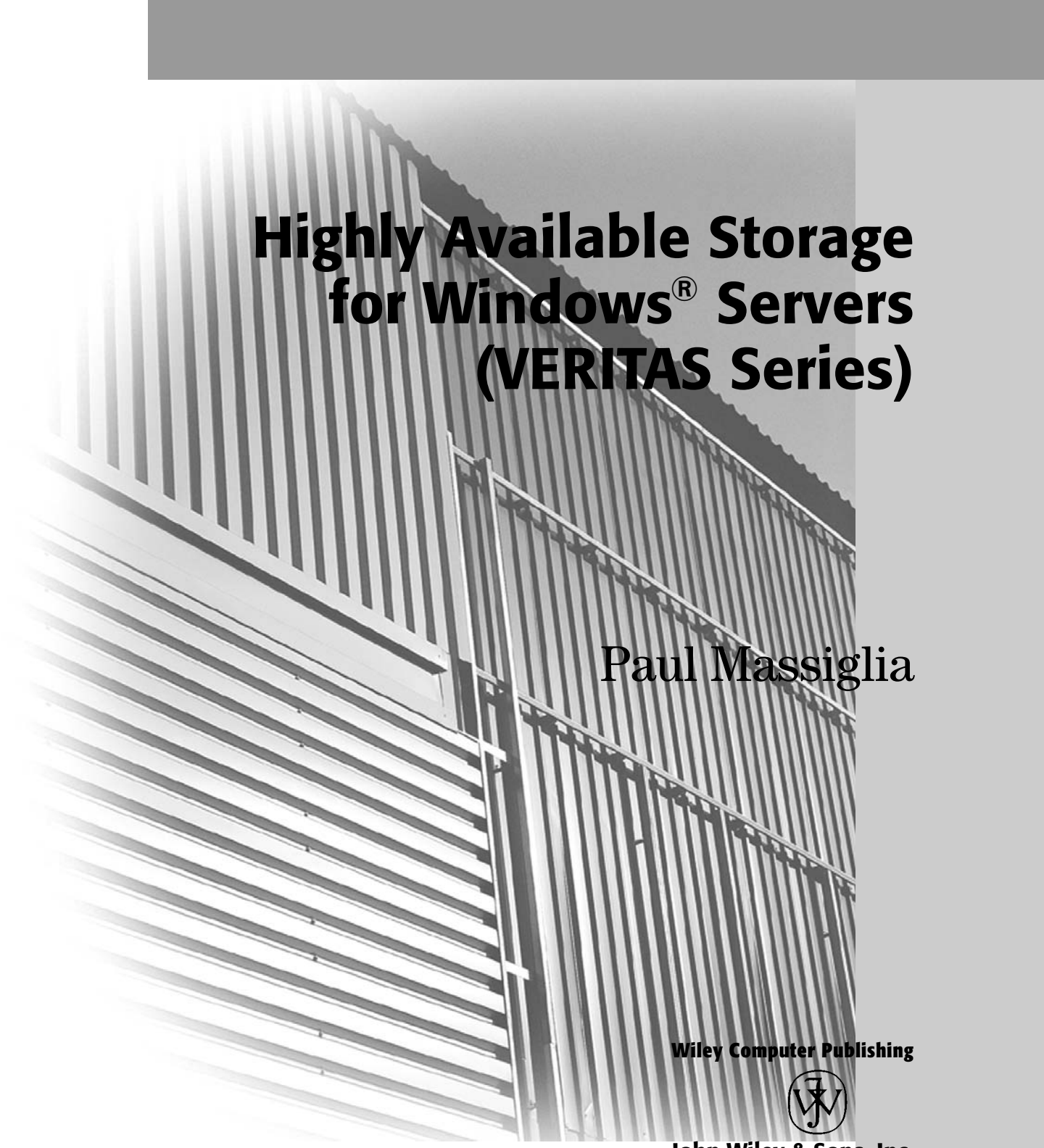


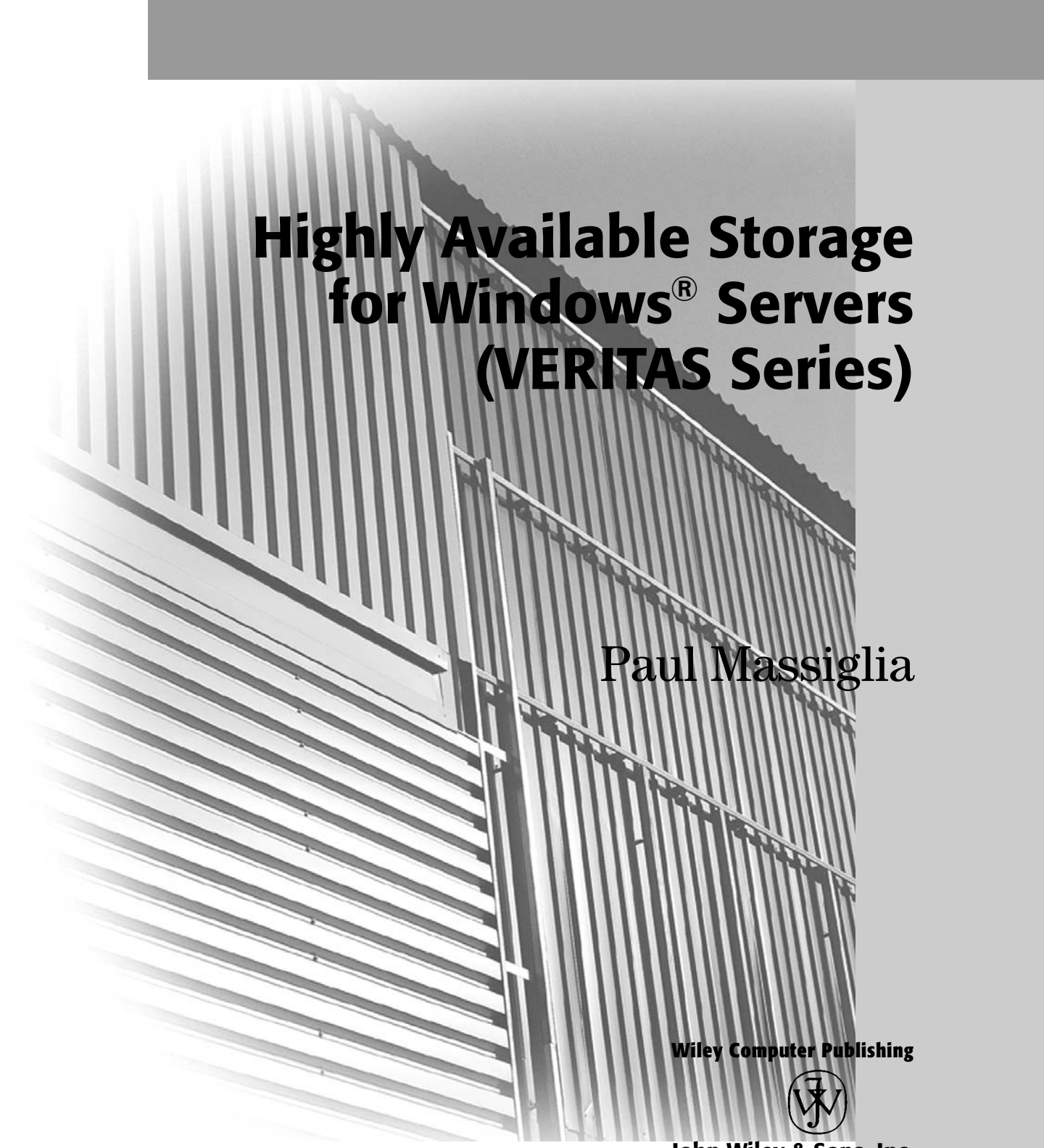
Highly Available Storage for Windows® Servers (VERITAS Series)

Paul Massiglia

Wiley Computer Publishing

John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO



Highly Available Storage for Windows® Servers (VERITAS Series)

Paul Massiglia

Wiley Computer Publishing

John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

Publisher: Robert Ipsen
Editor: Carol A. Long
Assistant Editor: Adaobi Obi
Managing Editor: Micheline Frederick

Text Design & Composition: North Market Street Graphics

This book is printed on acid-free paper.

Published by John Wiley & Sons, Inc.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Massiglia, Paul.
Highly available storage for Windows servers / Paul Massiglia.
p. cm.
ISBN 0-471-03444-4
1. Microsoft Windows server. 2. Client/server computing. 3. Computer storage devices.
I. Title.
QA76.9.C55 M394 2002
004.4'476—dc21 2001006393

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Acknowledgments

The title page bears my name, and it's true, I did put most of the words on paper. But as anyone who has ever written a book—even a modestly technical book like this one—is aware, it is inherently a team effort.

This project wouldn't have come to fruition without a lot of support from a number of talented people. Pete Benoit's Redmond engineering team was of immeasurable technical assistance. I single out Terry Carruthers, Debbie Graham, Pylee Lennil, and Mike Peterson, who all put substantial effort into correcting my mistakes.

Philip Chan's volume manager engineering team reviewed the original manuscript for accuracy, and met all my requests for license keys, access to documents, and software, server accounts and technical consulting.

Particular thanks go to the engineers and lab technicians of VERITAS West, who made both their facilities and their expertise available to me unstintingly. Hrishi Vidwans, Vipin Shankar, Louis MacCubbin, T. J. Somics, Jimmy Lim, Natalia Elenina, and Sathaiah Vanam were particularly helpful in this respect.

Karen Rask, the VERITAS product marketing manager for the Volume Manager described in this book saw value in the concept and drove it through to publication. Thanks, too, to other members of the VERITAS Foundation and Clustering engineering and product management teams who supported the project.

Richard Barker, my manager, had the forbearance not to ask too often what I was doing with all my time. This book actually stemmed from an idea of

Richard's almost two years ago—although in retrospect, he may view it as proof of the adage, "Be careful what you wish for. You may get it."

Many other people contributed, both materially and by encouraging me when necessary. You know who you are.

Errors that remain are solely my responsibility.

Paul Massiglia
Colorado Springs
August, 2001

The Importance of Understanding Online Storage

In recent years, the prevailing user view of failure-tolerant storage has progressed from "seldom-deployed high-cost extra" to "necessity for important data in mission-critical applications," and seems to be headed for "default option for data center storage." During the same period, the storage industry has declared independence from the computer system industry, resulting in a wider range of online storage alternatives for users.

Today, system administrators and managers who buy and configure online storage need to understand the implications of their choices in this complex environment. A prerequisite for making informed decisions about online storage alternatives is an awareness of how disks, volumes, mirroring, RAID, and failure-tolerant disk subsystems work; how they interact and what they can and cannot do.

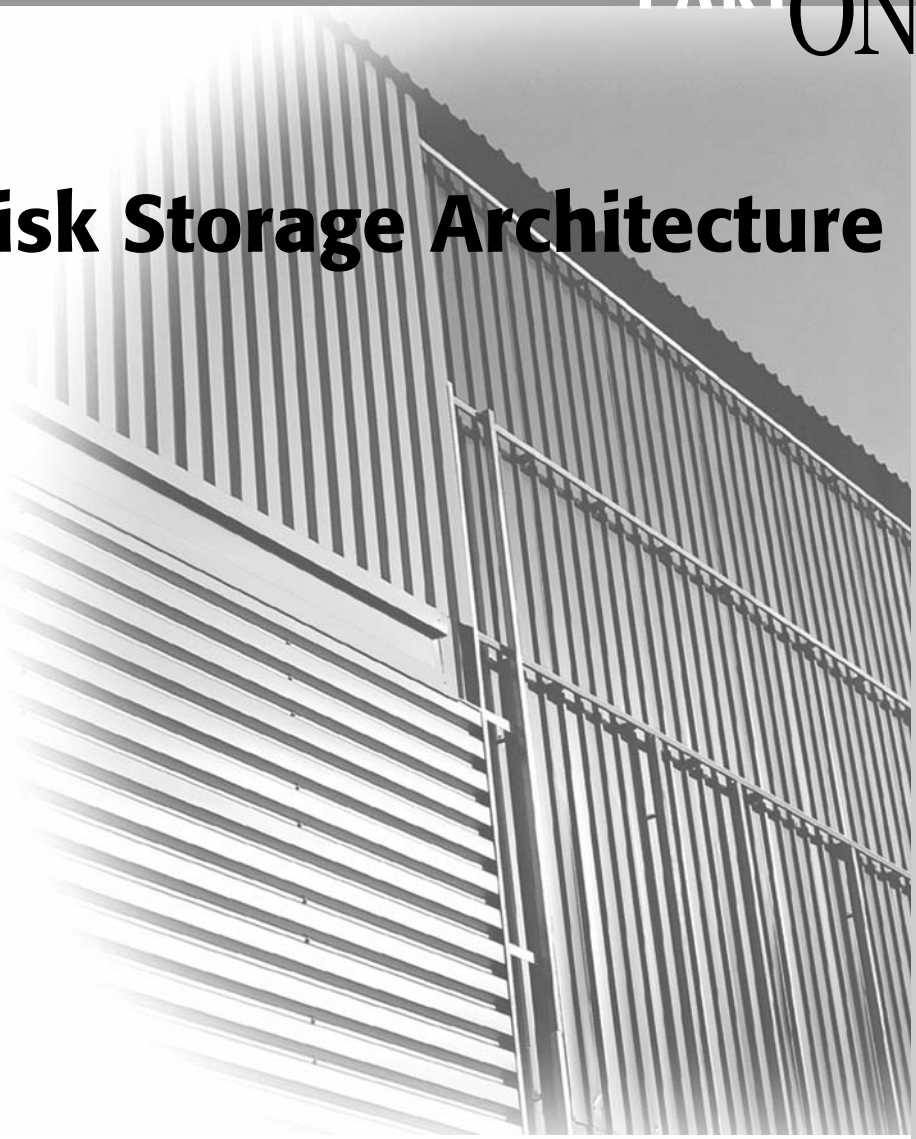
Similarly, client-server application developers and managers must concern themselves with the quality of online storage service provided by their data centers. Understanding storage technology can help these users negotiate with their data centers to obtain the right cost, availability, and performance alternatives for each application.

Moreover, volume management technologies are now available for the desktop. As disk prices continue to decline, widespread desktop use of these techniques is only a matter of time. Desktop users should develop an understanding of storage technology, just as they have done with other aspects of their computers.

Highly Available Storage for Windows Servers (VERITAS Series) was written for all of these audiences. Part I gives an architectural background, to enable users to formulate online storage strategies, particularly with respect to failure tolerance and performance. Part II describes how VERITAS volume management technologies apply these principles in Windows operating system environments.

PART ONE

Disk Storage Architecture



CHAPTER 1

Disk Storage Basics

Data Basics

Computer systems process data. The data they process may be *transient*, that is, acquired or created during the course of processing and ceasing to exist after processing is complete or it may be *persistent*, stored in some permanent fashion so that program after program may access it.

Transient Data

The solitaire game familiar to Windows users is an example of transient data. When a solitaire player starts a new game, transient data structures representing a deck of cards dealt into solitaire stacks is created. As the user plays the game, keystrokes and mouse clicks are transformed into actions on virtual cards. The solitaire program maintains transient data structures that describe which cards are exposed in which stacks, which remain hidden and which have been retired. As long as the player is engaged in the game, the solitaire program maintains the data structures. When a game is over, however, or when the program ceases to run, the transient data structures are deleted from memory and cease to exist.

In today's computers, with volatile random access memory, programs may cease to run and their transient data cease to exist for a variety of uncontrol-

lable reasons that are collectively known as *crashes*. Crashes may result from power failure, from operating system failure, from application failure, or from operational error. Whatever the cause, the effect of a crash is that transient data is lost, along with the work or business state it represents. The consequence of crashes is generally a need to redo the work that went into creating the lost transient data.

Persistent Data

If all data were transient, computers would not be very useful. Fortunately, technology has provided the means for data to last, or *persist*, across crashes and other program terminations. Several technologies, including battery-backed dynamic random access memory (*solid state disk*) and optical disk, are available for storing data persistently; but far and away the most prevalent storage technology is the magnetic disk.

Persistent data objects (for example, files) outlast the execution of the programs that process them. When a program stops executing, its persistent data objects remain in existence, available to other programs to process for other purposes. Persistent data objects also survive crashes. Data objects that have been stored persistently prior to a crash again become available for processing after the cause of the crash has been discovered and remedied, and the system has been restarted. Work already done to create data objects or alter them to reflect new business states need not be redone. Persistent data objects, therefore, not only make computers useful as recordkeepers, they makes computers more resilient in the face of the inevitable failures that befall electromechanical devices.

Persistent data objects differ so fundamentally from transient data that a different metaphor is used to describe them for human use. Whereas transient data is typically thought of in terms of *variables* or *data structures* to which values are assigned (for example, **`let A = 23`**), persistent data objects are typically regarded as *files*, from which data can be read and to which data can be written. The file metaphor is based on an analogy to physical file cabinets, with their hierarchy of drawers and folders for organizing large numbers of data objects. Figure 1.1 illustrates key aspects of the file metaphor for persistent data objects.

The file metaphor for persistent computer data is particularly apt for several reasons:

- The root of the metaphor is a physical device—the file cabinet. Each file cabinet represents a separate starting point in a search for documents. An organization that needs to store files must choose between smaller number of larger file cabinets and a larger number of smaller filer cabinets.

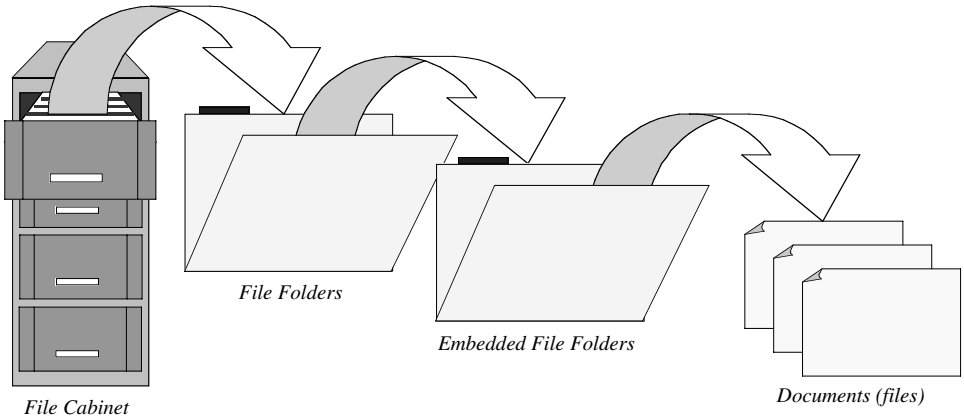


Figure 1.1 The file metaphor for persistent data objects.

- *File cabinets fundamentally hold file folders.* File folders may be hierarchical: They may hold folders, which hold other folders, and so forth.
- *Ultimately, the reason for file folders is to hold documents, or files.* Thus, with rare exceptions, the lowest level of hierarchy of file folders consists of folders that hold documents.
- *File folders are purely an organizational abstraction.* Any relationship between a folder and the documents in it is entirely at the discretion of the user or system administrator who places documents in folders.

The file cabinet/file folder metaphor has proven so useful in computing that it has become nearly universal. Virtually all computers, except those that are embedded in other products, include a software component called a *file system* that implements the file cabinet metaphor for persistent data. UNIX systems typically use a single cabinet abstraction, with all folders contained in a single *root* folder. Windows operating systems use a multicabinet abstraction, with each "cabinet" corresponding to a physical or logical storage device that is identified by a drive letter.

Disk Basics

Of the technologies used for persistent data storage, the most prevalent by far is the *rotating magnetic disk.* Magnetic disks have several properties that make them the preferred technology solution for storing persistent data:

Low cost. Today, raw magnetic disk storage costs between 1 and 5 cents per megabyte. This compares with a dollar or more for dynamic random access memory.

Random access. Relatively small blocks of data stored on magnetic disks can be accessed in random order.[1] This allows programs to execute and process files in an order determined by business needs rather than by data access technology.

High reliability. Magnetic disks are among the most reliable electromechanical devices built today. Disk vendors routinely claim that their products have statistical mean times between failures of as much as a million hours.

Universality. Over the course of the last 15 years, disk interface technology has gradually become standardized. Today, most vendors' disks can be used with most computer systems. This has resulted in a competitive market that tends to reinforce a cycle of improving products and decreasing prices.

Disks, Data and Standards

Standardization of disk interface technology unfortunately has not led to to standardization of data formats. Each operating systems and file system has a unique "on disk format," and is generally not able to operate on disks written by other operating systems and file systems without a filter or adapter application. Operating systems that use Windows NT technology include three major file systems (Figure 1.2 shows the latter two):

File Allocation Table, or FAT. DOS-compatible files system retained primarily for compatibility retained for compatibility with other operating systems, both from Microsoft and from other vendors.

FAT32. 32-bit version of the FAT file system originally developed to allow personal computers to accommodate large disks, but supported by Windows operating systems that use NT technology.

NTFS. The native files system for NT Technology files systems.

NT Technology operating systems also include an Installable File System (IFS) facility that enables software vendors to install additional software layers in the Windows operating system data access stack to filter and preprocess file system input/output (I/O) requests.

The three Windows NT file systems use different on-disk formats. The ***format*** function of Windows NT Disk Administrator prepares a disk for use with

[1]Strictly speaking, disks do not provide random access to data in quite the same sense as dynamic random access memory (DRAM). The primitive actions (and therefore the time) required to access a block of data depend partly upon the last block accessed (which determines the seek time) and partly upon the timing of the access request (which determines the rotational latency). Unlike tapes, however, each disk access specifies explicitly which data is to be accessed. In this sense, disks *are* like random access memory, and operating system I/O driver models treat disks as random access devices.

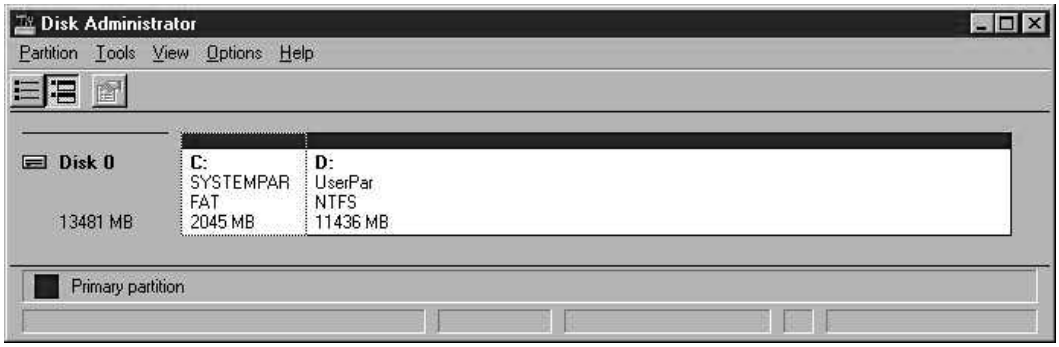


Figure 1.2 FAT and NTFS file systems on different disk partitions.

one of the file systems by writing the initial file system *metadata*[2] on it. The operating system *mount* function associates a disk with the file system for which it is formatted and makes data on the disk accessible to applications.

Windows operating systems mount all visible disks automatically when they start up, so the act of associating a disk with its file system is generally transparent to system administrators and users once the disk has been formatted for use.

Magnetic Disk Operation

While magnetic disks incorporate a diverse set of highly developed technologies, the physical principles on which they are based are simple. In certain materials, called *ferromagnetic* materials, small regions can be permanently magnetized by placing them near a magnetic field. Other materials are *paramagnetic*, meaning that they can be magnetized momentarily by being brought into proximity with an electrical current in a coil. Figure 1.3 illustrates the components of a magnetic recording system.

Once a ferromagnetic material has been magnetized (e.g., by being brought near a strongly magnetized paramagnetic material), moving it past a paramagnetic material with a coil of wire wrapped around it results in a voltage change corresponding to each change in magnetization direction. The timing of these pulses, which is determined by the distance between transitions in field direction and the relative velocity of the materials, can be interpreted as a stream of data bits, as Figure 1.4 illustrates.

[2]File system metadata is data about the file system and user data stored. It includes file names, information about the location of data within the file system, user access right information, file system free space and other data.

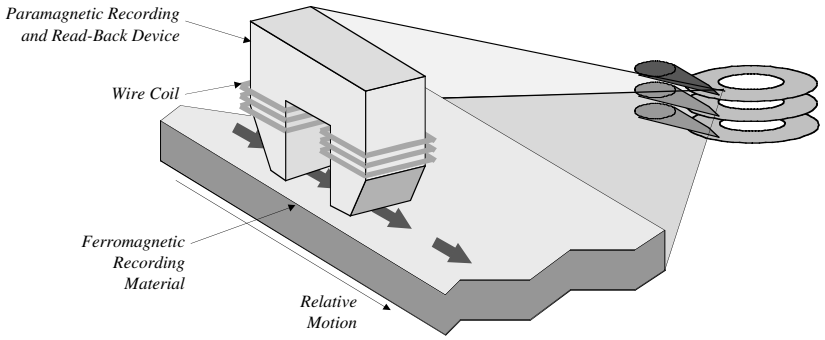


Figure 1.3 General principle of magnetic data recording.

Magnetic disk (and tape) recording relies on relative motion between the ferromagnetic recording material (the *media*) and the device providing recording energy or sensing magnetic state (the *head*). In magnetic disks, circular platters rotate relative to a stationary *read/write head* while data is read or written.

Pulse Timing and Recording Codes

Disk platters rotate at a nominally constant velocity, so the relative velocity of read/write head and media is nominally constant, allowing constant time slots to be established. In each time slot, there either is or is not a pulse. With constant rotational velocity and an electronic timer generating time slots, pulses could be interpreted as binary ones, and the absence of pulses could be interpreted as zeros. Figure 1.5 illustrates this simple encoding. Pulses peak in the

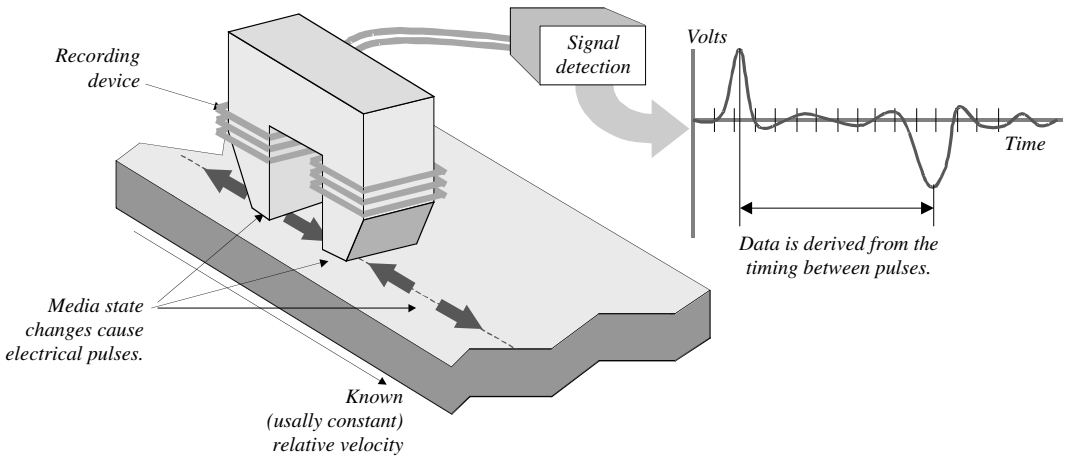


Figure 1.4 Recovering data recorded on disks from pulse timing.

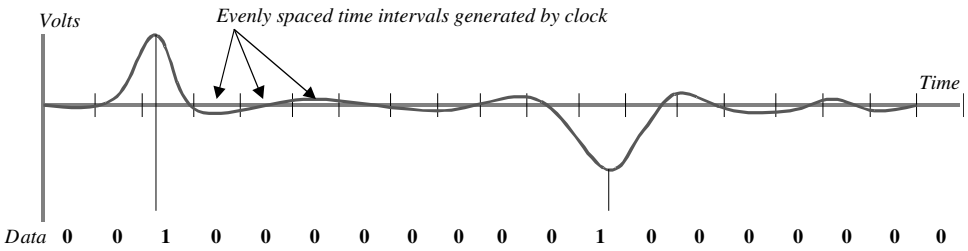


Figure 1.5 Inferring data from a combination of voltage pulses and timing.

third and twelfth time intervals and are interpreted as "1" bits. Other intervals lack pulse peaks and are interpreted as "0" bits.

If rotational velocity were truly constant and the electronics used to establish time slots were perfect, this simple encoding scheme would be adequate. Unfortunately, minor variations in rotational speed and timer electronics can cause pulses to drift into adjacent time slots and to be interpreted incorrectly, as illustrated in Figure 1.6.

Using the encoding scheme illustrated in Figures 1.5 and 1.6, an entire block of binary zeros would produce no pulses when read back. To guard against this, on-disk data is encoded using algorithms that guarantee the occurrence of frequent pulses independent of the input data pattern of ones and zeros. Pulses are input to a phase-locked loop, which in turn adjusts time slots. The constant fine-tuning maximizes the likelihood that pulses will be interpreted correctly. Encoding schemes that guarantee frequent pulses (or 1 bits) independent of the application data pattern are called *run-length-limited*, or RLL codes. Figure 1.7 illustrates a very simple RLL code.

RLL codes are characterized by the smallest and largest possible intervals between 1- bits in the encoded bit stream. Thus, the code illustrated in Figure 1.7 would be characterized as a 0,4 code, because:

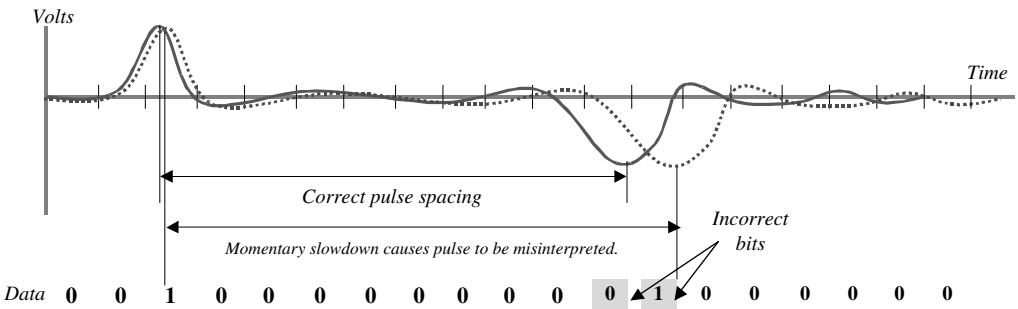


Figure 1.6 Effect of timing on data recovery.

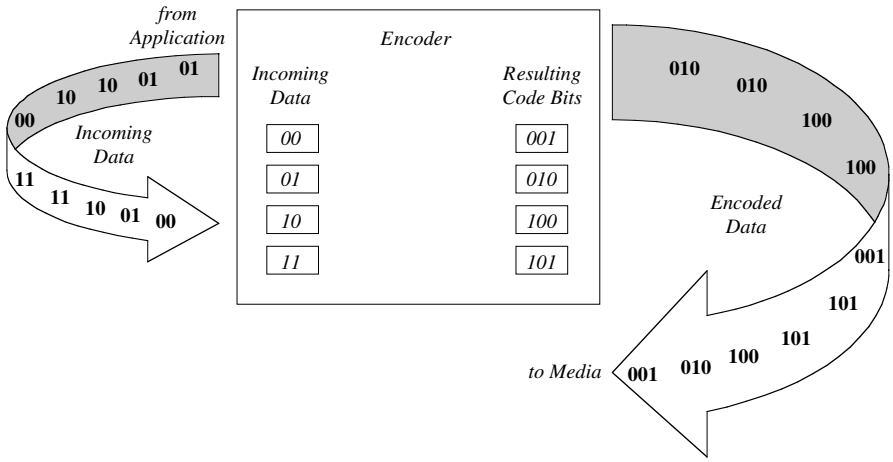


Figure 1.7 Example of a run-length-limited data encoding.

- Pulses in adjacent time slots can occur (e.g., user data 0010, which encodes into 001100).
- There can be at most four time slots between pulses (user data 1000, which encodes into 100001).

With this code, a pulse is guaranteed to occur in the encoded bit stream at least every fifth time slot. Thus, the maximum time that the timing generator must remain synchronized without feedback from the data stream itself is four time slots.

Actual RLL codes are typically more elaborate than the one illustrated in Figure 1.7, sometimes guaranteeing a minimum of one or more time intervals between adjacent pulses. This is beneficial because it decreases the frequency spectrum over which the disk's data decoding logic must operate.

Error Correction Codes

Even with run-length encoding and phase-locked loops, errors can occur when data is read from disks. Mathematically elaborate checksum schemes have been developed to protect against the possibility of incorrect data being accepted as correct; and in many instances, these checksums can correct errors in data delivered by a disk. In general, these error correction codes, or ECCs, are generated by viewing a block of data as a polynomial, whose coefficients are consecutive strings of bits comprising the data block. As data is written, specialized hardware at the source (e.g., in a disk, just upstream of the write logic) divides the data polynomial by a smaller, fixed polynomial called a *generating polynomial.* The quotient of the division is discarded, and the

remainder, which is guaranteed to be of limited size, is appended to the data stream as a checksum and written to disk media, as illustrated in Figure 1.8.

When data is read back from the disk, the read logic performs the same computation, this time retaining the quotient and comparing the computed remainder to the checksum read from the disk. A difference in the two is a signal that data and/or checksum have been read incorrectly. The mathematical properties of the checksum are such that the difference between the two remainders plus the coefficients of the quotient can be used, to correct the erroneous data within limits.

The specialized hardware used to compute checksums is usually able to correct simple data errors without delaying the data stream. More complex error patterns require disk firmware assistance. So when these patterns occur, data may reach memory out of order. Error-free blocks that occur later in the data stream may be delivered before earlier erroneous ones. Thus it is important for applications and data managers not to assume that data read from a disk is present in memory until the disk has signaled that a read is complete.

Locating Blocks of Data on Magnetic Disks

For purposes of identifying and locating data, magnetic disks are logically organized into concentric circles called *tracks*, as illustrated in Figure 1.8. Read/write heads are attached to *actuators* that move them from track to track.

On each track, data is stored in blocks of fixed size (512 bytes in Windows and most other systems). Each disk block starts with a synchronization pattern

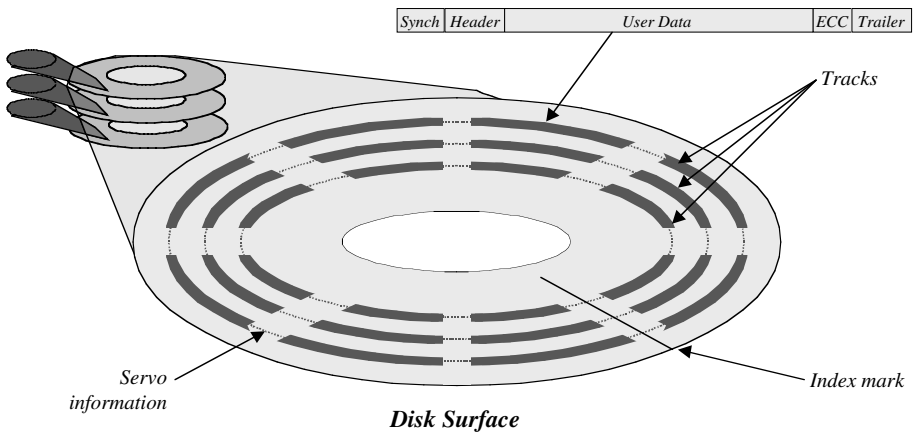


Figure 1.8 Magnetic disk data layout.

and identifying header[3] followed by user data, an error correction code (ECC) and a trailer pattern. Adjacent blocks are separated by *servo signals*, recorded patterns that help keep the read/write head centered over the track. An index mark at the start of each track helps the disk's position control logic keep track of rotational position.

Figure 1.8 illustrates one surface of one disk platter. All the blocks at a given radius comprise a track. On a disk with multiple recording surfaces, all of the tracks at a given radius are collectively known as a *cylinder*. The disk illustrated in Figure 1.8 has the same number of blocks on each track. The capacity of such a disk is given by:

$$\begin{aligned}\text{Disk capacity (bytes)} = {} & \text{number of blocks per track} \\ & \times \text{number of tracks} \\ & \times \text{number of data heads (data surfaces)} \\ & \times \text{number of bytes per block}\end{aligned}$$

Each block of data on such a disk can be located ("addressed") by specifying a cylinder, a head (recording surface) and a (relative) block number. This is called cylinder, head, sector, or C-H-S addressing. Figure 1.9 illustrates C-H-S addressing.

[3]In some newer disk models, the header is eliminated to save space and increase storage capacity.

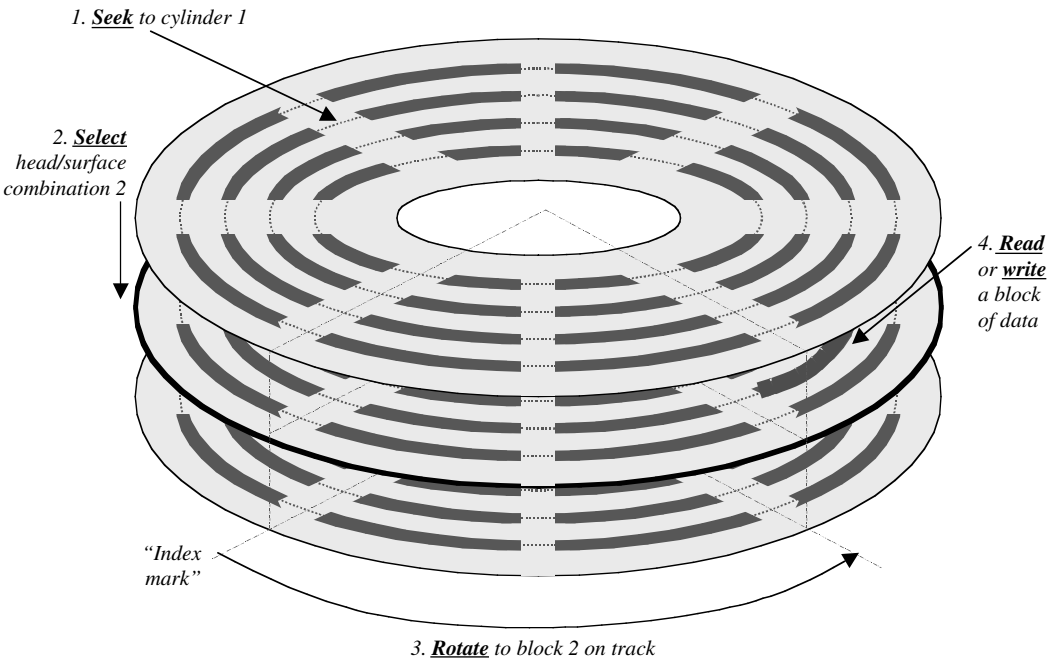


Figure 1.9 Locating data on a disk.

Figure 1.9 illustrates the three distinct operations required to locate a block of data on a multisurface disk for reading or writing:

- *Seeking* moves the actuator to position the recording heads approximately over the track on which the target data is located.
- *Selection* of the head that will read or write data connects the head's output to the disk's read/write channel so that servo information can be used to center the head precisely on the track.
- *Rotation* of the platter stack brings the block to be read or written directly under the head, at which time the read or write channel is enabled for data transfer.

Logical Block Addressing

C-H-S addressing is inconvenient for disk drivers and file systems because it requires awareness of disk *geometry.* To use C-H-S addressing to locate data, a program must be aware of the number of cylinders, recording surfaces, and blocks per track of each disk. This would require that software be customized for each type of disk. While this was in fact done in the early days of disk storage, more recently the disk industry has adopted the more abstract *logical block addressing* model for disks, illustrated in Figure 1.10.

With logical block addressing, disk blocks are numbered in ascending sequence. To read or write data, file systems and drivers specify a logical block number. A microprocessor in the disk itself converts between the logical block address and the C-H-S address, as illustrated in Figure 1.11.

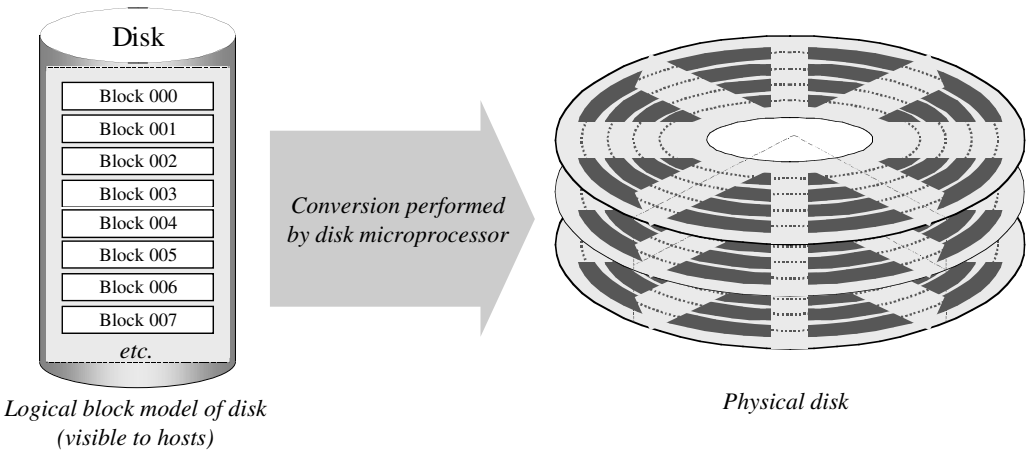


Figure 1.10 The logical block disk data-addressing model.

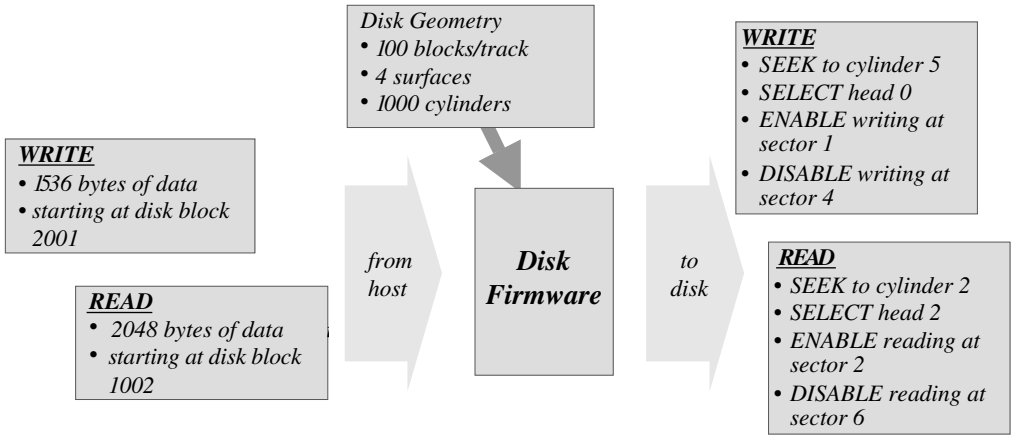


Figure 1.11 Conversion between logical block and C-H-S data addressing.

Desktop computer disks typically support both C-H-S and block data addressing, largely for reasons of backward compatibility. The SCSI and Fibre Channel disks typically used in servers and multidisk subsystems use block addressing exclusively.

Zoned Data Recording

Each block on the outermost track of the disk platter illustrated in Figure 1.9 occupies considerably more linear distance than the corresponding block on the innermost track, even though it contains the same amount of data. During the early 1990s, in an effort to reduce storage cost, disk designers began to design disks in which the longer outer tracks are divided into more blocks than the shorter inner ones. Although this increased the complexity of disk electronics, it also increased the storage capacity for any given level of head and media technology by as much as 50 percent. Today, this technique goes by the names such as *zoned data recording* (ZDR). The cylinders of a ZDR disk are grouped into *zones*, each of which is formatted to hold a different number of 512 byte blocks. Figure 1.12 illustrates a platter surface of a ZDR disk with two zones.

In Figure 1.12, each track in the inner zone contains 8 blocks, while each track in the outer zone contains 16. Compared to the disk illustrated in Figure 1.9, capacity is 50 percent higher, with little if any incremental product cost. (Figure 1.12 uses unrealistically low numbers of blocks per track for the sake of clarity of the diagram. Typical ZDR disks have 20 or more zones, with between 100 and 200 blocks per track. For 3.5-inch diameter disks, the outermost zone usually has about twice as many blocks per track as the innermost zone.)

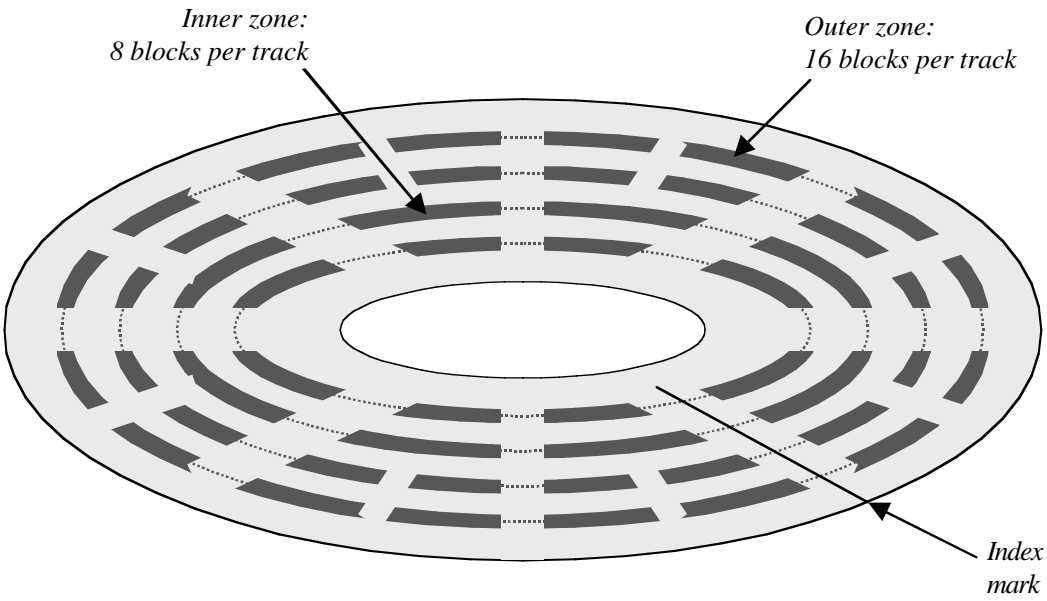


Figure 1.12 One platter of a zoned data-recorded disk.

Because of the beneficial effect on cost per byte of storage, zone bit recording has essentially become ubiquitous.

Disk Media Defects

With the high recording densities in use today, miniscule material defects can render part of a recording surface permanently unrecordable. The blocks that would logically lie in these surface areas are called *defective blocks*. Attempts to read or write data in a defective block always fail. The conventional way of dealing with defective blocks is to reserve a small percentage of a disk's block capacity to be substituted for defective blocks when they are identified. Correspondence tables relate the addresses of defective blocks to addresses of substitute blocks and enable file systems to treat disks as if they were defect-free. Figure 1.13 illustrates such a correspondence table. These tables are sometimes called *revectoring tables* and the process of converting a host-specified block number that maps to a defective block into the C-H-S address of a substitute block is called *revectoring*.

In the early days of disk technology, defective blocks were visible to hosts, and operating system drivers maintained revectoring tables for each disk. Like address conversion, however, revectoring is highly disk type-specific.

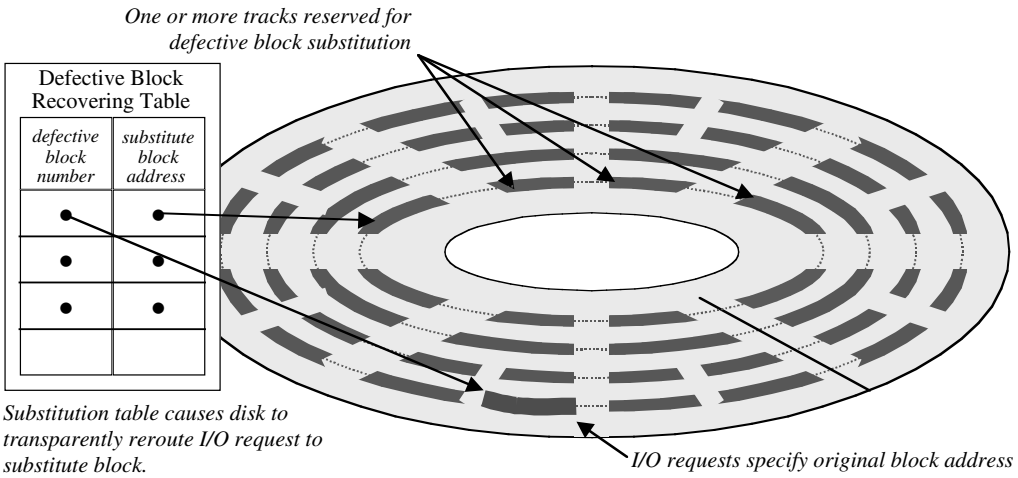


Figure 1.13 Defective block substitution.

Consequently, it became apparent to the disk industry that revectoring could be performed most effectively by the disks themselves. Today, most disks revector I/O requests addressed to defective blocks to reserved areas of the media using a correspondence table similar to that illustrated in Figure 1.13.

Inside-the-disk *bad block revectoring* (BBR) allows host drivers as well as file systems to treat disks as if there were no defective blocks. From the host's point of view, a disk is a consecutively numbered set of blocks. Within the disk, block addresses specified by file systems are regarded as *logical.* The disk translates them into physical media locations, and in so doing, transparently substitutes for defective blocks as necessary.

Writing Data on Magnetic Disks

A little-recognized fact about magnetic disks is that, there is very little physical feedback to confirm that data has been correctly written. Disk read/write logic and the read/write channel write each block's preamble, header, user data, ECC, and trailer, and disengage. Hardware in the read/write channel verifies signal levels, and head position is validated frequently with servo feedback, but data written to the media is not verified by, for example, immediate rereading, as is done with tape drives. Disks are able to reread and verify data after writing it, but this necessitates an extra disk revolution for every write. This affects performance adversely (single-block write times can increase by 50 percent or more), so the capability is rarely used in practice. Fortunately, writing data on disks is an extremely reliable operation, enough so that most of data processing can be predicated upon it. There is always a miniscule chance, however, that data written by a host will not be readable. Moreover,

unreadable data will not be discovered until a read is attempted, by which time it is usually impossible to re-create. The remote possibility of unreadable data is one of several reasons to use failure-tolerant volumes for business-critical online data.

Intelligent Disks

Electronic miniaturization and integration have made it technically and economically feasible to embed an entire disk controller in every disk built today, making the disk, in effect, a complete subsystem, as the block diagram in Figure 1.14 illustrates.

A subtle but important property of the architecture illustrated in Figure 1.14 is the *abstraction* of the disk's external interface. Today, host computers no longer communicate directly with disk read/write channels. Instead, they communicate with the logical interface labeled "Host I/O Bus Interface" in Figure 1.14. I/O requests sent to this logical interface are transformed by a microprocessor within the disk. Among its activities, this processor:

- *Converts* logical block addresses into C-H-S addresses and performs revectoring as necessary.
- *Breaks down* hosts' read and write requests into more primitive seek, search, and read and write channel enable and disable operations.
- *Manages data transfer* through the disk's internal buffers to and from the host.

All of this activity is transparent to hosts, which use simple read and write commands that specify logical block addresses from a dense linear space.

Abstract host I/O interfaces allow disks to evolve as component technologies develop *without significant implications for their external environment.* A

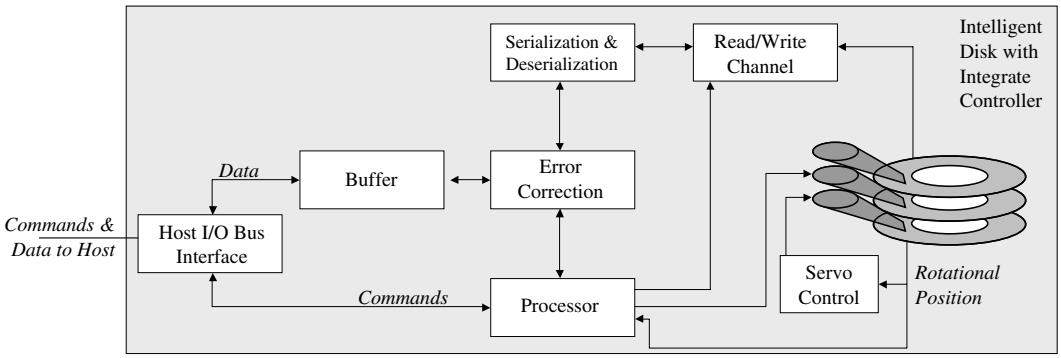


Figure 1.14 Block diagram of an intelligent disk with embedded controller.

disk might use radically different technology from its predecessors, but if it responds to I/O requests, transfers data, and reports errors in the same way, the system support implications of the new disk are very minor, making market introduction easy. This very powerful abstract I/O interface concept is embodied in today's standard I/O interfaces such as SCSI, ATA (EIDE)[4] and FCP. Disks that use these interfaces are easily interchangeable. This allows applications to use the increased storage capacity and performance delivered as technology evolves, with minimal support implications.

Other Applications of Disk Intelligence: SMART Technology

One innovative use of disk intelligence that has emerged in recent years is disk self-monitoring for predictive failure analysis. With so many millions of samples upon which to base statistical analyses, disk manufacturers have developed significant bodies of knowledge about how certain physical conditions within disk drives indicate impending failures before they occur. In general, these physical conditions are sensed by a disk and are implementation-specific. Such factors as head flying height (distance between read/write head and disk platter), positioning error rates, and media defect rates are useful indicators of possible disk failure.

A SCSI standard called Self-Monitoring, Analysis, and Reporting Technology (SMART) provides a uniform mechanism that enables disks to report normalized predictive failure information to a host environment. Disks that use the ATA interface report raw SMART information when polled by their hosts. The hosts then make any predictive failure decisions. Large system disks use built-in intelligence to analyze SMART information themselves and only report danger signals to their hosts when analysis indicates that a failure might be imminent.

Hosts that support SMART alert system administrators when a disk is in danger of failing. The system administrator can then take action to protect data on the failing disk—for example, by scheduling an immediate backup.

Though SMART improves the reliability of data stored on disks, the technology is not without its limitations. It is chiefly useful to predict failures that are characterized by gradual deterioration of some measurable disk parameter. SMART does not protect against sudden failures, as are typical of logic module failures.

Because SMART reports refer to conditions that are deteriorating with time, system administrators must receive and act on them promptly. SMART is thus most useful in environments that are monitored constantly by administrators.

[4]AT(advanced technology) Attachment; also known as extended IDE intelligent drive electronics, FCP = Fibre Channel Protocol

SMART is indeed a useful technology for improving the reliability of data stored on disks, but it is most effective in protecting data against disk failures when used in conjunction with volume management techniques described later in Chapter 2.

Disk Controller and Subsystem Basics

As disks evolved during the 1980s into the self-contained intelligent subsystems represented in Figure 1.14, separate controllers were no longer required for low-level functions such as motion control, data separation, and error recovery. The concept of aggregating disks to improve performance and availability is a powerful one, however, so *intelligent disk subsystems* with *aggregating disk controllers* evolved in their place. Figure 1.15 illustrates the essentials of an intelligent disk subsystem with an aggregating controller.

The aggregating disk controller that is shown in Figure 1.15 with four disk I/O bus interfaces that connect to a memory access bus internal to the controller. The disk I/O buses connect intelligent disks to the controller. The disk controller coordinates I/O to *arrays* of two or more disks, and makes them appear to host computers over the host I/O bus interface as *virtual disks*.

Aggregating disk controllers can:

- *Concatenate* disks and present their combined capacity as a single large *virtual disk*.
- *Stripe* or distribute data across disks for improved performance and present the combined capacity as a single large virtual disk.

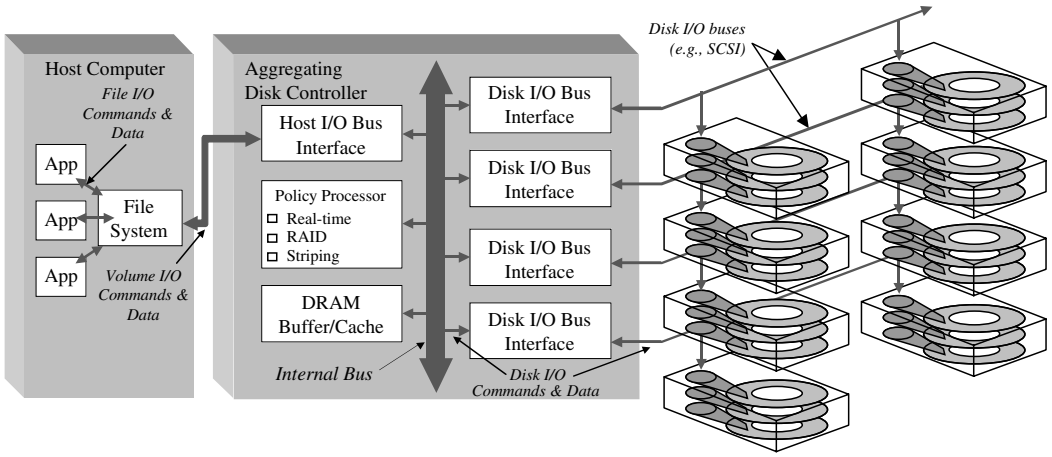


Figure 1.15 Intelligent disk subsystem with aggregating disk controller.

- *Mirror* identical block contents on two or more disks or striped volumes, and present them as a single failure-tolerant virtual disk.
- *Combine* several disks using Redundant Array of Independent Disk (RAID) techniques to stripe data across the disks with parity check data interspersed, and present the combined available capacity of the disks as a single failure-tolerant virtual disk.

Each of the disk I/O bus interfaces in Figure 1.15 sends I/O requests to, and moves data between, one or more disks and a dynamic random access memory (DRAM) buffer within the aggregating controller. Similarly, a host I/O bus interface in the aggregating controller moves data between the buffer and one or more host computers. A *policy processor* transforms each host I/O request made to a volume into one or more requests to disks, and sends them to disks via the disk I/O bus interfaces. For example, if two mirrored disks are being presented to host computers as a single failure-tolerant virtual disk, the aggregating controller would:

- *Choose* one of the disks to satisfy each application read request, and issue a read request to it.
- *Convert* each host write request made to the volume into equivalent write requests for each of the mirrored disks.

Similarly, if data were striped across several disks, the aggregating controller's policy processor would:

- *Interpret* each host I/O request addressed to the striped volume to determine which data should be written to or read from which disk(s).
- *Issue* the appropriate disk read or write requests.
- *Schedule* data movement between host and disk I/O bus interfaces.

For RAID arrays, in which data is also typically striped, the aggregating controller's policy processor would perform these functions and would update parity each time user data was updated.

External and Embedded Array Controllers

From a host computer standpoint, a RAID controller is either *external* or *embedded* (mounted) within the host computer's housing.

External RAID controllers function as many-to-many bridges between disks and external I/O buses, such as parallel SCSI or Fibre Channel, to which physical disks can also be attached. External RAID controllers organize the disks connected to them into *arrays* and make their storage capacity available to

host computers by emulating disks on the host I/O buses. Figure 1.16 illustrates a system configuration that includes an external RAID controller.

External RAID controllers are attractive because they emulate disks and, therefore, require little specialized driver work. They are housed in separate packages whose power, cooling, and error-handling capabilities are optimized for disks. They typically accommodate more storage capacity per bus address or host port than the embedded controllers discussed next. Moreover, since they are optimized for larger systems, they tend to include advanced performance-enhancing features such as massive cache, multiple host ports, and specialized hardware engines for performing RAID computations. The main drawbacks of external RAID controllers are their limited downward scaling and relatively high cost.

Embedded, or *internal*, *RAID controllers* normally mount within their host computer enclosures and attach to their hosts using internal I/O buses, such as PCI peripheral component interconnect. Like external RAID controllers, embedded controllers organize disks into arrays and present virtual disks to the host environment. Since there is no accepted standard for a direct disk-to-PCI bus interface, embedded controllers require specialized drivers that are necessarily vendor-unique. Figure 1.17 illustrates a system configuration that includes an embedded RAID controller.

Embedded RAID controllers are particularly attractive for smaller servers because of their low cost and minimal packaging requirements. An embedded RAID controller is typically a single extended PCI module. Some vendors design server enclosures that are prewired for connecting a limited number of disks mounted in the server enclosure itself to an embedded RAID controller. The disadvantages of embedded RAID controllers are their limited scaling and failure tolerance and their requirement for specialty driver software.

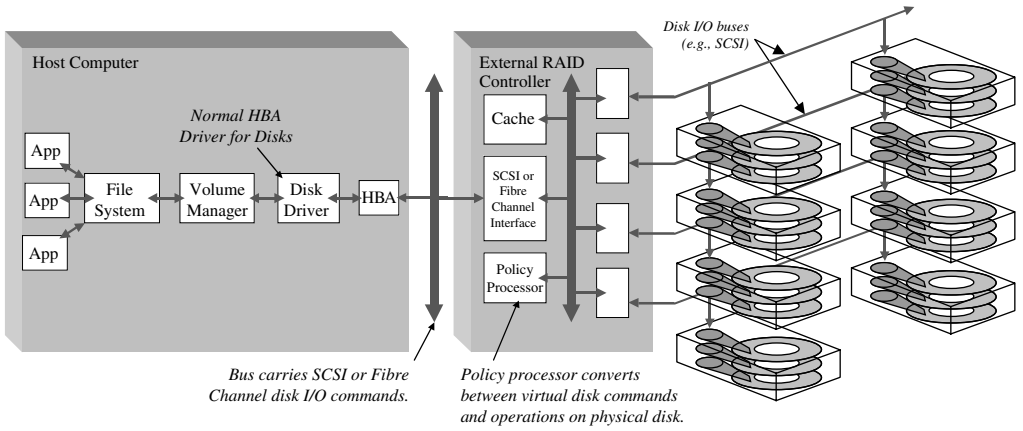


Figure 1.16 External RAID controller.

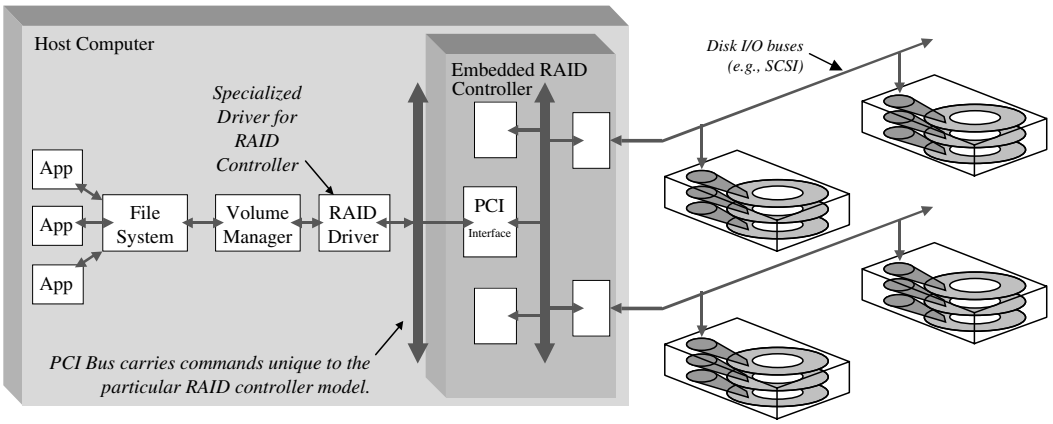


Figure 1.17 Embedded RAID controller.

The virtual disks presented by external RAID controllers are functionally identical to physical disks and are usually controlled by native operating system disk drivers with little or no modification. Embedded RAID controllers, on the other hand, typically require unique bus interface protocols and, therefore, specialized drivers, typically supplied by the controller vendor. Both external and embedded RAID controllers' virtual disks can be managed by host-based volume managers as though they were physical disks.

Both external and embedded RAID controllers require management interfaces to create and manage the virtual disks they present. Embedded RAID controllers typically have *in-band* management interfaces, meaning that management commands are communicated to the controller over the same PCI interface used for I/O. External controllers typically offer both in-band management interfaces using SCSI or FCP commands and *out-of-band* interfaces using Ethernet or even serial ports. Out-of-band interfaces enable remote management from network management stations and preconfiguration of disk array subsystems before they are installed.

Host-Based Aggregation

The architecture of the aggregating disk controller block diagrammed in Figure 1.15 is very similar to that of a general-purpose computer. In fact, most disk controllers use conventional microprocessors as policy processors, and several use other conventional computer components as well, such as PCI bus controller *application-specific integrated circuits* (ASICs), such as the single-chip PCI interfaces found on most computer mainboards. As processors became more powerful during the 1990s, processing became an abundant resource, and several software developers implemented the equivalent of

aggregating disk controllers' function in a host computer system software component that has come to be known as a *volume manager*. Figure 1.18 depicts a system I/O architecture that uses a host-based volume manager to aggregate disks.

This figure represents a PCI-based server, such as might run the Windows NT or Windows 2000 operating system. In such servers, disk I/O interfaces are commonly known as *host bus adapters*, or HBAs, because they adapt the protocol, data format, and timing of the PCI bus to those of an external disk I/O bus, such as SCSI or Fibre Channel. Host bus adapters are typically designed as add-in circuit modules that plug into PCI slots on a server mainboard or, in larger servers, a PCI to memory bus adapter. Small server main boards often include integrated host bus adapters that are functionally identical to the add-in modules.

Host bus adapters are controlled by operating system software components called *drivers*. Windows operating systems include drivers for the more popular host bus adapters, such as those from Adaptec, Q Logic, LSI Logic, and others. In other cases, the vendor of the server or host bus adapter supplies a Windows-compatible HBA driver. Microsoft's Web site contains a hardware compatibility list that contains information about host bus adapters that have been successfully tested with each of the Windows operating systems.

HBA drivers are *pass-through* software elements, in the sense that they have no awareness of the meaning of I/O requests made by file systems or other applications. An HBA driver passes each request made to it to the HBA for transmission to and execution by the target disk. For data movement efficiency, HBA drivers manage mapping registers that enable data to move

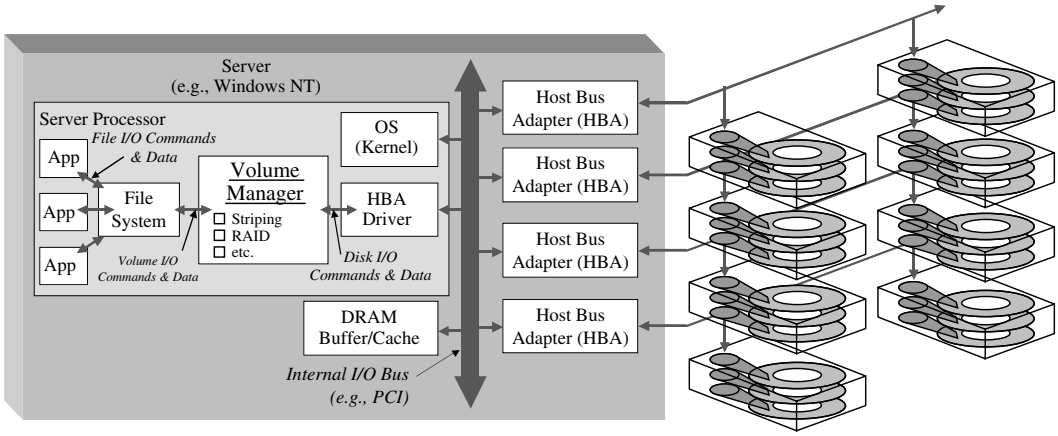


Figure 1.18 Host-based disk subsystem with aggregating software.

directly between the HBA and main memory. HBA drivers do not filter I/O requests, nor do they aggregate disks into volumes (although there are some PCI-based RAID controllers that perform disk aggregation).

In systems like the one depicted in Figure 1.18, the *volume manager* aggregates disks into *logical volumes* that are functionally equivalent to the virtual disks instantiated by aggregating disk controllers. The volume manager is a software layer interposed between the file system and HBA drivers. From the file system's point of view, a volume manager behaves like a disk driver. The volume manager responds to I/O requests to read and write blocks of data and to control the (virtual) device by transforming each of these requests into one or more requests to disks that it makes through one or more HBAs. The volume manager is functionally equivalent to the aggregating disk controller depicted in Figure 1.15.

Like aggregating disk controllers, host-based volume managers can:

- *Concatenate* two or more disks into a single large volume.
- *Stripe* data across two or more disks for improved performance.
- *Mirror* data on two or more disks or striped volumes for availability.
- *Combine* several disks into a RAID volume.

Chapters 3 and 4 describe the capacity, performance, and availability characteristics of these popular volume types.