Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Farewell to Disks

Diomidis Spinellis

A classic Web comic illustrates how idle Wikipedia browsing can lead us from the Tacoma Narrows Bridge to Fatal Hilarity (and worse; see http://xkcd.com/214). The comic doesn't show the path leading from A to B, and finding it is an interesting challenge—think how you would engineer a system that could answer such questions. I believe that this problem and a solution I'll present here demonstrate some programming tools and techniques that will become increasingly important in the years to come.

Challenges

Questions whose answer requires sophisticated processing of huge datasets come up increasingly often in our networked, interlinked, and (increasingly) DNA-sequenced world. Attacking such problems with traditional techniques, such as loading data into memory for processing or querying a relational database, is cumbersome and inefficient because during recent years, several developments have conspired to create a perfect storm.

First, dataset sizes that used to be seen only in high-end scientific computations are turning up all around us. Facebook has half a billion active users, Wikipedia's graph contains 95 million links, the Netflix Prize competition involved 100 million movie ratings, Twitter records billions of tweets each year, and DNA pyrosequencing platforms generate 40 million nucleotide data in an hour.

All this data must be stored persistently on disk files, yet the disk-based data structures we currently employ are often inadequate for running sophisticated algorithms on them. A relational database is perfect for storing a company's ledger but falls short when we need to perform something as simple as a graph traversal, let alone recommend a movie, group related friends, or patch together DNA fragments.

A major stumbling block is performance. The file systems and databases we use are optimized to handle sequential retrievals and relational joins but inadequate when it comes to optimizing data structures for running more complex algorithms. At best, some systems let us view our disk-based data as key value pairs, often forcing us to pay the penalty of a few system calls for the privilege of each such access. Huge datasets, abysmal disk throughput, and application-specific access patterns require us to optimize data access operations in ways that are beyond what a relational database or even a noSQL system can offer. The current situation, where fast RAM is used to cache and buffer data structures stored on slow disks, is clearly a case of putting the cart before the horse.

Technologies

Surprisingly, very few modern languages, libraries, and applications take advantage of the advances made in computer architecture and operating systems during the past three decades. Granted, the designers of those innovations go to extreme lengths to hide their complexity by maintaining backward compatibility with older systems. However, the end result is that, barring CPU speed and memory size limitations, most of today's code could run unaltered on a 1970s-era PDP-11. Yet we're now in a position where we can perform all our processing with readable and efficient RAM-based algorithms, using clunky disks and file systems only for their large capacity and to secure the data's persistence.

0740-7459/10/$26.00 © 2010 IEEE

Modern 64-bit architectures allow us to address in RAM all the data our application could ever need. Therefore, there's no need to maintain separate data structures on disk. Doing away with disk-based structures saves us from using expensive seek and read system calls to access them. Even better, in RAM, we can use powerful abstractions, such as objects and pointers, rather than error-prone, weakly-typed keys and file offsets.

In addition, memory-mapped files can secure the persistence of our RAM data while virtual memory lets us cope with data sizes larger than the available RAM. Both are features provided by all modern operating systems, allowing us to map the contents of an arbitrarily large file into a process's address space. When our program reads data in that area, it's paged-in on demand from the file; when our program stores something, the RAM's contents are (eventually) transferred back to the file. The processor's memory management unit and the operating system's paging mechanism cooperate so that all this happens transparently without requiring us to litter our code with pesky disk read/write operations. Clever caching algorithms ensure that the data we most need is available in RAM, while the rest is stored on the disk.

Current implementations of memory mapping require us to preallocate all the space our application might need. Again, however, this isn't a problem for modern operating systems because through sparse-file implementations they can store efficiently huge empty files. The operating system stores on disk only the portions of the file that are filled with actual data and provides our application with the illusion that the rest of the file contains zeroes. For example, an empty half-terabyte file occupies just 4 Kbytes on a Windows NTFS and 28 Kbytes on a Linux ext3 file system.

When we want to share memory-mapped files among many processes, we can select between three efficient alternatives. If we can do with read-only access, the hardware can enforce this constraint, ensuring that no process cheats. The operating system can also arrange a shared memory setup, in which all processes can concurrently view and modify the same data. A more sophisticated "copy on write" scheme shares the data among the processes while providing each one with the illusion of operating on its unique copy. In this case, the hardware detects write operations and the operating system clones the corresponding pages to an area private to the process that performed the write. Copy-on-write is an extremely powerful feature that we can use to create data snapshots and ensure the data's consistency.

In Practice

Luckily, modern programming languages and platforms are powerful enough to abstract the influential technologies I've described in a way that makes them transparent to the application developer. On the column's blog site (www.spinellis.gr/tools), you'll find a short program that can store Wikipedia's graph on a persistent disk image and then allow you to search for the shortest path among two entries, thus answering the challenge I described in the introduction. Although the program uses the disk for persistence and to overcome the RAM-size limitations of the machine I'm using, it operates as if the data were transiently stored in RAM. The program's two key parts, the loop for storing the graph on disk and the graph's breadth-first-search algorithm, don't contain a single reference to files or records.

I've written the program in C++ using existing Boost and Standard Template Library features to provide the functionality I've described. This demonstrates that all the facilities for programming in the way I advocate in this column are already available. Specifically, I'm using the Boost. Interprocess library to create a memory allocation pool backed by a memory-mapped file. I'm then passing the pool's allocator as a type parameter to a string class used for storing the node names, a set container that stores the nodes, and a linked list containing the edges. This parameterization makes all the graph's data reside transparently on disk. Internally, the containers abstract the pointers they're using into an offset-based implementation so that the backing file can be mapped on an arbitrary memory address between successive process invocations. A Java-based implementation of the approach I'm describing might require changes to the underlying virtual machine but would be even more transparent to the application programmer.

Barring CPU speed and memory size limitations, most of today's code could run unaltered on a 1970s-era PDP-11.

Processing all our data in RAM opens again many problems that database systems already solve: the atomicity, consistency, isolation, and durability properties of the transactions; the management of data and its schema using a standardized language; the portability of data between diverse architectures and applications; and the implementation of concurrency, distributed processing, and ingenious performance optimizations. Yet I believe that this is the right move because it provides us with a unified programming and performance model for all our data operations irrespective of where the data resides. Advanced programming language features will be usable for manipulating all data in a powerful and safe manner, while data structures and optimizations will be directly available to all programmers. Crucially, implementing RAM-based data structures with good data locality properties will benefit not only disk-backed stores but also boost the utilization of the processor's L1 and L2 caches.

Although I can't change the way we handle our data with a two-page column, I hope I've whetted your appetite to consider a RAM-based approach the next time you encounter a problem that cries for it.

Diomidis Spinellis is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business. Currently, he serves as the Secretary General responsible for information systems at the Greek Ministry of Finance. Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**