any* of these internal searches, they are probably an easy way to gain performance. The small amount of "overhead" time in each bar is the time required to do any pre-processing before the report starts. In program 1, the time is spent only in opening the data base and output files. In programs 2 and 4, there is additional time required to read the master data set, and in program 3, some additional time is required to sort the array. The overhead time is very low for all three "fast" methods.

These techniques will work on small master data sets (a few hundred entries or so), but what about masters too large to fit into the stack? (Unless you are sorting using a separate process, you should leave at least 12K words for any required sort.) In the case of the vendor file, the maximum would be about 1100 entries. In the case of the part master, however, the maximum would be only about 650 entries. (In a practical program, there would be even fewer entries, since these small test programs make no allowance for V/3000 space, extra files open, or other uses of the stack. Moreover, there would probably be several manual masters on which lookups were to be performed.) A modification of the technique used in Program 4 can still be used as long as the master data set or sets are not extremely large.

In this case, rather than reading the entire master data set into memory at once, entries are read in "on demand." No initial "preload" of the array is performed. Instead, the table size is set to zero during initialization, and as each detail record is read, a table search is performed. If the table search fails, the program looks up the key in the appropriate master data set, and the newly read record is added to the growing array. (If the requested record *is* found in the array, its position is adjusted as in Program 4.) When the array becomes full, there are two choices. First, the entry at the bottom of the array can be thrown away to make room for the new entry. This method will efficiently handle cases in which detail records with a particular key are "clustered" - newly-read records will tend to migrate to the top of the array as their "cluster" is read.

6. A perspective on performance.

With the exception of searching and sorting, most of the performance questions analyzed here, and most performance debates in general are matters of microseconds on modern computers. You should take this into account before embarking on any large rewrite projects. For example, changing a subscript from COMP-3 to COMP may save 50 microseconds (millionths of a second) per array access. If you access the table in question ten million times over the course of a three-hour run, your rewrite will save 500 seconds, or about eight minutes out of 180 - a 4-1/2 percent improvement.

Conversely, if you are doing an external sort followed by a report, for a total run-time of three hours, changing the application to run with a single internal sort might save an hour or more, a savings which would justify a two-day redesign. (Assuming that the report is run more often than once a year.)

Finally, performance is much more a matter of choosing the right design in the first place, rather than "tweaking" a poor design to get more speed out of it. As the sorting example shows, shaving 20 percent off the run-time of a bad design will usually pay a lot less than finding a good design to start with. For an excellent discussion of this, see The Elements of Programming Style, Kernighan and Plauger (1974), Chapter 6.