The Performance Art of Spatial Mechanics

Jonathan W. Lowe

This column covers the role of emerging technologies in the exchange of spatial information.

Hardware bravado battles—we've all witnessed at least one. They begin with a casual mention of CPU speed, RAM cache, or hard disk size. "Dude, you could smell the smoke from my two 800 MHz processors when I rendered every street in the country in 20 seconds!" It's a challenge; will any listener dignify this meaningless comment with a response? Hopefully not, but if another macho device-lover overhears, he'll be unable to resist replying, "Oh yeah? Well my server's 2 GB of RAM could've done it in half the time by caching the whole dataset!" With two bloated tech-egos now mutually invested, the exchange could drag on for hours.

Annoying or fun as a macho hardware debate may be, its pivotal topic — system performance — remains as important as ever in the delivery of information from a computer system to a human being. In the spatial data industry, computers deliver information as graphic maps or images; both are extremely dense data formats. In other words, it's possible to pack more information onto the fixed area of a computer display using a map image than using plain text. The cost of high-density information, though, is large datasets and slow performance. The more data in storage, the more to sift through to deliver what users need.

Despite the fact that computer hardware is always getting faster, datasets are simultaneously getting larger to feed people's ever-growing appetite for sophisticated information. No matter how the technology improves, performance tuning skills will remain valuable when managing large geospatial datasets, especially when the delivery mechanism is the Internet. This column explains some of the issues involved in tuning a spatial system for optimal performance. If you haven't yet had to solve a data performance problem in the course of your spatial career, stick around; it will come.

Bravado's basis

The pseudo-religious zeal that flavors macho discussions about system performance stems in part from our very human tendency to talk the loudest when we know the least and are afraid our peers will discover our ignorance. Human frailty isn't the only culprit, though. True performance tuning begins as a complex art and only appears to be a science in retrospect. This art requires its practitioner to synthesize not only hardware components, but data structures and application design. Any complex art has plenty of room for debate even over simple problems, and the art of performance-tuning is no exception.

"An art?" you may ask, "but this is engineering, not watercolor painting!" But consider the nonlinear relationship between the number of CPUs and a system's performance. Upgrading your computer from one to two CPUs does not automatically double the performance speed of spatial applications. In fact, it may even degrade performance! Anyone who implies that his or her fast, large, or expensive hardware components are the sole source of a system's fast performance is unwittingly broadcasting his or her ignorance of the real issue: speed comes only after varying the configuration of hardware components, the data storage strategy, and the application code, followed by repeated testing, searches for bottlenecks, more testing, more changes, and so on, until all the pieces work together. Speed can't just be bought; it must be coaxed out of the system by a crafty, patient, artful mechanic.

Coaxing performance

The process of coaxing an optimal performance from a large spatial application involves balancing hardware quality, I/O channels, data structure and storage, and query optimization. Hardware quality is the easiest to understand and yet the most often misunderstood. Certainly, faster CPUs and more RAM can speed up processing, but that's only part of the picture. The way the mechanic configures a

Glossary

CPU: central processing unit

I/O: input/output

RAM: random access memory

Net Results columnist Jonathan W. Lowe is the owner of Local Knowledge Consulting (Berkeley, California), where he designs and implements spatial Web sites. Lowe can be contacted at info@giswebsite.com.

system to move information as the application runs, from hard drive to RAM to CPU to display and so on, can markedly influence an application's speed. Along the same lines, the mechanic's choices about how and where to store the spatial data will affect speed. Further complicating the tuning process, there are many ways to ask a computer for the same piece of information. Careful design of the order in which an application retrieves data can accelerate its response. Finally, lurking in the background of each of these aspects of the design process is the bottom line — money. In general, the fastest possible solutions usually carry the greatest expense, especially when data security is also important. As intimidating as messing around with low-level hardware can be at first, the concepts that spatial mechanics use are straightforward and simple. It's their synthesis that's the art. A few examples should illustrate the relative simplicity of a performance-tuner's portfolio of strategies.

Divide and conquer slow disk I/O

Spatial applications respond to peoples' requests for data. Zooming or panning a map window causes the application to search for the subset of map data within the extent chosen. In most spatial implementations, that map's data is stored on a hard drive. Too bad all the data can't be stored in RAM, where retrieval times are infinitesimal compared with disk access times. But even on really macho machines with several gigabytes of RAM, there's usually too much data, so it must be semi-permanently stored on a hard disk. The slowest process for a hard disk is seeking; that is, moving the mechanical access arm to the track on the spinning cylinder that holds the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts, and varies from zero to nearly a full second. One second may not sound like much, but what if every street segment in a map of your neighborhood took a second to seek on disk? A neighborhood rendering would take a while.

Spatial performance-tuners minimize seek time by loading the most popular subset of the data onto the middle of the hard drive cylinders. Then, no matter where the disk arm is at the moment of the data request, it will seldom have to travel more than halfway across the disk (at most) to get popular data. For spatial applications, this might mean loading the rural areas on the inner and outer rings of the cylinders and the urban ones in the middle.

Going up?

If two or more applications require access to the same hard disk simultaneously, I/O requests can be delayed, a situation called contention. The simple strategy for avoiding disk contention is to divide the data based on its probable use and spread the different divisions across separate hard disks and I/O channels.

This approach is similar to the design of elevators in tall office buildings. In a building with 30 stories, one elevator may service the first fifteen floors, and another, floors 16 through 30. Then, when two people arrive at the elevator bay at the same time, one going to the 14th floor and the other to the 20th, they can both ride their own elevator to their own floor directly. If one elevator existed to service all floors (a cheaper solution to build), the person going to the 20th floor would have to wait while the elevator first stopped at the 14th floor.

Disk stripping. The analogy is the same with a strategy called disk striping in which divided datasets reside on several drives. If the system can access each disk independently, then two requests for different parts of the same large dataset can be retrieved simultaneously without disk contention. In a dataset in the United States, this would mean storing eastern states on one disk, midwestern states on another, and western states on a third. (As with the elevator example, buying and configuring an array of individual disks will probably cost more than buying one larger disk.)

Mirroring. Taking the analogy further, what if the two elevators in our 30-story building both went to all floors? The same two people could still get to their destinations uninterrupted by taking separate elevators. This is the most costly elevator design, requiring complete duplication of all elevator features on all floors, but it is also the most versatile and flexible if one elevator goes out of service. The parallel in a computing environment is called a *mirrored drive array* or a redundant array of inexpensive drives. The mirroring strategy copies an entire dataset to two (or more) hard disks, but reads data from alternate clusters on both (or all) disks, cutting retrieval times by half for two drives, by two thirds for three drives, and so on. If a drive fails, the complete copy is still available on any of the other mirrored drives. Handy, but expensive.

Even with the best elevator design, our imaginary building might have a problem if there was only one narrow entry door to the lobby. In the morning, this single access point will be mobbed with people trying to get in. Better have several doors or a very wide single door. Similarly, multiple daisy-chained hard drives all connecting to the same computer through a single small computer systems interface port will underperform the same hard drives each connected to the CPU by separate ports.

Speed versus space?

"Disk is cheap," mutter the spatial mechanics, as they build indexes on large datasets. They're reassuring themselves about a dilemma they face for every spatial dataset — namely, to sacrifice storage space by building an index or allow performance to suffer?

Indexes are presorted lists describing columns of records in spatial datasets. Like an indexe in the back of a book, they speed up retreival time by preventing a complete scan of the entire dataset for every search request, but (like book indexes) they take up space. Without indexes, a mechanic has to find a way to speed up a complete data scan — upgrading the CPU speed is one alternative. Because buy-

ing extra disks costs less than buying more or faster CPUs, indexes are a wise choice financially, and are easy enough to build. As with base data, the "elevator" strategies also apply. If a big dataset resides on one disk and its index resides on a different disk, then disk contention and seek time can both be eliminated or reduced — one disk arm reads the index and another reads the data, neither arm moving much between seeks. A dataset with its index on the same cylinder forces its disk's arm to jump back and forth from index to data with each new seek, slowing down the response time. Also, like striping and mirroring of raw data, indexes can be intentionally divided and spread across multiple disks, a strategy called *index fragmentation*.

Too many cooks? Delegate courses

Suppose the budget is infinitely large, allowing for both extra disk space and more CPUs? Plunk several new CPUs into the system and, *voila!*, huge gains in performance, right? Maybe not. Performance might even get worse. But if a faster CPU means faster processing, why don't more CPUs also mean faster processing? In fact, multiple CPUs can process faster than single CPUs, but only if there are simultaneous multiple tasks to be processed. In other words, when a system gets a request for data, only one of its CPUs accepts responsibility for responding. Except on custom systems designed for very specific tasks, it's too difficult to break apart a single request, send its parts to separate CPUs, and then weave each CPU's response back together again at the end. Instead, gains from multiple CPUs start to appear when the system handles multiple simultaneous requests, a common scenario for an Internet server.

On a single-processor machine, for example, two requests for maps could arrive at once. The solitary CPU can only do one thing at a time, so if each request (or thread) requires one second to complete, it will take no less than two seconds to respond to both requests. A dual-processor machine in the same two-thread situation could finish both requests in only one second. The first CPU handles thread-1 in a second, and, simultaneously, the second CPU handles thread-2 in a second. At least, that's the lofty theory. In reality, things are not quite so cut-and-dried.

Multiprocessor machines include the overhead of managing all their processors, an ongoing task that requires threads (and a processor) of its own. At some point, the gains of multiple CPUs outweigh the loss due to overhead; multiprocessor systems are best suited to multiuser environments.

Not just how many, but how

No matter what the system's size or the project's budget, there's always a place for the design of the data and the cleverness of the query. Duplicating the same spatial theme in different formats is one clever trick for speeding up retrieval and display. For instance, if the freeway theme includes on- and off-ramps that can only be seen below 1:50,000 scale, store a second copy of the freeway data that does not include the ramps, and use this smaller dataset when drawing the freeways at scales greater than 1:50,000. Similarly, the accuracy of the freeway lines may be critical at scales below 1:24,000, but impossible to even recognize when zoomed out any farther. Storing a generalized copy of the freeways that drops every other verticy or even every other node of each freeway segment will dramatically reduce the copy's size and speed up its rendering time. At scales above 1:24,000, the generalization will be undetectable if done properly. (Of course, storing copies of data requires more disk space, and so incurs an expense.)

Query optimization. Data designs like these speed up the responses to spatial questions. It's also possible to tinker with the questions themselves and, though the answers may be the same, to improve response times even more. The art of query optimization relies on smart decisions about the order in which related datasets are filtered prior to being compared against each other. For example, imagine a U.S. census application that lets users ask for the number of homes worth more than a given dollar value in tracts within a certain population density range. The attribute-only census data contains records of home value and population. The spatial tract data contains records of tract area. Neither dataset has precalculated values for population per tract area. The population density, then, must be calculated on-the-fly — a big time sink.

Although the users select their own settings for population density and home value, the application designer decides how to execute the search, which involves four parts — calculating population density, filtering by population density, filtering by home value, and counting. Will the query's order of execution influence the retrieval speed? Absolutely.

One of the tricks of query optimization is to narrow the selection set as early as possible in the search sequence. That way, subsequent steps involve a smaller dataset and hence a shorter seek time. In the census application, when a user requests the number of homes worth more than $1 million in tracts with population densities less than 2,000 people per square mile, a clever query would first eliminate tracts with home values less than $1 million. (Hopefully, the home value records are indexed and so will be almost instantaneously filtered.) Now the query process can join this small subset of census records to the tract dataset and perform the time-expensive, on-the-fly density calculation, its filter, and then the count. If the query was designed to first join, calculate, and filter by population density, then the system would needlessly crunch through every value in both tables for every user request.

There are times when even an optimized query just isn't fast enough. In this census example, it may be worth adding a permanent new set of records for population density and indexing them. The extra space required for the new records and their index could be costly, but justified by

the increase in user satisfaction when the answers come back more quickly.

The dove is in the details

When people get into car accidents, they often say, "He hit me," rather than "His car hit my car." Our cars become extensions of our bodies when we drive, both physically and psychologically. We don't "become one" with our computers like we do with cars, but some hardcore techies subconsciously equate the power of their systems with their own personal competence as human beings. No wonder the hardware bravado battles get so heated.

Next time you're caught in the crossfire of a verbal hardware volley, do your part to restore peace to the world of spatial computing with some healing performance art. In response to the comment, "Dude, you could smell the smoke from my two 800 MHz processors when I rendered every street in the country in 20 seconds!" just say, "Oh cool, man, so, did you normalize the spatial and attribute values into separate relational tables in the database? Or did you decide to stripe the data across three spindles with a fourth independently connected disk capturing log entries, or just mirror the whole thing? Were you using a fragmented index? Did you limit the display at a countrywide scale to rampless freeways only, or duplicate and thin them with a generalization? See, I'm just a newbie at this performance-tuning stuff, so I want to learn from pros like you."

Mystified by your refinement, the macho squadron may suddenly recall other meetings they ought to be attending and pause the warfare until another day. Maybe they'll reengage after they've tried performance-tuning for themselves. If so, what they gain in expertise and experience may begin to release them from the urge to connect their value as people with the power of their hardware devices. Organizational (human) performance-tuning will always demand more sophistication than any artful tinkering with a computer system ever could. We people are so darned complicated. Hopefully our machines will always agree, free of bravado, to perform ever faster at the artful coaxing of a well-intentioned spatial mechanic.