Some Things Never Change

by Don Gall

Introduction

In the fall of 1995, the Mechanical Engineering Department at the University of Illinois Urbana-Champaign decided to have an open house and luncheon for former graduates in conjunction with the homecoming football game. At the last minute, the TV network covering the game decided to move the time of the game to get better coverage. This, in turn, caused all of the plans for the department open house to be moved to a time later in the day.

About 6 P.M. I went to one of my former professors who had organized the event to explain that I needed to leave because my mother, who was 89 at that time, would be worried if I did not get back to her house around the time she was expecting me. You need to understand that I was 60 years old, married, with 4 grown children and had not lived at home in 39 years. His reply was "Well, some things never change."

Old (really old) Computers and Their Limitations

My first experience programming computers was on the original ILLIAC computer at the University of Illinois in 1956. It was a first-generation programmable computer with paper tape input and output. It had 1024 words of vacuum tube memory and a huge 12,800 word drum storage unit. The ILLIAC terminology always referred to words. In today's terminology, it would be called a 40 bit byte. The 40 bits allowed a word to have a precision of about 12 decimal digits which was adequate computational accuracy for a computer of that era.

A program instruction consisted of two hexadecimal instruction bytes and a 10 bit address which was needed to uniquely locate each of the very large number (1024) of words of memory. Since a program instruction took up 18 bits, two instructions were combined in a single word with four bits left unused. The hexadecimal characters that were used were not as we know them today. Apparently, there were many World War II Army teletypes available in that era. Thus, a few of these teletypes became our off-line input and output devices. The keys that you pressed to produce the standard four bit hexadecimal hole patterns that we got to know and love were the numbers 0 through 9, and the letters K, S, N, J, F and L, which we remembered from the phrases King Size Numbers Just For Laughs or Kind Souls Never Jostle Fat Ladies. We now have 0 through 9 followed by A, B, C, D, E and F. (I can only remember this unusual sequence if I think of Always Buy Chicken, Don't Eat Fish which is not nearly as funny as the previous two!)

The execution times for a number of the program instructions are of historical interest. To move a word from memory to a CPU register took 55 microseconds. To add two registers took 90 microseconds. To read a word from the paper tape took four milliseconds. To punch a word in a paper tape took 17 milliseconds.

The drum storage system made 1 rotation every 16.9 milliseconds. Because of this relatively long interval, random access was not something anyone even thought seriously about. Serial reading and writing was the only method used. Even so, the maximum rate to read or write was 1.32 milliseconds per word. This meant that it took 24 (1320 / 55) times longer to read or write from the drum than from memory if we did it serially. It would take 154 (8450 / 55) times longer if we had not done the drum accessing serially. In 1956, CPUs were very slow, but drums were significantly slower.

About 20 years later, minicomputers had progressed to provide us with 64 KB of memory with transfer rates to the CPU in the 10 microsecond range. The disk drives of the era had mean access times of 80 milliseconds. This meant that it took 8,000 (80,000 / 10) times longer to read and write from the disk than from memory.

We now have PCs with 200 megaHertz CPUs and disk

drives with 8 millisecond access times. This means that we can move over 1 million bytes from memory while we do a single disk access. Everything has gotten a whole lot faster, but the relative cost in time of using the disk has gotten much higher over the years.

Is there a point here someplace? I certainly hope so! In the "old days," a programmer worked very hard to reduce the number of disk accesses his programs made in order to produce user-acceptable software. With the advent of faster and faster computers with larger and faster disks, there has been a tendency to not worry about disk accessing so much. On the contrary, we need to continue to worry about it. Some things never change!

How do we minimize disk accessing? Let us add a little more history before proceeding with that issue.

The Evolution of M[UMPS] Systems

The M[UMPS] systems prior to the introduction of the b-tree file structure had, by today's standards, very restrictive global file structures and a very limited amount of data which could be stored at any global node. In these systems, one-and-two-dimensional globals (e.g., ^GLB(LEV1) and ^GLB(LEV1,LEV2)) worked very well and three-dimensional globals worked well if they were designed properly. Anything above three dimensions tended to chew up vast amounts of disk space which was not something you wanted to do in an era when 10 megabytes of disk cost over $10,000. In this early M[UMPS], the indexes (e.g., LEV1 and LEV2) had to be positive integers. These early M[UMPS] systems could not store a data string with a maximum length greater than about 75 characters at any global node.

In spite of what now seem to be serious limitations, a lot of programmers produced a lot of good software that was functionally far superior to software developed in other programming languages. But, they produced that software using integer indexes with a small amount of data, sometimes only one field, at a global node.

The transition to ANSII Standard M removed these old limitations, but did data structures change? For some developers, the newly found freedom led to a complete rewrite of the software and global file structures. Other developers, who either did not see the need or who thought the cost to be too high, continued on with the old structures. Those that did not change now find that mapping these structures to a Relational Data Base Model is either extremely difficult and time-consuming or impossible.

The M literature and the M sessions at the annual meeting seem to emphasize writing good M code. Not much attention is paid to creating efficient data structures. Both good code and good structures are essential for an efficient and responsive system.

There was a time when M could compete with other languages and methodologies because the development time in M was more than one order of magnitude faster than with these other languages and technologies. One of my favorite quotes from James Martin is, "A language should not be called fourth-generation unless its users obtain results in one-tenth of the time as with COBOL, or less."[1] With the introduction of improved Rapid Application Development tools, the M community must move forward soon or risk losing this development time advantage.

At the risk of alienating members of that M community, let me state the following:

The ability to provide efficient data clustering using multidimensional arrays is the most significant advantage that M now has over other data base management languages or methodologies.

It is my contention that since this is a major advantage, it is therefore very important that we maximize that advantage to the fullest.

It is possible to write a DBMS in M which completely adheres to the RDBMS model. If, in the implementation of this structure, M stored its tables in the same way that most RDBMS systems store theirs, it would be, at best, less efficient and less responsive than those RDBMS systems.

How to Minimize Disk Accesses

There are two major factors in minimizing disk accesses:

1. Design the data structure of the disk so that data fields which are commonly used together are stored as close to one another as possible. This is the concept of data clustering which is discussed very infrequently by the implementers of the Relational Data Base Model.

2. Never write a routine which accesses the disk twice when once will suffice. A corollary of this is to not use the disk as if it were local storage.

Data Clustering

There are a number of good examples of the uses of data clustering. Two of the most common examples are discussed here.

The vendor invoice is an example of a real world problem which is handled much better in M than with standard RDBMS methods. The vendor invoice consists of two parts:

1. The name of the vendor, the date of the invoice, the total amount of the invoice and other such important items which appear once on the invoice.

2. One or more individual items purchased with the identifier, description, quantity, unit price and extended price for each item.

This is the classical "many-to-one" issue which is typically handled within an RDBMS by having two tables, one table for each of the parts 1 and 2 above. Each time a user references the entire invoice, it is necessary to do a logical join of the two tables to gather the necessary information. Lots of disk accessing!

The inherent multidimensional nature of M allows us to define a vendor invoice global with two indexes: 1. *accounting date* and 2. *invoice item* and store all of the information of part 1 above at this level. At a third index level we can define a debit item and store all of the information of part 2 above as individual debit items.

What does this give us? All of the information about any given vendor invoice will, at worst, be logically contiguous on the disk. In the majority of instances, one disk access at this level will not only give us all of the information about a single invoice, it will give us all of the information about a number of invoices on the same date. If you are only interested in one invoice, you are ahead of the game. If you are printing the invoice journal for a given date, you are well ahead of the game. This is one of the advantages of data clustering.

The standard double entry bookkeeping system leads to many other such examples within the accounting software industry. Customer invoices, cash disbursements, cash receipts and journal entries all lead to this "many-to-one" real-world issue.

A second ugly problem for RDBMS solutions (which occurs repeatedly in medical applications) is the "multiple members of a table" issue. A patient typically has more than one diagnosis, complication, drug, problem, etc. The RDBMS solution must create a separate linked table for each occurrence of these multiple members from a table of data types.

In one medical application that I know, 175 of the 475 total data fields are of this data type. The normalized Relational Model requires a very large number of tables. Lots of disk accessing! The attempt to move this data base from an M application (actually an early Meditech MIIS application written in 1976) to an RDBMS was a disaster from both a time and financial point of view. The original MIIS development took one programmer less than a year at a total cost including hardware of under $100,000. The move to a 6 gigabyte Sequent computer and an RDBMS was abandoned after two years at a cost of over $1,000,000.* It failed because the very large number of tables and complexity that number of tables introduced made data entry extremely slow and resulted in reporting which was so inaccurate that it could not be relied upon. Too much disk accessing!

M can handle the multiple members of a table in one of two ways. The easiest way is to pack the multiple members into a single field using a delimiter such as a comma to separate the individual table members. The Omega OODB development system^2 has a data type TMM (multiple members of a table) which knows how to deal with what otherwise appears to be a single field. This does not directly map to any relational data base because it can not be placed in normal form.

A second method which can be mapped to the relational model adds two additional dimensions to the node at which the other related data is stored. The first, a literal (e.g., "DX"), identifies the particular attribute which can have multiple values, and the second contains the value of each of the individual table members. The literal "DX" and possibly other such literals allow you to have more than one such multiple members of a table data type in a single data collection. Once again, M will provide useful data clustering in that a single disk access will likely pull in all of the related information. The relational model will have to join two (or more) tables to collect all of the related information.

*This entire data base will be moved this year to a 4 gigabyte Pentium PC. With the new tools developed by Omega^2 and the use of a TMM data type (multiple members from a table), it will require one programmer to spend about two months for programming.

Programming Access of the Disk

One of my favorite laws is more or less related to the Second Law of Thermodynamics. The law states that "it is possible to screw up a one car funeral." Over the years, I have seen many examples of this law as the law applies to disk accessing.

Without question, the worst instance that I have ever seen was a structure set up to facilitate cursor positioning for a number of different dumb terminals. The 80 commands for each possible X position and the 24 commands for each possible Y position were stored individually by terminal type in two different M globals, ^POSX(TYPE,X) and ^POSY(TYPE,Y). Moving the cursor to a specific X and Y location on the screen was accomplished by assembling a command from the two globals and then eXecuting the command. An incredible amount of totally unnecessary disk accessing! A few users in screen formatted entry routines brought the entire system to a crawl.

A second type of example appears in report generating routines which have a tendency to evolve over a period of time. Rather than rewriting the routine to minimize disk accessing, a patch is put in to $ORDER through all of the data again to pick up the new information. This may be a quicker solution for the programmer, but not an efficient solution for the system in general.

As we get down the ugly practices curve, we find areas which may or may not be inefficient depending upon how bad the practice is and how much cache memory the system has available for you. One of these practices is to use the same global variable two or more times in the same section of a routine without making this variable a local variable. In a second variation on this theme, the programmer negates a good data clustering design by making a number of other disk accesses in between the retrieval or writing of the disk data which has been so well clustered.

If there is adequate disk cache and the disk referencing occurrences are close enough together during execution, you may luck out with one or both of these practices. At times when the disk is not particularly busy, you may have enough cache of your own to make these practices work. Unfortunately, just when resources are scarce, these practices can make a slow system even slower. The main message here is to not rely on cache memory to rescue sloppy programming.

Summary

Every year around Super Bowl time, you can always find at least one sports broadcaster declaring, with heartfelt incredulity, that this particular team got to the Super Bowl because it had somehow rediscovered that paying attention to the fundamentals of football paid off. What a concept!

In football, the fundamentals consist of blocking, tackling, running, passing, kicking, not (getting caught) violating rules and some others. Computer programming, on the other hand, is a simpler game. All we have to worry about are the bottlenecks that the hardware dudes have caused. To their credit, the old bottlenecks of memory size, CPU speed, disk size, disk speed and dumb terminals have gotten much better. The interesting thing is that the use of long-term storage, be it the old drums or the new disk drives, remains the single biggest bottleneck for data base management systems. Relative to CPU speed, disk accessing has become an even worse bottleneck than it was 20 or 40 years ago.

The fundamentals remain the same. Design your data structures carefully, and treat the disk drive with respect. It can be your best friend or your worst enemy. To compete with the RDBMS technology, we must continue to do things that that technology cannot do and we must continue to do everything faster than can be done with that technology.

Some things never change. **M**

References

1. Martin, James. 1982. *Application Development Without Programmers*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

2. Gall, Don. "An M Implementation of Object Oriented Programming." *M Computing* 3, 1, (1995): 13-19.

Don Gall is CEO of Omega Legal Systems in Phoenix, AZ and a member of the MTA Board of Directors.
