Category Overview

8/14/07

Recollection

Getting it back later

Given the mysteries and fascinations of algorithms and computation, it is easy to forget a simple and basic truth: all computations take place in storage systems. Every representation is inscribed in a medium, from which it can be recalled later. The response of the medium affects computation time as much as the logic of an algorithm.

The wide use of inscription to record human knowledge is relatively recent in human history. The practice became widespread after the printing press around 1450. Before then, oral traditions dominated the way our ancestors kept and transmitted their knowledge. Today, everything is in writing, from ads to contracts, screen storyboards to film, musical scores to symphonies, blueprints to building inspector reports. With the computing age we have moved extensively to recordings in digital magnetic and optical media. The size of storage systems has grown from single computers with databases to the entire Internet.

Because of the richness of the kinds of objects we store, retrieve, and share, and the types of media and systems that hold them, recollection rests on a rich trove of principles that deeply affect our ability to compute and our perceptions of what computing accomplishes.

Falling Flat

The arrangement of data in storage significantly affects the performance of the algorithm using the data. For example, if the data are N items arranged in a random order, the time to find a particular item will be proportional to N. But if the N items are arranged in increasing order, the search time drops to order log N.

Most of complexity theory assumes that the data are laid arranged in a “flat” medium that stores and retrieves any item in the same amount

of time. For flat storage, counting an algorithm's steps is a reasonable way to assess its running time.

But what happens when the storage medium is not flat? When some items take significantly longer than others to retrieve? This is not a hypothetical question because almost all storage systems are not flat, and in fact only a small fraction of the total storage in a system is flat. Most real computations don't take place in flat memories.

It is not enough to think of storage simply as a memory component of a computer, for then we will be limited to thinking in terms of bytes and words. Storage systems deal with many complex objects including files, documents, folders, records, images, sounds, movies, Web sites, and more. The Internet itself looks like a global storage system encompassing billions of Web pages linked to hundreds of millions of such objects. Storage systems contain a variety of media such as RAM, flash memories, hard disks, portable disks, CDs, DVDs, and tapes, each having its own access time and transfer time for a typical object.

Consider what happens when we take a straightforward arithmetic problem into a non-flat memory. Multiplying two NxN matrices in a flat memory takes time proportional to N^3. The algorithm needs $3N^2$ memory cells to hold the two input and one output matrices. Suppose we have matrices so large that the flat part of memory can only hold a single row or column from each input matrix. In this system, each output element will require a load operation for a row or column of an input matrix, for a total of N^2 loads. Because the load operations can take much longer than individual instructions, the total loading cost can easily dwarf the computational cost. To demonstrate, consider a hypothetical weather forecasting problem using a grid of 4K ($N=2^{12}$) points on a side. The computation for a matrix multiply would then be about 2^{36} operations. A gigaops ($10^9 = 2^{30}$ operations/sec) desktop computer would take about $2^6 = 64$ seconds for the computation. However, in the constrained memory, it would need about $2^{24} = 10^{11}$ load operations. If the loads came from a disk that takes 10 milliseconds per row or column, the total load time would be about 10^9 seconds, or about 30 years. Yikes! The cost of loading far exceeds the cost of computing.

Thus our first principle of recollection is that storage systems are not flat. They have hierarchical structures. Hierarchical structure significantly affects performance.

Storage Hierarchies

Every storage system is arranged as a hierarchy with fast-access storage at the top and slow-access at the bottom. For example, a traditional library has a reading room, a warehouse-like space with many books on shelves (the stacks), and off-site space for little used books (the archive). You bring books to the reading from the stacks and return them, and the librarian moves books between the archive and the stacks. A similar organization exists with company and home reference document libraries, music libraries, photos, and indeed almost anything that is stored and retrieved.

Computer storage systems are no different. A storage system consists of a collection of devices of different media, each of its own price, access time, and transfer rate. The top of the hierarchy is called the "main store" or "computational store"; it holds the data directly accessible to the CPU.

Many storage systems (such as the Internet) are shared among many users. Each pulls data from the storage system into its local RAM (random access memory). Thus each user sees a unique hierarchy, with his own RAM at the top.

CPUs can only process data that are in the local RAM. The RAM on memory chips is the most common type. Modern RAM is very fast, with access times on the order of 10^{-9} second, enough to keep up with the fastest CPUs. However, RAM technology is volatile (non-persistent), meaning that if power goes off memory is lost. (RAM technology was not always volatile. Magnetic cores, common in the 1960s, were not volatile, but they were thousands of times slower and more bulky and expensive per bit than today's memory chips.)

The rest of a storage hierarchy is called "secondary store" or "long-term store". They are persistent media, retaining data indefinitely until they are explicitly erased. Modern examples are disks, CDs, flash memories, and tapes. Older examples were disks, drums, and in the early computers, acoustic delay lines. Many secondary media are portable, such as floppy disks and zip disks.

Some of the technologies (notably RAM and flash) have the same access time for every location; the total time to move a block of n bytes is of the form $T(n) = An$, where A is the location access time. Other technologies (notably disks, CDs, tapes) have a latency time in addition to the block transfer time. Latency is the mechanical time to position the media or the read-write heads. The total time to move a block of n bytes is of the form $T(n) = An+B$, where A is the time to read successive bytes from the moving medium and B is the latency.

For a typical RAM, access time is 1 nanosecond. For typical disk, latency is 5 msec and transfer time another 5 msec. The speed gap is about 10^6 -- the processor can execute a million instructions in the time required to retrieve a block from disk.

The cost per bit of main store is much higher than for secondary store. The differences can be quite pronounced. Today a 100 gigabyte (GB) disk or a 1 GB RAM costs $200; the RAM is 100 times more expensive per bit than disk. The disk has access time around 10^{-3} second compared to the RAM's 10^{-9} second, a gap of 10^6 RAM cycles for one disk access. The same amount of computational core storage in 1960 would cost about $250 million. Memory price has dropped by a factor of 1 million over 4 decades.

Many local RAMs are augmented with a cache, a small amount of ultra-fast, expensive RAM. The cache can run at the CPU speed, but the RAM cannot. The cache therefore allows the CPU to run faster. Here is a summary of the relative speeds and costs of storage devices:

	Access time	Cost per bit
Cache	10^{-3}	10^3
RAM	1	1
Disk	10^6	10^{-2}

A measure called hit ratio tells the percentages of CPU references that land on objects loaded in RAM. Even when the hit ratio seems high, the gap between primary and secondary store makes the effective speed considerably slower than the CPU. For example a 99.99% hit ratio gives an effective memory access time of $(1)(0.9999)+(10^6)(0.0001) = 100$ RAM cycles; in other words, even with such a high hit ratio, the CPU runs at an average rate of 1/100 of the actual RAM speed.

Managers of libraries have to make similar calculations when they decide how large the reading room is, how far from the stacks it is, how large the stacks are, and how far away the archives are. Amazon.com, the on-line bookstore, keeps its most popular books in warehouses ready to ship. The less popular books are stored by their printers, who fulfill orders forwarded from Amazon.com.

For these reasons, storage system designers are under a strong pressure to minimize the number of accesses to secondary storage and maximize the number of useful items moved to or from main store with each access.

Locality

Imagine yourself in a library, watching people using it. A person working on a project locates several books in the stacks and brings them to the reading room. There he reads sections of the books and switches attention from one book to another. Occasionally he rises, returns books to the stacks, and obtains additional books from the stacks. He does a lot of work with the books in the reading room in between the times he visits the stacks. Most of his attention is focused on the books in the reading room.

The reader's behavior is called "locality". He has localized his attention on a small subset of the library and keeps it on that subset for an extended time. Locality means that a person or a computation directs all memory accesses to a small subset of objects for extended periods.

Locality is what makes the reading room effective. If every minute the reader selected a next book at random from the whole library, the reading room would be an impediment. It would be better to let the reader wander through the stacks, going directly to the books needed at the times needed. Even so, the total time to read all the needed books in this case would be much larger than if he localized his reading.

Reading a book also illustrates the locality pattern. The reader limits attention to the pages of the book. While reading the book, the reader normally does not make random references to pages of other books.

Another example of locality behavior can be found in biological systems. A subsystem such as an organ or section of the brain receives an input that triggers actions inside the subsystem. Many internal actions will occur before the subsystem returns a signal to its external stimulator. Most of the state changes are localized for long intervals between interactions with the external environment.

The first kind of locality, where people cluster their accesses to stored objects within time intervals, is called temporal locality. It is motivated by a desire for efficiency because every switch of attention adds to the work of completing all the accesses. The second kind of locality is called spatial locality. The next accesses are close to the current ones because they are for related objects stored nearby; in other words, given that you enter a neighborhood it is highly likely

that you will access many members of the neighborhood before leaving.

The situation inside a storage system of a computer is no different. Computations, which act on behalf of human users, exhibit temporal and spatial locality. The storage system can take advantage of temporal locality by placing the most recently accessed objects at the top of the hierarchy for quick access. It can take advantage of spatial locality by storing related data in the same blocks of secondary memory.

We have made this precise inside a computing system by defining the "working set" of a computation at a given time as the set of objects referenced (read or written) during a time window into the recent past. The storage system will be most efficient when it measures the working sets of computations and places them at the top of the hierarchy.

In the early days of computing, we measured reference maps to convince ourselves that computations really do have locality behavior. A reference map had a horizontal axis corresponding to execution time of the program, and a vertical axis corresponding to the addressable objects. A point on horizontal axis represented an interval of execution equal to a short working-set window, say 100 milliseconds; a point on the vertical axis represented one object. A column of bits above a time point displayed the working set at that time, black dots for objects in the working set and white dots for objects not in. Reference maps showed substantial intervals of constant behavior and extensive white spaces in striking patterns. They were sometimes used as "fingerprints" to uniquely identify programs. They were clear visual illustrations of locality.

Early programmers also constructed reference maps manually to plan which objects needed to be in main memory during each phase of execution. (A phase of execution is an interval of constant working set.) At the end of a phase, programmers inserted commands to move blocks of data into and out of main memory.

In 1959, the designers of the Atlas Operating System at the University of Manchester in England proposed to automate the process of moving blocks of data. Their memory architecture came to be called virtual memory. The proposal was initially controversial because most programmers believed they could manually achieve lower data rates than the Atlas mechanism. By 1969, however, researchers at IBM showed not only that virtual memory would do as well or better than the best programmers, but also that it doubled programmer productivity by relieving them of the work of planning block transfers.

In other words, locality behavior was sufficiently pronounced that the operating system could exploit it to automatically manage the placement of data in the hierarchy, more efficiently than the best programmers. Experience has shown that virtual memories that retain working sets in main memory perform the best.

The locality principle implies that caching is an efficient way to manage objects needed by a user or computation. Recall that a cache is a storage device that holds a small number of objects physically very close to the user (or the CPU executing the computation). The ideal cache holds the user's working set. It guarantees the user experiences minimal delay to obtain objects from the memory hierarchy. With a relatively small cache, a storage system can achieve operating speeds not much slower than the cache speed at a total cost not much larger than the RAM cost.

The caching principle is used in nearly every computing technology. Computers put a cache memory next to the CPU chip so that most of the time the CPU will find the data it needs nearby and can proceed at maximum speed. Local networks cache recently-used web pages on a local server, bypassing a long access time to a remote server for many web page lookups. Web browsers retain copies of web pages viewed most recently. Application programs retain menus of the files most recently referenced. Email programs retain lists of the email addresses used most recently. Video display cards retain a copy of the video image, updating only the changed bits rather than regenerating the entire image.

Caching is as common outside computing systems as inside. The reading room in a library is a cache. Your desktop at home or work is a cache. Your knapsack is a cache.

Some people like to speculate that eventually our ability to build memory technology will be so good that making the entire memory space "flat" will be feasible. In such a world, all the complications of memory hierarchy would disappear. The principle of locality tells us this is wishful thinking. Some objects in the very large memory will be much more popular than others; all the users seeking access to the popular objects will queue up at the servers holding those objects and will experience significant congestion delays. Users will not see a flat memory at all.

Thrashing

Psychologists have measured human productivity as a function of the rate of requests coming to a person. As the request rate rises,

productivity increases toward a saturation limit. But if the request rate rises beyond a critical threshold, productivity will drop suddenly and become much less than the person's saturation limit. The critical threshold value is not predictable. The explanation is that the brain gets working so hard to process the incoming requests that it can no longer attend to the work of answering the requests. We call this behavior "information overload" and say that a person in the low productivity state is "thrashing".

A similar thing happens in a waiting line for a popular service. If the rate of arrivals gets too large the management of the service gets diverted from serving to trying to maintain order in the waiting room. The output of the service drops and the customers say it is thrashing.

Storage systems are subject to thrashing as well. The problem was first observed in the first multiprogramming systems in the 1960s. Those systems assigned each user a share of the main memory. When the share is smaller that the user's program, the program will be forced to stop from time to time to request a data block transfer from the disk system. Each incoming block displaces an already-loaded block, which must be recalled in the future. As the number of logged-in users increases, the individual shares of memory become smaller and the recall traffic becomes more intense. Eventually, almost all the user processes are waiting in queue for the secondary memory system to respond to their recall requests. From the user standpoint, the system appears to stop processing.

In those operating systems, thrashing became a multimillion dollar liability. If the computing system suddenly stopped (because it was thrashing), its users will be quite dissatisfied. Dissatisfied customers don't buy million-dollar computing systems. The computer companies had to solve the thrashing problem or go bust.

The same problem was encountered in the 1970s in the construction of a packet-radio system called Aloha. This was the first system to extend the ARPANET packet switching idea to radio communications. When a transmitter wanted to send a packet to a receiver, it listened on its assigned frequency and waited until no one else was transmitting; then it sent its packet. If it discovered that another transmitter had started at about the same time -- the transmission is jammed -- it stopped and retried. The thrashing problem arose when too many transmitters tried to send packets at once: they all wound up "spinning" in their retry loops and no data were transmitted.

A 1990s version of the problem appeared in the World Wide Web. Popular web servers got overloaded and their response times collapsed. Merchants who ran these servers found that customers

would disappear rather than wait for the congestion to abate. Web designers addressed this problem with "mirror servers" which were copies of the original server located close to a set of users. The user load was distributed among the mirror servers and did not force any of them into thrashing.

The solution in all these cases is some sort of load-limiting feedback control system. For multiprogramming, the key idea was to measure each user's working set, granting a new process a share of memory only when there is sufficient space for its working set. For packet transmission, the key idea was to insert a delay into the retry loop, growing progressively longer with each retry, thereby limiting the number of simultaneous contenders for the frequency band. (This protocol, called CDMA for collision-detection multiple access, is used in Ethernets and cell phone networks today.) For web page access, the mirror servers distributed the load and no one of them reached its thrashing threshold.

A conclusion is that failure to honor the locality principle will expose a system to thrashing.

Names and Addresses

An essential part of every storage system is a means of naming objects and assigning them addresses in the storage system. All operations in the storage system -- reading, writing, relocating in the hierarchy, searching -- rely on naming and addressing.

A name is a string of letters or numbers chosen by someone to refer to an object. An address is a string of letters or numbers that refers to a physical location in which an object is stored. The postal service illustrates these distinctions: letters are intended for a named person who can be found at the given address. If there is no such person at the address, or no such address, the letter will be returned to sender.

Names and addresses are distinct because different authorities control them. Individuals select names based on local, family, and historical reasons. Jurisdictional managers select addresses based on considerations of efficiency and capacity of their spaces. Because they are not the same, there must be a mapping that connects names to addresses. The postal service, for example, maintains a database of names and addresses and uses address change forms to change those associations.

Over the past century, the expansion of communication systems has enabled the range of selectable names and addressable spaces to become global. There are potentially an infinite number of names and

very large (but finite) numbers of addresses. In the process of managing such large spaces, the designers of these systems have had to consider how to accommodate various combinations of these three objectives:

- **Location independence.** Objects in the system can be addressed the same way regardless of their physical location. Their physical locations can change without changing their addresses. In other words, the address names an abstract representation of a location, but not any location in particular. Examples: (1) The cell phone system allows you to keep the same cell phone number for life and no matter where you live. Ditto for Internet Voice over IP telephony. (2) Internet IP addresses can be attached to different computers at different times. (3) The federal government provides military and state department personnel with APO addresses; send a letter care of the APO, and the APO postmaster forwards the letter to the addressee at their currently assigned location.
- **One-time global names.** Every object receives a unique, unchangeable global name called a "handle". New versions of an object receive new handles. Handles ensure that sharing is easy and care-free. Those who link to a Web page or a file have an expectation that the Web page or file will remain the same indefinitely. Renaming the Web page or making a substantive change in the file violates those expectations and can cause unexpected problems for the people who link. Examples: (1) The Library of Congress Digital Object Identifier system allows publishers to assign unique global names to all their publications, so that they can be found at any time in the future no matter where the publisher has stored them in the Internet. (2) County clerks issue unique record numbers for real estate transactions, birth certificates, and death certificates. (3) Libraries use the Dewey Decimal system to classify books and assign them permanent shelf space. (4) Publishers use ISBN identifiers on books. (5) The government issues social security numbers for individuals. (6) The motor vehicle department issues driver license numbers.
- **User-selected names.** Owners of objects and accounts want control over the names they assign to objects and accounts. They want names meaningful to them in their contexts of use. They want to work within naming systems that ensure that their local choices have a global meaning. They want simple ways to work within potentially very large name spaces. Examples: (1) Families choose names for their children. (2) Individuals choose

names for their accounts on Web sites, chat rooms, and mail servers. (3) Individuals choose names for objects in their personal directories of a computer system. (4) In the Internet, a locally chosen name becomes part of a global naming scheme by concatenating the name of the computer with the file's path name on the computer; the result is called the Uniform Resource Locator, URL. (5) In the Internet, host names are selected by hierarchical domain naming authorities.

These three objectives are independent; any one can be present or absent without jeopardizing the other two. For example, handles and names can be attached to locations, achieving the second two objectives without the first. Or, there are no handles and user names designate addresses, achieving the first and third objectives without the second. Or the authorities select names and identifiers for users, achieving the first and second objectives without the third.

Dynamic Bindings

Storage systems are used in all the applications discussed above, and more. To accommodate all the objectives listed above, storage systems manage four categories of identifiers:

(1) **Names** are user-chosen symbolic names for objects. Names are important to users, who want to choose identifying strings that mean something to them and are easy to remember.

(2) **Handles** are system-chosen, globally unique identifiers for objects. Handles are important for sharing; each object gets a unique identifier. User-chosen names are not reliable for this purpose because users may use the same name for different objects, and they may choose non-sharable private names for objects they later decide to share. One way to form a handle is to join a time stamp with the identifier of the machine creating an object. Another is to augment a name with a version number or time stamp.

(3) **Addresses** are bit-strings identifying individual bytes of an executable program. Addresses identify all the bytes in compiled programs and in the data used by those programs.

(4) **Locations** are bit-strings identifying physical places on devices where data items are stored. Locations are physical sites for data.

It was implicit in the descriptions of the storage system objectives that there must be mappings that associate each kind of identify ultimately

with a location. Memory systems make these mappings explicit. There are three, denoted by arrows in the series:

name → handle → address → location

This can be visualized with three system-maintained tables: one associates a user's names with internal handles, another associates handles with addresses, and the third associates addresses with locations. These tables can be dynamically updated.

The chart below illustrates these ideas. Individuals have family names (first, last) chosen by their parents. They have social security numbers assigned by the government. If they are in the military or diplomatic services, their postal mail is sent to an APO, which forwards to the current (possibly confidential) location. They also have cell phone numbers and emails that connect to them but are not tied to physical places or objects. The various entities who manage these types of identifiers also manage the mappings between the levels.

Names	Family names
Handles	SSN
Addresses	APO, cell phone number, email
Locations	Places

When a user creates a new object in a computer system, the system assigns the object a handle and binds it to the name chosen by the user; the system also assigns a set of addresses to the object and binds them to the handle; and finally it binds those addresses to physical locations on a storage device. All these bindings can be adjusted as the system evolves. For example, if an object is moved to a new position in the memory hierarchy, only the address-location binding needs to be updated. If a shared object is moved to new virtual addresses, only the handle-address binding needs to be updated. If a user renames an object, only the name-handle binding needs to be updated.

All the situations encountered in storage systems about object location, sharing, and naming can be accounted for as manipulations of these three dynamic bindings.

Examples of the Bindings

The virtual memory system (first introduced on the Atlas computer system in 1959) focuses mainly on the address-location binding. There are no handles; the compiler binds program names directly to addresses. Only the address-location binding is dynamic. Executing programs request data block moves from disk to RAM, forcing other blocks back to disk. Each such move is accounted for with simple updates to the address-location tables.

Another prominent example of the address-to-location binding is the storage of files. A file looks like a sequence of bytes. Many users think that the sequence is stored contiguously on a disk. In fact, the file is broken into blocks that are stored on disk sectors. A file's blocks can be scattered across the disk. To find them, we create a File Access Table (FAT) that maps file blocks to disk sectors.

Although the disk address of the FAT is a possible file identifier, it is subject to change if the FAT is moved. So we create a one-time global bit string to identify the file -- a handle -- and a table that maps handles to FATs. Then the FAT can be moved without changing the handle.

Finally, we store the handle in a directory along with the symbolic file name assigned by the user. The user can read or write the file by providing its symbolic name; the system uses the directory to map to the file's handle, the handle map to get the FAT, and the FAT to get all the blocks of the file.

How does a file become part of an address space? One common method is to have an executing computation "open" the file. This causes the operating system to temporarily bind the computation to the file. The binding is erased when the computation "closes" the file.

Another way to get a file into an address space is to copy it into a fixed set of addresses reserved for the purpose; this is called static binding. Still another way is to use a special program called a "loader" or "linker" that converts cross-references to the file into addresses; this also is a static binding.

The basic Internet uses just two of these three types of bindings. Computer owners chose host names, which are bound to IP addresses by the Domain Name Service (DNS). IP addresses are bound to physical machines by the IP protocol. On those computers, file owners create pathnames. The Internet URL is formed by joining the hostname with the pathname. Unfortunately, this is not very reliable for people who want to establish durable links to objects. There is nothing to prevent the computer owner from changing the hostname

or the file owner from changing the pathname (or deleting the file), whereupon the URL held by another is invalid and will not work.

Publishers and librarians are especially concerned about this. Suppose the publisher of an article on page 15 of the April 2007 issue of *Scientific American* stores a copy at the pathname sciam/2007/apr/p15. If the publisher later wanted to introduce a new category called readers to augment its magazines, it would want to rename the old article sciam/mag/2007/apr/p15 and the first reader of its new series sciam/reader/2007/No1. The URLs therefore are not a durable citation. The solution to this problem is the digital object identifier (DOI), a method of allowing each publisher to create a one-time unique string for each published article. All publication citations are to the DOI, not the URL. A special server translates DOIs to their current URLs. The DOI is a way to add handles to the Internet.

Most email systems have an alias list that maps user-assigned nicknames to email addresses. For example, the nickname "pjd" might map to "peter.denning@nps.edu". As the user types the nickname, the mailer converts it to the real email address. Mail aliases are a dynamic name-address binding with no handles.

In any dynamic binding system, the overhead of looking up the bindings can slow computations down significantly. A solution is to augment the mapping table with a small, fast cache that stores the most recent translations and allows the CPU to retrieve them very fast without a table lookup. This has proved to be a very effective method of making dynamic bindings fast, as we would expect from the principle of locality.

Hierarchical Naming

Let us return to the issue of user-chosen names. Users have to deal with tens of thousands of file names on their own systems, and (potentially) billions of names in the full Internet. The hierarchical naming principle is a way to construct a local name that is unique within the entire space.

The idea is to delegate naming to a hierarchy of naming authorities within a tree. Each authority can assign nonconflicting names to objects at its level. The global name is formed by prefixing the sequence of authorities on the object's chain of command. These global names are called pathnames because they are the labels of a path in the authority tree from the root to the object.

The advantage of this system is that each naming authority has to deal with only a limited number of objects and their names. Simply by

assuring no name conflicts in their limited spaces, they guarantee that whatever name they choose will be part of a unique global pathname. Without a hierarchical structure, the authority that permits names would have to search the entire space of billions of names before it could determine with a proposed new name conflicts with any existing name.

The world telephone system is hierarchical. An international body chooses country codes, a national authority area codes within a country, a regional body exchanges within the area code, and a local office the assignment of numbers to individuals. The postal system is similar.

In computers the directory structure is hierarchical. The system administrator (root) can create new user names and give each user a directory. Each user can choose file names and place them in the directory. The user can also define subdirectories and act as the authority that chooses names within them. In the resulting tree, directories are internal nodes and files are leaves. A pathname is the sequence of labels from the root to the named directory or file.

The Internet host-naming system is also hierarchical. An international authority (ICANN, the International Corporation for Assigned Names and Numbers) determines the top-level domains and their authorities. There are seven generic top-level domains (.com, .edu, .gov, .int, .mil, .net, .org) and several more specialized top-level domains. The domain registrars add new organizations to the domains after verifying that their names are not already used, and then authorizes them to be responsible for their subdomain names. Organizations can define subdomains at their discretion. A host's name is the concatenation of the domain names surrounding the host, with the host's local name. For example www.nps.edu is the web server of the nps subdomain within the edu domain.

Handles, Sharing, and Reuse

The reusability of pathnames in a hierarchical naming system sometimes creates a problem. This problem is familiar to telephone customers who get phone calls for the previous owner of their newly assigned number. In the Internet, a user can create a file, give the pathname to friends, and then later delete the file. And then even later, the same user can create a different file and reuse the same old pathname. The friend who remembers the URL can be in for a nasty surprise on discovering that the URL no longer designates the file originally promised.

This issue is solved if every object in the system can be given a unique, permanent, never-reused name. This is not hard to do. The local machine on which a file is created already has a unique numerical address in the Internet. Concatenate that with a time-stamp from the local computer's clock. Since at most one file creation can occur in one clock tick, the resulting number will not be used for any other file. These long bit strings are called handles.

While handles solve the reuse and sharing problems, they usually consist of a large number of bits that have no meaning to humans. (For example, a handle might be 128 bits, compared with 32 bits for a normal address.) This is why the handles are stored in directories: users never see them and are not responsible for maintaining their integrity.

The conclusion is that handles provide a powerful means to solve sharing and reuse problems that otherwise exist in hierarchical naming systems.

Retrieval

We put data objects into storage systems because we want to retrieve them later. What if we don't know the names but want to retrieve all objects that have content we specify? Can we design the way we represent objects and arrange them in the storage system to facilitate retrieval?

A simple example will illustrate content-based retrieval. An ordinary phone book is an alphabetic list of names and their associated phone numbers. We can retrieve a phone number quickly by opening the book to the section containing the name of interest and then scanning down the names until we find a match. The phone book is ideally arranged for fast retrieval of phone numbers given the names. The situation is much harder if we know the phone number and want to find the name. In that case, we are forced to scan all the phone numbers from the beginning until we find a match. In the worst case, we will have to scan the entire phone book before we know whether or not there is a match. Because reverse lookup is common, we first create a copy of the file sorted by phone numbers.

One of the earliest uses of large storage systems was for "information retrieval." These systems contained a large collection of text documents. A keyword table, constructed by scanning all the documents, listed the page and line numbers at which each keyword appeared. Someone could retrieve documents on particular subjects by giving all the keywords of interest.

Database systems generalized this principle. Objects are records containing several fields; records are grouped into tables where each field is a column and each record is a row. Records can be selected by stating retrieval criteria, which are values (or ranges) that can occur in selected fields. All the records matching the retrieval criteria will be selected and retrieved. Many businesses keep track of customers, vendors, orders, and inventory with database systems. They can retrieve records matching criteria across all these categories, such as a list of all customers whose incomplete orders require parts from a certain vendor.

The Internet is often regarded as an immense sea of documents for which search engines are the retrieval mechanism. Search engines are services that maintain processes (called webcrawlers) that systematically probe Internet addresses, read web pages, and add references to those pages to a master keyword index. When a user types keywords to the search engine, it can quickly locate the URLs and opening sentences of documents containing those keywords. Even though only about 5% of all web pages are known to any search engine, and even though Internet data is not arranged in an order to facilitate retrievals, most people are surprisingly satisfied with the documents retrieved by their search engines. Even more remarkable are the millions of people who have contributed billions of web pages that can be found and indexed by search engines.

Internet search has proved to be much more challenging than information retrieval. The user (searcher) usually has a question in mind but gives only a few keywords. The search engine must infer aspects of the user's context that allow narrowing the search and ranking the responses. The search engine is a supercomputer with an enormous database that requires (literally) warehouses full of computers to process queries rapidly.