Active Disks - Remote Execution for Network-Attached Storage

Erik Riedel

November 1999
CMU-CS-99-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A Dissertation submitted to the
Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Thesis Committee:
David Nagle
Christos Faloutsos
Garth Gibson
Pradeep Khosla
Jim Gray, Microsoft Research

This research was sponsored by DARPA/ITO through Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by NSF under grants EEC-94-02384 and IRI-9625428 and NSF, ARPA and NASA under NSF Agreement IRI-9411299. Further support was provided through generous contributions from the member companies of the Parallel Data Consortium: Hewlett-Packard Laboratories, LSI Logic, Data General, Compaq, Intel, Quantum, Seagate, Wind River Systems, Siemens, and StorageTek. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

Keywords: storage, active disks, embedded systems, architecture, databases, data mining, disk scheduling

Dedicated to the extraordinary people without whom this would not have been possible,

my parents

Jürgen and Gerda Riedel

and their parents before them

Johannes and Gertrud Riedel

Albert and Anna Schumm

Abstract

Today's commodity disk drives, the basic unit of storage for computer systems large and small, are actually small computers, with a processor, memory, and 'network' connection, along with the spinning magnetic material that permanently stores the data. As more and more of the information in the world becomes digitally available, and more and more of our daily activities are recorded and stored, people are increasingly finding value in analyzing, rather than simply storing and forgetting, these large masses of data. Sadly, advances in I/O performance have lagged the development of commodity processor and memory technology, putting pressure on systems to deliver data fast enough for these types of data-intensive analysis. This dissertation proposes a system called *Active Disks* that takes advantage of the processing power on individual disk drives to run application-level code. Moving portions of an application's processing directly to the disk drives can dramatically reduce data traffic and take advantage of the parallelism already present in large storage systems. It provides a new point of leverage to overcome the I/O bottleneck.

This dissertation presents the factors that will make Active Disks a reality in the not-so-distant future, the characteristics of applications that will benefit from this technology, an analysis of the improved performance and efficiency of systems built around Active Disks, and a discussion of some of the optimizations that are possible with more knowledge available directly at the devices. It also compares this work with previous work on database machines and examines the opportunities that allow us to take advantage of these promises today where previous approaches have not succeeded. The analysis is motivated by a set of applications from data mining, multimedia, and databases and is performed in the context of a prototype Active Disk system that shows dramatic speedups over a system with traditional, "dumb" disks.

Acknowledgments

I would like to thank my advisors, David Nagle, Christos Faloutsos, Garth Gibson, Bernd Bruegge, and Mark Stehlik who helped guide me along the path I took through ten eventful and thoroughly enjoyable years at Carnegie Mellon. They presented me the opportunities that I have been fortunate enough to have and provided guidance and support throughout. They also put up with me as I learned.

I also thank Pradeep Khosla for agreeing to be on my committee and providing his insight and support. I thank Jim Gray who provided guidance from the world outside and gave me a glimpse of what being a visionary really means.

I thank Catharine van Ingen, Eric Schoen, and Peter Highnam for helping in my introduction to the real world during the course of two extremely valuable summers away from Pittsburgh. I thank Ted Russell and Greg McRae for providing and building the opportunity that originally kept me at Carnegie Mellon to pursue graduate school.

I thank Khalil Amiri and Howard Gobioff, the best pair of officemates, neighbors, research mates, and friends ever assembled in that combination. I also thank David Rochberg and Fay Chang who complete the Gang of Five grad students on the NASD project. We started our research careers together, and have grown together over many ups and downs. I will be the second from this group to successfully make my escape, and wish continued luck as the others follow and as we spread our wings from here.

I also thank other present and former members of the Parallel Data Lab including Jim Zelenka, Dan Stodolsky, Chris Demetriou, Jeff Butler, Charles Hardin, Berend Ozceri, Tammo Spalink, Mike Bigrigg, and Hugo Patterson for the many discussions on all topics and the experiences of all sorts over the years. I thank the Parallel Data Lab as a whole for providing me the environment in which I could pursue this work, without having to worry about where the funding would come from, or who would listen when I thought I had something interesting to say. I thank Patty Mackiewicz for making the whole organization go, and always having a smile ready. I also thank those who worked behind the scenes to make the PDL a success over the years, including Bill, Jenn, Paul, Sharon, Marc, Sean, Joan, Meg, Karen, and LeAnn.

I thank our industry partners, including all the members of the Parallel Data Consortium, for helping us to understand the issues, problems, and opportunities surrounding this work, particularly John Wilkes and Richard Golding of Hewlett-Packard, Satish Rege and Mike Leis of Quantum, and Dave Anderson of Seagate.

I thank the university as a whole for supporting me through three degrees under three administrations, and in particular the School of Computer Science and the Department of Electrical and Computer Engineering for providing an environment in which excellence is everywhere, and mediocrity is not tolerated. Many individuals have provided a hand, and I will thank just a few by name, including Catherine Copetas, Raj Reddy, Angel Jordan, Dan Sieworik, John Shen, Ron Bianchini, Thomas Gross, Peter Steenkiste, Mary Shaw, Phyllis Lewis, Phil Koopman, Greg Ganger, and Ava Cruse for providing variously inspiration, knowledge and support.

I thank all of the friends who have helped me and taught me in so many ways, and who put up with the long hours and the stress that creating a doctoral thesis and building a research career entails: Heather, Maureen, Hilary, Anne, Cheryl, Tara, Beth, Laurie, Laurie, Sandra, Todd, Neal, Samantha, Bob, Greg, Jens, Sean, Diana, Susan, Becky, Joe, and Doug. Thanks to you all. I could not have done this without you.

I thank my parents, to whom this thesis is dedicated, who gave me the basic tools I needed, and then set me free to pursue my goals as I saw them. Who provided the support that was quietly in the background and allowed me to look forward.

Yippee!

Erik Riedel
Pittsburgh, Pennsylvania
November 1999

Chapter 1: Introduction and Motivation

The cost and silicon real estate needed for any particular computational need is continually dropping. At some point, additional processing power can be had at negligible cost. The question then becomes simply where to place this computation power in a system to support the widest range of tasks efficiently. The contention of this work is that processing power is already moving into peripheral devices and that applications can achieve significant performance gains by taking advantage of this trend. Specifically, this work focuses on how data-intensive applications can directly exploit the processing power of the controllers in individual commodity disk drives to improve both individual application performance and system scalability.

The same trends in chip technology that are driving microprocessors toward ever-larger gate counts drive disk manufacturers to reduce cost and chip count in their devices while simultaneously increasing the total amount of local processing power available on each device. One use for this increasing computation power on disk controllers is to enrich their existing interface. For example, recent advances in network-attached storage are integrating storage devices into general-purpose networks and offloading a range of high-level functions directly to the devices. This eliminates servers as a bottleneck for data transfers between disks and clients and promises significant improved scalability through higher-level interfaces.

At the same time, as systems get faster and cheaper, people compute on larger and larger data sets. A large server system today will easily have a hundred disk drives attached to it. This large number of drives is necessary either to provide sufficient capacity or sufficient aggregate throughput for the target application. Taking this trend and extrapolating to future drive capabilities gives a promising picture for on-drive processing.

A pessimistic value for the on-drive processing already in today's commodity SCSI disk controllers is 25 MHz, with perhaps 15 MB/s of sustained bandwidth in sequential access. This means that a system with one hundred disks has 2.5 GHz of aggregate processing power and 1.5 GB/s of aggregate bandwidth at the disks. There are not many server systems today that can provide this level of computation power or I/O throughput. A typical multiprocessor system with one hundred disks might have four processors of 400 MHz each and 200 MB/s of total I/O throughput, much less than the aggregate 100-

disk values. Further extrapolating today's figures to next generation disks with 200 MHz processors and 30 MB/s transfer rates in the next few years, the potential power of a large disk farm is more than an order of magnitude more than the server system.

In addition, as storage is connected to a large collection of hosts by taking advantage of network-attachment and storage area networks, the interconnection network will rapidly become a principle bottleneck in large-scale applications. If data can be processed directly by the devices at the "edges" of the network, then the amount of data that must be transferred across this bottleneck can be significantly reduced.

This work proposes *Active Disks*, next-generation disk drives that provide an environment for executing application code directly at individual drives. By partitioning processing across hosts and storage devices, it is possible to exploit the cycles available at storage, reduce the load on the interconnection network, and perform more efficient scheduling. For example, an application that applies selective filters to the stored data and only ships summary information across the network, or that makes scheduling decisions based on local information at the individual drives can make more effective use of network and host resources. This promises both improved individual application performance and more scalable systems.

The thesis of this work is that:

> *A number of important I/O-intensive applications can take advantage of computational power available directly at storage devices to improve their overall performance, more effectively balance their consumption of system-wide resources, and provide functionality that would not otherwise be available.*

which will be supported in detail by the arguments in the chapters that follow.

This work addresses three obstacles to the acceptance of Active Disks within the storage and database communities. One objection that has been made to this work is that Active Disks are simply a reincarnation of the database machines that were studied extensively in the mid 1970s and 1980s that never caught on commercially and will not catch on now. A second objection is that the performance benefits possible through the use of Active Disks are too small to warrant wide interest. Finally, the third widely-heard objection is that the programming effort required to take advantage of Active Disks is too large, and that users will not be willing to make the necessary modifications to their code. This dissertation responds to each of these objections in turn.

The concept of Active Disks is very similar to the original database machines, but this dissertation will argue that the technology trends since then make this a compelling time to re-examine the database machine work and re-evaluate the conclusions made at the time. Active Disks are not special-purpose architectures designed for only a single application, but a general-purpose computing platform uniquely positioned within a system architecture - close to the stored data on which all applications operate and at the edge of the interconnection network that binds storage, computation, input, and output devices.

The core of this dissertation presents a model for the performance of applications in an Active Disk system, motivates a number of important data-intensive applications for further study, and presents the measured performance of a prototype Active Disk system with these applications. These chapters show both that dramatic benefits are possible in theory and are realistically achievable on a range of applications, including all the core functions of a relational database system.

Finally, the chapter on software structure will outline the basic structure of on-disk code in an Active Disk system and discusses the modifications made to a relational database system to allow it to take advantage of Active Disks. This answers the final objection and shows that the changes are straightforward and that extracting a "core" portion of data-processing code for execution on Active Disks can be accomplished with a reasonable amount of effort.

The dissertation commences with a study of background material in Chapter 2 and identifies the technology trends that make this work possible. Chapter 3 discusses the potential benefits of Active Disks and provides a model for estimating the speedups possible in an Active Disk environment. Chapter 4 discusses a number of compelling applications and describes their structure in an Active Disk context, including all the basic functions of a relational database system. Chapter 5 discusses the impact on performance and scalability of systems with Active Disks and provides results from a prototype system on this same set of applications. Chapter 6 describes the software structure of on-disk code and the changes necessary to allow an existing system to take advantage of Active Disks. Chapter 7 addresses a number of additional issues, including the details of how to efficiently support applications inside Active Disks. Chapter 8 revisits the work on the database machines that were extensively studied a decade and a half ago, but did not have the technology drivers to make them successful at the time. Chapter 9 discusses additional areas of related and complimentary work. Finally, Chapter 10 concludes and discusses areas of, and challenges for, future work.

Chapter 2: Background and Technology Trends

A proposal to perform data-intensive operations using processing elements directly attached to disk drives may seem familiar to anyone who has worked in the database area as the *database machines* that dominated database research in the 70s and 80s. The concept of Active Disks is close to the ideas pursued in the work on database machines and many of the lessons from that work are applicable today. Since that time, technology trends in several areas have shifted the picture considerably, providing an opportunity to revisit the arguments explored in these projects.

The major changes since the time of the original database machines are the performance of individual disks (which has increased thirty-fold), the cost of integrated circuits (which has decreased by several orders of magnitude), and the availability of *mobile code* (that allows the motion of not just data, but also code and "function" among the components of a computer system). What has not changed is the desire of users to store and operate on more and more data. The capacity of individual devices has increased 200-fold, but the desire of users and applications for more storage space has not abated.

This chapter outlines the database machine architectures proposed years ago to set the stage for their rebirth as Active Disks. Then it examines the trends in technology that have made storage fast and processing power cheap. Next, it details the trends in user requirements and applications that call for higher data rates and more data-intensive processing. Finally, it surveys the state of mobile code technology and the crucial role that it will play in making high-function devices widely applicable.

2.1 Database Machines

This section provides a brief tour of some of the database machine technology studied during the mid-70s to the late 80s. More detailed discussion of specific projects and how their results may aid the design of Active Disk systems is provided in Chapter 7.

2.1.1 Specialized Hardware

The database machines proposed and developed in the 70s and 80s consisted of specialized processing components that performed portions of the function of a database system. These devices were custom-designed to implement a particular algorithm and assumed a particular set of queries and a particular data layout. There are several classes

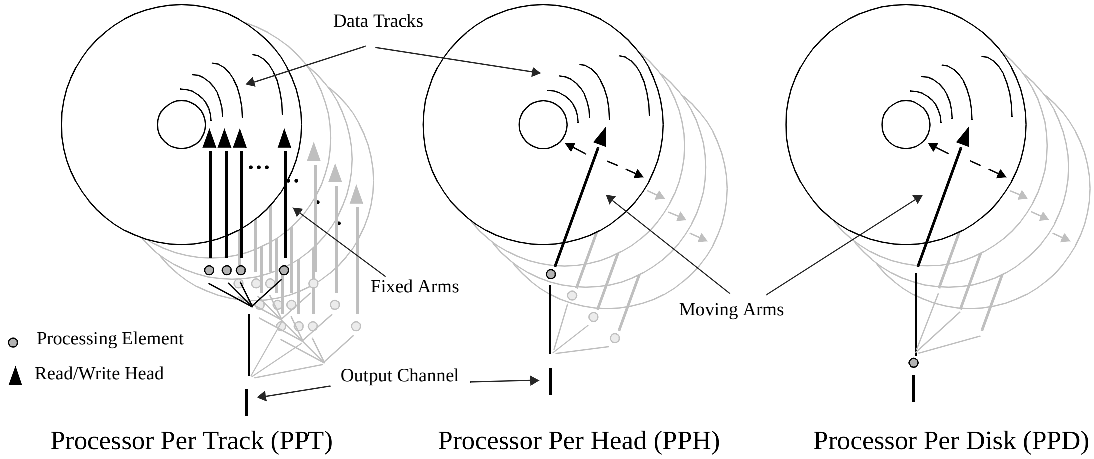


Figure 2-1 Architectural diagram of several database machine architectures. The diagram illustrates the three most popular database machine architectures over a decade of database machine research.

of machines, with differing levels of complexity. The full taxonomy is presented in Chapter 7; the intent of this section is simply to give a brief overview of a sampling of machines.

A survey paper by David DeWitt and Paula Hawthorn [DeWitt81] divided the space into several architectures based on where the processing elements were placed relative to the data on the disk, as illustrated in Figure 2-1. The processing elements could be associated with each track of the disk in a fixed-head device (processor-per-track, or PPT), with each head in a moving-head system (processor-per-head, or PPH), with the drive as a whole (processor-per-disk, PPD), or without any special processing elements (conventional system, CS). Each of these architectures depended on a control processor that acted as a front-end to the machine and accepted queries from the user. In the conventional systems, the front-end performed all the processing while in all the other systems processing was split between the front-end and the specialized database machine logic attached directly to the disk.

As an example of how these architectures operate, Figure 2-2 illustrates a `select` operation in the RAP database machine. A `select` searches for records that match a particular condition (e.g. `state = PA`, to find all customers in Pennsylvania). In order to perform a select using RAP, the search condition is loaded into the logic, which consists of *k* comparators at each track. This means that *k* comparisons are performed in parallel and the entire database can be searched in one revolution of the disk. If a record on a particular track matches the search condition, it is output to the front-end. However, since the output channel has a limited capacity, it may not be possible for multiple tracks to report their matches at the same time. In the case of contention for the output channel, the matching recorded is marked, using mark bits on the disk, for output on a future revolution. This

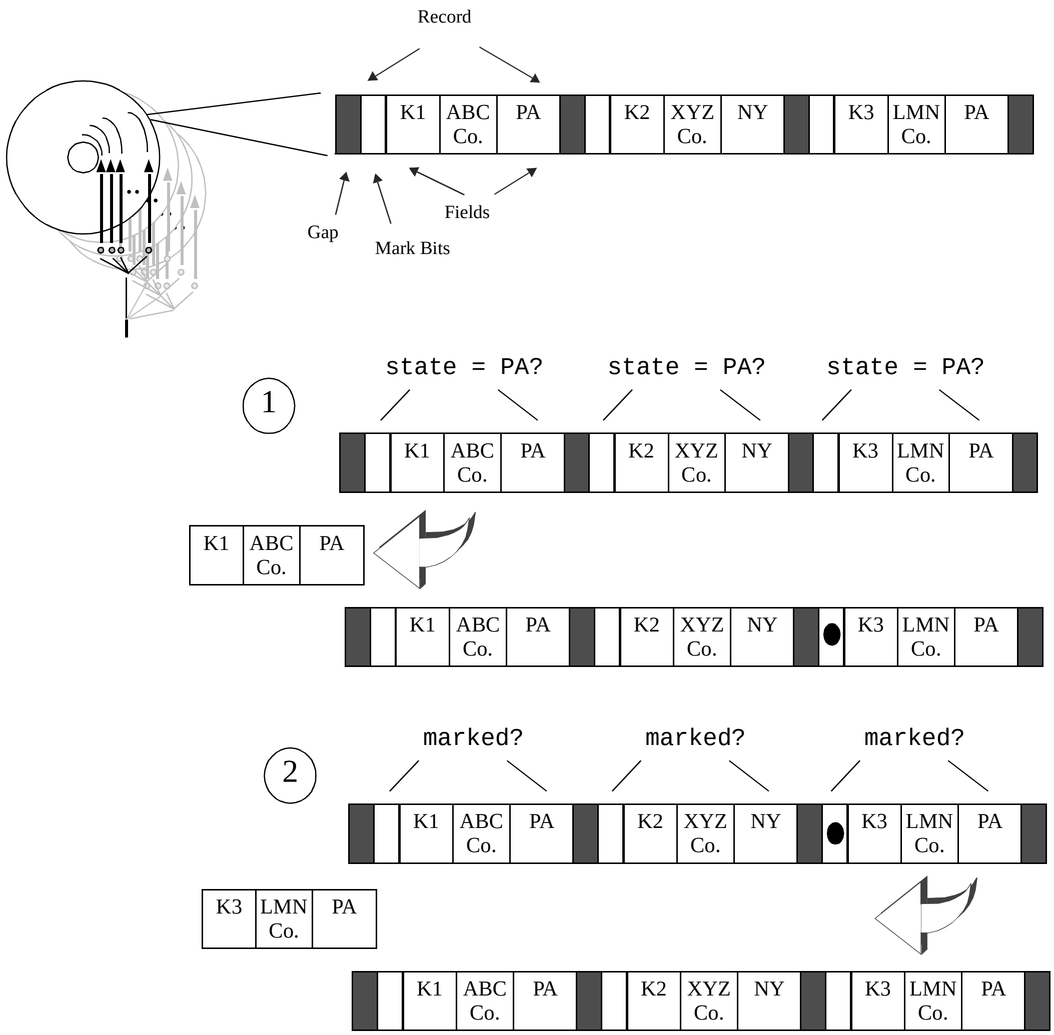


Figure 2-2 Select operation in the RAP machine. The diagram at the top illustrates the format of a data track in RAP. The lower diagram shows two passes of a select operation. All the records are searched in parallel on the first pass across the track, and matching records are output. If there is contention for the output channel, additional matching records are marked, and output on the second pass.

means that a condition with many matches (i.e. low *selectivity*[1]), it may require several revolutions before all of the matching records are output.

In addition, if more than *k* conditions are required for a particular query, multiple passes across the data must be made. This is particularly complex in the case of joins, where the length of the search condition is determined by the keys of a second relation. The straightforward procedure for joins is to load the first *k* keys of the inner relation and

1. This dissertation will use the term *selectivity* in a way that is different than normal usage in the database literature. As a parameter of an Active Disk application, *selectivity* will mean the amount of data reduction performed by the application, with higher values meaning greater reductions. Applications with the highest selectivities will perform the greatest data reductions, and applications with low selectivities will transfer the most data on the interconnect. In other contexts, selectivity is expressed as a percentage of data transferred, with lower values being better. Apologies for any confusion.

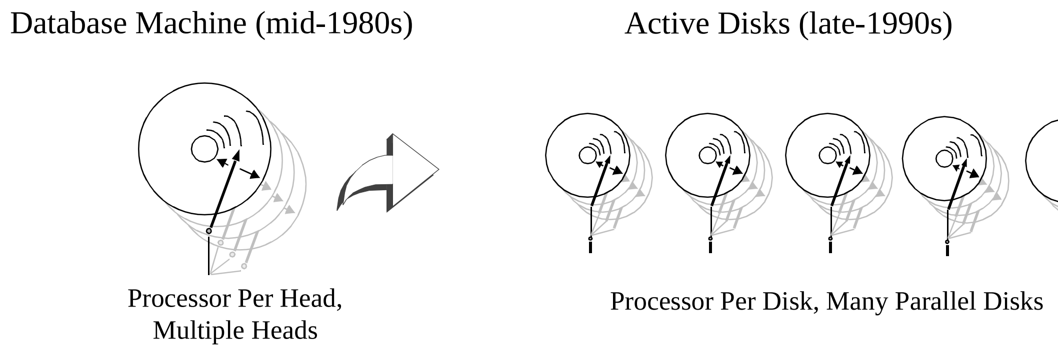


Figure 2-3 Evolution of database machines to Active Disks. The proposal for Active Disks is most similar to the Processor-Per-Head database machines, but using multiple commodity disks, each a single Processor-Per-Disk design instead of a processor per head on each disk.

search, then replace them with the next *k*, and so on. This means it will require *n/k* revolutions to search for a total of *n* keys.

The primary difficulty with the processor-per-track system is that it requires a huge number of processing element, along with a huge number of read/write heads to operate on all the tracks in parallel. This is what makes a processor-per-head system much more attractive, for the same reason that fixed-head disks were quickly replaced by moving-head disks. The cost of the read/write element can be amortized over a much larger amount of data, at the expense of having to *seek* to reach different locations on the disk. Processing can still be done in parallel as data is read from the disk platters, but there are many fewer processing elements. The disk gives up the ability to read any track with equal delay, in order to greatly reduce the total cost. Disk drives at the time these systems were proposed contained hundreds of tracks per surface while drives today contain several thousand. The processor-per-track technology clearly depended on the rapid development of bubble memory or other, similarly exotic technologies (see Chapter 8 for additional examples), that have not come to pass.

This leaves the processor-per-head and processor-per-disk systems as viable options. Since the time of the database machines, disk drives have evolved to a point where they still have multiple heads (one for each recording surface), but only one head can be actively reading or writing at a time. The increasing density and decreasing inter-track spacing makes it necessary to perform a significant amount of micro-actuation to correctly position the head even as data is being read. This means that the arm can "follow" only one track at a time, making it infeasible to read from multiple heads at the same time.

The proposal for Active Disks creates what is essentially a processor-per-disk machine, but is more comparable to the processor-per-head design because systems today contain many individual disk drives operating in parallel, while the original database machines were based on single disks. This evolution is traced in Figure 2-3. Each individ-

System	Use	Processor	Memory	I/O System	Disks
Compaq ProLiant 5500 6/400	TPC-C, OLTP	4 x 400 MHz Xeon	3 GB	32-bit PCI	141 disks = 1.3 TB
Digital AlphaServer 4100	Microsoft TerraServer, Satellite Imagery	8 x 440 MHz Alpha	4 GB	2 x 64-bit PCI	324 disks = 1.3 TB
Digital AlphaServer 1000/500	TPC-C, OLTP	500 MHz Alpha	1 GB	64-bit PCI,	61 disks = 266 GB
Digital AlphaServer 8400	TPC-D 300, DSS	12 x 612 MHz Alpha	8 GB	2 x 64-bit PCI	521 disks = 2.2 TB

Table 2-1 Example of several large database systems. We see that these systems have only a small number of processors, but a large number of individual disk drives. Data from [TPC98] and [Barclay97].

ual disk has only a single head (and processor) operating at a time, but all the heads (and processors) in a group of disks can operate in parallel.

The processor-per-disk design was dismissed out-of-hand by Hawthorn and DeWitt since it used a single, less powerful processing element to perform functions that could be done much more efficiently in the front-end. It is true that, in a system with a single disk, there is no parallelism benefit, and use of the additional logic has only minor benefits. Chapter 3 will show that this is also true for Active Disk systems. If there is only a single disk, the performance benefits are relatively small. However, storage systems today contain tens to hundreds of disk drives, as shown in Table 2-1.

There are many additional database machines discussed in Chapter 8, but the most interesting and long-lived is probably CAFS (content-addressable file store) and SCAFS (son of CAFS) from ICL, which provide search acceleration for relational database systems. SCAFS went through several generations of technology and was being sold, in a 3.5” form factor that looked very much like a disk drive, plugged into a drive cabinet, and communicated through a SCSI interface into the mid-90s. These devices were available as optional components on Fujitsu, ICL, and IBM mainframes. A report from ICL estimates that at one point up to 60% of these systems shipped with the accelerators installed [Illman96]. The accelerator was a specialized processor in the same cabinet as a number of commodity SCSI disk drives and was addressed as if it were a disk itself. Several versions of the INGRES, Informix, and Oracle database systems had extensions that allowed them to take advantage of the accelerator when appropriate. The accelerator was primarily used for large scans, and provided significant gains across a range of workloads [Anand95]. This architecture is similar to the proposal for Active Disks, but Active Disks go further by providing computation power on each individual disk drive, rather than across a number of disks in an array, and by allowing full, general-purpose programmability of the additional logic.

Table 2-2 traces the evolution of database machine architectures. We see that the amount of storage and amount of logic increased with trends in silicon and chip integration. Also note the change from specialized processing elements to general-purpose processors in the mid-1980s.

Architecture	Year	Disk	Memory	Logic	Size	Nodes	Network	Notes
RAP.2	1977	4 Mbits	1 K x 16 bits	64 ICs	412 ICs	2		“disk” is CCD
CASSM	1976				220 ICs	1		prototype development suspended, simulation only
CAFS	198x				50,000 transistors			first in VLSI technology
DIRECT	198x	32x16K	28K words	lsi 11/03		8	6 MB/s	“disk” is CCD, has additional mass storage
GAMMA 1.0	1985	333 MB	2 MB	vax 11/750		17	80 Mbit/s	
GAMMA 2.0	1988	330 MB	8 MB	i386	~300,000 transistors	32		
SCAFS	1994						10 MB/s	“son of CAFS”
Active Disk	1999	18 GB	32 MB	StrongARM	2.5 million transistors		100 MB/s	proposed system

Table 2-2 Sizes and functionalities of various database machine architectures. The chart shows the size and functionality of several database machine architectures from the original RAP in 1977 to Active Disks today. Data from [Schuster79], [Su79], [DeWitt79], [DeWitt90], and [Illman96].

2.1.2 Performance

A performance evaluation by DeWitt and Hawthorn [DeWitt81] compares the relative performance of the various architectures for the basic database operations: select, join and aggregation. In their comparison, the PPT and PPH systems performed significantly better on searches with high selectivities, where contention for the output channel was not an issue, and on non-indexed searches where the massive parallelism of these systems allowed them to shine. When indexed searches were used, the benefit was not as large. The systems with less total processing power were able to “target” their processing more effectively. They had a smaller number of more powerful processing elements that could be applied more selectively, while the PPT and PPH systems simply had an excess of computation power “lying around” unused. For join operations, the key parameter is the number of keys that the PPT and PPH machines can search at a time. For large relations and small memory sizes, this can require a significant number of additional revolutions.

Similar characteristics apply to Active Disk processing and are discussed in more detail in the subsequent chapters. The full details of the performance study conducted by DeWitt and Hawthorn, as well as modifications to some of the pessimistic assumptions they made with respect to the PPT and PPH architectures, are discussed in Chapter 8.

2.2 Changes Since Then

In a paper entitled “Database Machines: An Idea Who’s Time Has Passed?”, Haran Boral and David DeWitt proclaimed the end of database machines on the following set of objections:

- that a single host processor (or at most two or three) was sufficient to support the data rate of a single disk, so it was unnecessary to have tens (in PPH) or thousands (in PPT) of processors

- that the specialized database machine hardware elements were difficult to program, requiring specialized microcode which was difficult to develop, and
- that the simplest database machines, while good for simple scans, did not efficiently support the more complex database operations such as sorts or joins [Boral83]

The next several sections will explore each of these arguments in turn and explain how technology has changed to overcome these objections. First, disks are much faster, and there are a lot more of them than there were in 1983. Second, there is now a compelling general-purpose mechanism for programming devices. Finally, many of today's most popular applications require data-intensive scans over more complex sorts or joins and modern processing elements can support joins and sorts as well as simple scans, although the speedups are indeed less dramatic than for scans.

2.2.1 Disk Rates

The speed of a single disk drive has improved considerably since the days of the database machines, as shown in Table 2-3. Bandwidth from a top-of-the-line disk is now

	1980	1987	1990	1994	1999	80-99	80-87	87-90	90-94	94-99
Model	IBM 3330	Fujitsu M2361A	Seagate ST-41600n	Seagate ST-15150n	Quantum Atlas 10k	Annualized Rate of Improvement				
Average Seek	38.6 ms	16.7 ms	11.5 ms	8.0 ms	5.0 ms	11%/yr	13%/yr	13%/yr	9%/yr	10%/yr
Rotational Speed	3,600 rpm	3,600 rpm	5,400 rpm	7,200 rpm	10,000 rpm	6%/yr	0%/yr	15%/yr	7%/yr	7%/yr
Capacity	0.09 GB	0.6 GB	1.37 GB	4.29 GB	18.2 GB	32%/yr	30%/yr	32%/yr	33%/yr	34%/yr
Bandwidth	0.74 MB/s	2.5 MB/s	3-4.4 MB/s	6-9 MB/s	18-22.5 MB/s	20%/yr	19%/yr	21%/yr	20%/yr	20%/yr
8 KB Transfer	65.2 ms	28.3 ms	18.9 ms	13.1 ms	9.6 ms	11%/yr	13%/yr	14%/yr	10%/yr	6%/yr
1 MB Transfer	1,382 ms	425 ms	244 ms	123 ms	62 ms	18%/yr	18%/yr	20%/yr	19%/yr	15%/yr

Table 2-3 Disk performance parameters. The table compares a number of parameters of commodity disk drives from 1987 to 1999. We see that the rate of improvement is relatively constant across the periods listed, and consistent across the entire period. Data for 1980 is from [DeWitt81], data for 1987 and 1990 from [Gibson92], and data for 1994 from [Dahlin95a].

more than 30 times that of the disk used in the [DeWitt81] study to conclude that database machines were past their time. Seek time and overall latency for small requests has not increased nearly as much as sequential bandwidth or the latency for large requests. In addition, capacity has increased by a factor of 200.

The most noticeable disparity in Table 2-3 is that read bandwidth is increasing much more slowly than the media capacity. This divergence occurs because increases in areal density occur in two dimensions while linear read rate increases in one dimension with the decreasing size of magnetic domains on the media. This means that bandwidth will increase less quickly than areal density. In fact, increases in density of tracks per inch actually decreases bandwidth somewhat due to the increased complexity of micro-actuation and track-following required during track and head switches. The overall trend is illustrated graphically in Figure 2-4 which shows the diverging capacity and bandwidth

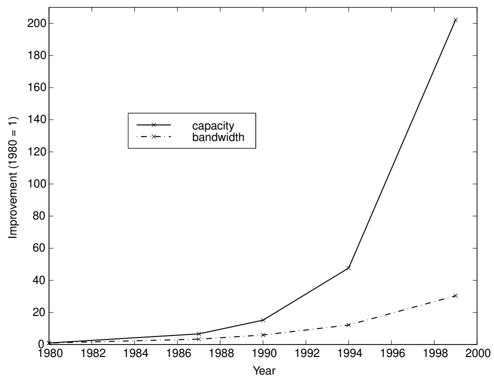


Figure 2-4 Trends in disk capacity and bandwidth. The chart compares the capacity and read bandwidth of the disks listed in the table above, spanning 20 years of disk technology. We see that the capacity of the disks increases much faster than the bandwidth. Although much more data is stored on a single disk, it takes progressively longer to read it. If we consider the time it takes to read the entire disk, this was just over 2 minutes in 1980, and is almost 14 minutes in 1999. Note that this is reading sequentially, reading in random 8 kilobyte requests is much worse, taking 13 minutes in 1980 and more than 6 hours in 1999.

lines over the last 20 years. We see that capacity has increased 200-fold while bandwidth has increased only 30-fold. If we consider a metric of *scan time* (how long it takes to read the entire disk), we can calculate that it took only two minutes to read the 1980 disk, whereas it takes more than fourteen minutes for the 1999 disk. This assume that we are reading sequentially using large requests, if we instead read the entire disk in small, random requests, it would have taken thirteen minutes in 1980, but would take more than six hours today.

While this analysis paints a bleak picture for disk drive technology, there is one factor that is omitted from Table 2-3 but which is near and dear to the heart of almost any system designer, and that is cost. Figure 2-5 illustrates the trend in the cost of storage, plotting the cost per megabyte of disks over the period from 1982 to today. The cost has gone from \$500 per megabyte to about 1.5 cents per megabyte in the space of 17 years - a

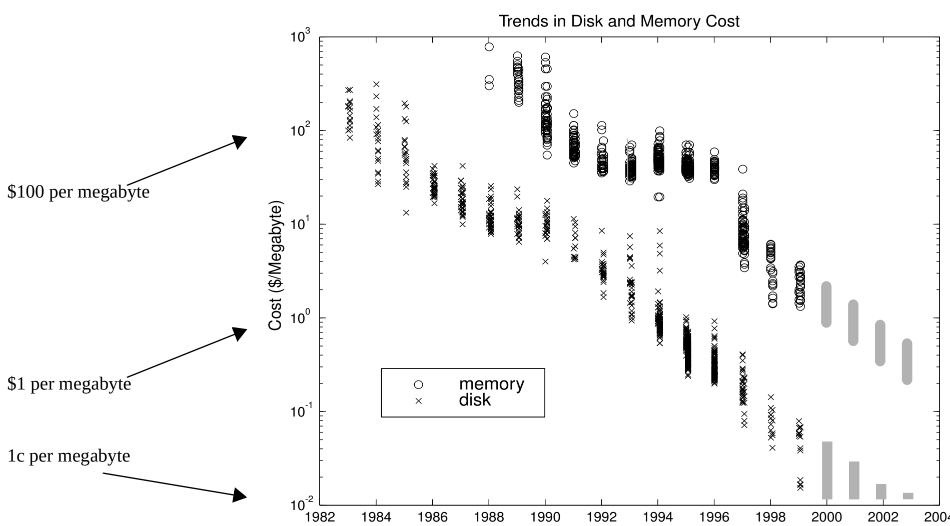


Figure 2-5 Trends in disk and memory costs. Figures before 1998 are from [Dahlin95], advertised prices in *Byte* magazine in January and July of the year listed. *Byte* ceased publication in 1998, so figures for 1998 and 1999 are from similar advertisements in *PC Magazine*. Numbers after 1999 are extensions of the previous trend lines.

30,000-fold reduction. This means that much larger database can be built today than were even dreamt of in the 1980s. As we will see in the discussion of specific application areas below, some of the specific dreams of the time are becoming reality today, and are driving the requirements for storage system design.

It is, however, important to remember the trend of Figure 2-4 which makes it clear that there are now two parameters to consider when purchasing a disk system 1) how much capacity is required to store the data and 2) how much bandwidth is required to practically make use of the data. In many cases, as we will see, it is necessary to purchase more disks than what is required simply for capacity in order to get the necessary level of aggregate bandwidth (and performance). This is true for sequential workloads, where absolute bandwidth is the primary metric, and even worse for random workloads where seek time and small request latency dominates.

2.2.2 Memory Sizes

There is a second piece of good news for disk technology in the data of Figure 2-5 and that is the data at the top of the graph showing the cost per megabyte of commodity memory chips. Computer architects have often proclaimed the imminent replacement of disks by large memories, but the chart shows that disk technology has consistently been able to stay ahead on the cost curve.

However, even with this fixed separation, the lines do show the same trend, the price of memory is falling at roughly the same rate as the price of disks. This growth rate in amount of memory per dollar cost has led many to proclaim the advent of main memory databases to take over many of the functions that traditionally required the use of disk storage [Garcia-Molina92, Jagadish94]. This trend has clearly helped transaction processing workloads with relatively small working sets - proportional to the number of “live” customer accounts or the number of unique items in a store’s inventory, for example - but does not address the data growth when historical records - all transactions since a particular account was opened or a record of every individual sale made by a large retail chain over the course of a year - are taken into account.

This means that in the context of a transaction processing workload, disk storage is no longer the primary driver of performance. If a large fraction of the live database can be stored in a very large memory system (systems with up to 64 gigabytes of main memory are available today [Sun98]), then the disks are necessary only for cold data and permanent storage, to protect against system failures. This means that optimizations in this area are primarily focussed on write performance and reliability. Optimizations such as immediate writes [Wang99] or non-volatile memory for fast write response [Baker92] are the most helpful[1]. These types of optimization can also benefit from increased intelligence at the individual devices to implement a variety of application-optimized algorithms (such as

1. but not for individual disk drives, where the performance benefits of non-volatile memory do not yet justify the increase in cost [Anderson99].

the one discussed in Section 5.4), but not from the same parallelism and bandwidth reduction that are the focus here and in the following chapters.

2.2.3 Aggregation/Arrays

When a database fits on a single disk, the general-purpose processor in front of it can easily keep up, but systems today require multiple disks for several reasons. The first is simply capacity: if your database does not fit on one disk, you have to wait until the areal density increases, or buy additional disks. The second reason was discussed above: the need to use multiple disks to provide higher bandwidth access to the data. The third reason has not been mentioned yet, but comes up as a consequence of the first two, and that is reliability.

The reason that the use of multiple disks to form a single coherent store is widespread today is as a result of the development of disk array hardware and software [Livny87, Patterson88]. The core idea of this work is to replace a single large disk (with the best, fastest, and densest technology money can buy) with a number of smaller, less expensive disks (that are manufactured in larger quantities and benefit from economies of scale, but are individually slower or less dense). Or less reliable. This last point was the key insight of using a redundant array of inexpensive disks (RAID). Using a variety of techniques for error correction and detection, RAID provides a way to make a large number of individual disks, with perhaps low individual reliabilities, into a coherent storage sub-system with a much higher aggregate reliability. Through the use of a variety of encoding schemes, some amount (up to 50%) of the disks' capacity is given up to keep redundant copies of the user data. Depending on the design of the system, this means that the failure of any single disk (or even a small subset of the disks), does not cause user data to be lost. The failed disk is then replaced and the damaged information reconstructed without user intervention and without stopping the system. This mechanism provides the benefits of increased overall capacity and increased bandwidth, while retaining a high level of reliability. RAID technology is now sufficiently well understood and established that it comes standard with many classes of computer systems and that software RAID is packaged as a basic service in some operating systems (e.g. Windows NT 4.0).

The simplest form of array uses two disks, with one operating as a *mirror* copy of the other. This is the most expensive in terms of space, since twice the disk capacity must be used for any given amount of user data. The mechanism is straightforward, in that each write is simply duplicated on the mirror disk, and a read can be serviced by either disk in the pair [Bitton88]. The prevailing disk trends make the capacity overhead less critical, and make mirroring an attractive solution for fault-tolerance, although larger numbers of disks must still be combined to provide the aggregate capacity necessary to store today's large data sets.

System	Processor	Host Processing	Disks	On-DIsk Processing Today	Disk Advantage	On-Disk Processing Soon	Disk Advantage
Compaq TPC-C	4 x 400 MHz	1,600 MHz	141	3,525 MHz	2.2 x	28,200 MHz	17.6 x
Microsoft TerraServer	8 x 440 MHz	3,520 MHz	324	8,100 MHz	2.3 x	64,800 MHz	18.4 x
Digital TPC-C	1 x 500 MHz	500 MHz	61	1,525 MHz	3.0 x	12,200 MHz	24.4 x
Digital TPC-D 300	12 x 612 MHz	7,344 MHz	521	13,025 MHz	1.3 x	104,200 MHz	14.2 x

Table 2-4 Comparison of computing power vs. storage power in large server systems. Estimating that current disk drives have the equivalent of 25 MHz of host processing speed available, large database systems today already contain more processing power on their combined disks than at the server processors. Extending this to the 200 MHz processors that will be available in the near future gives the disks a factor of 10 and 20 advantage.

2.2.4 Silicon

The second objection to database machines was the cost and complexity of the special-purpose hardware used in these machines. Here again technology trends have changed the landscape. The increasing transistor count in inexpensive CMOS is driving the use of microprocessors in increasingly simple and inexpensive devices. Network interfaces, digital cameras, graphics adapters, and disk drives all have microcontrollers whose processing power exceeds the most powerful host processors of 15 years ago. Not to mention the cellular phones, microwave ovens, and car engines that all contain some type of microprocessor completely outside the realm of the traditional computer system [ARM99]. A high-end Quantum disk drive of several years ago contains a Motorola 68000-based controller that is solely responsible for managing the high-level functions of the drive. This is the same microprocessor that Boral and DeWitt suggested in 1983 would be sufficient to handle *all* of the database processing in several years time [Boral83]. If we consider this change with respect to the machines in Table 2-1, and assume a modest 25 MHz of processing power at the individual disk drives, we see that these large data systems already have more than two or three times as much aggregate processing power at the disks as at the hosts, as shown in Table 2-4.

2.2.5 Drive Electronics

Figure 2-6 shows the effects of increasing transistor density and integration on disk drive electronics. In Figure 2-6a, we see that the electronics of a disk drive include all the components of a simple computer: a microcontroller, some amount of RAM, and a communications subsystem (SCSI), in addition to some specialized hardware for drive control. Figure 2-6b shows how a number of these special-purpose control chips have been integrated into a single piece of silicon in current-generation drives. The figure then extrapolates to the next generation of process technology (from .68 micron to .35 micron CMOS in the ASIC). The specialized drive control hardware now occupies about one quarter of the chip, leaving sufficient area to integrate a powerful control processor, such as a 200 MHz StrongARM [Turley96], for example. Commodity disk and chip manufacturers

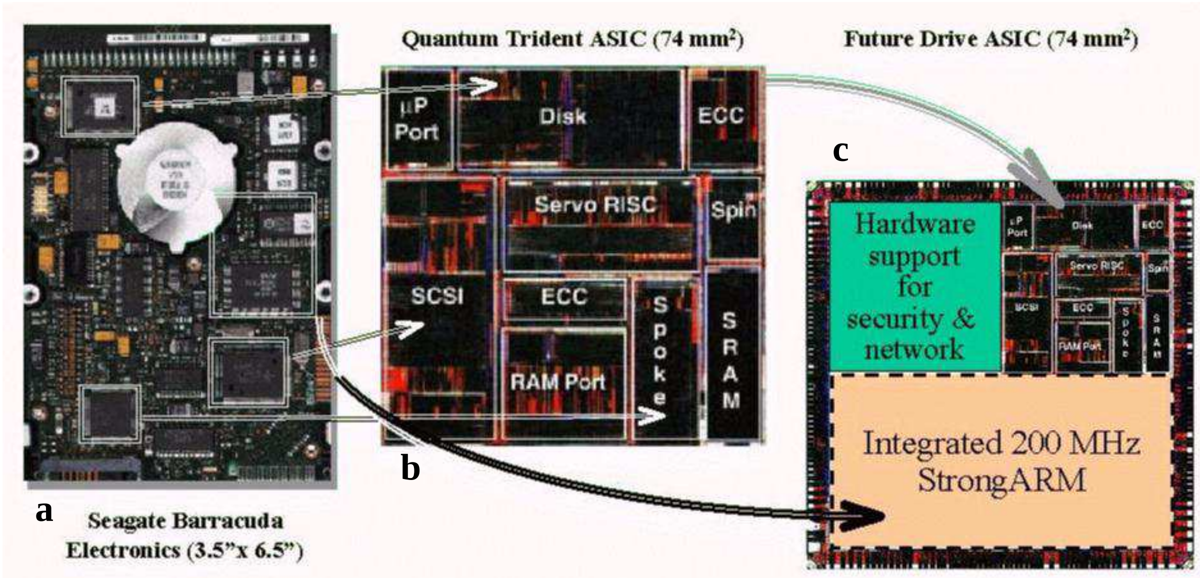


Figure 2-6 The trend in drive electronics toward higher levels of integration. The Barracuda drive on the left contains separate chips for servo control, SCSI processing, ECC, and the control microprocessor. The Trident chip in the center has combined many of the individual specialized chips into a single ASIC, and the next generation of silicon makes it possible to both integrate the control processor and provide a significantly more powerful embedded core while continuing to reduce total chip count [Elphick96, Lammers99].

are already pursuing this type of integration. Siemens Microelectronics has produced first silicon for a family of chips that offer a 100 MHz 32-bit microcontroller, up to 2 MB of on-chip RAM, external DRAM and DMA controllers and customer-specific logic (that is, die area for the functions of Figure 2-6b) in a .35 micron process [Siemens97, Siemens98]. Cirrus Logic has announced a chip called 3Ci that incorporates an ARM core on the same die as the drive control circuitry to provide a system-on-a-chip controller as a single part [Cirrus98]. The first generation of this chip contains an ARM7 core, and the next generation promises a 200 MHz ARM9 core.

VLSI technology has evolved to the point that significant additional computational power comes at negligible cost. Table 2-5 compares the performance and size of several generations of the ARM embedded processing core [ARM98]. We see that today's embed-

Chip	Speed	Dhrystone	Cache	Process	Size	Available	Notes
ARM7TDMI	66 MHz	59 MIPS	-	0.35 um	2.1 mm^2	now	core only
ARM710T	59 MHz	53 MIPS	8K unified	0.35 um	9.8 mm^2	now	simple memory protection
ARM740T	59 MHz	53 MIPS	8K unified	0.35 um	11.7 mm^2	now	full MMU/virtual memory
ARM940T	200 MHz	220 MIPS	8K unified	0.25 um	8.1 mm^2	now	MMU, 4 GB addr spc
ARM10	300 MHz	400 MIPS	32K/32K	0.25 um		mid 1999	including FP unit
StrongARM	200 MHz	230 MIPS	16K/16K	0.35 um	50.0 mm^2	now	
Alpha 21064	133 MHz		8K/8K	0.75 um	234.0 mm^2	now	75 SPECint92
StrongARM	600 MHz	750 MIPS				2000	

Table 2-5 Several generations of ARM 32-bit cores. The ARM7TDMI is included in the current Cirrus Logic 3Ci chip. ARM9 is shipping, and the ARM10 is the next generation.

ded chips provide nearly the power of the workstations of several years ago. In terms of raw integer instruction throughput, the embedded cores approach that of much more complex workstation processors. The decreasing feature size also makes this level of processing power available in ever smaller form factors, easily allowing such cores - and the ability to execute arbitrarily complex code - to be included in a single die with the much more specialized silicon optimized for particular functions like drive control. This combination allows the performance of specialized ASICs for those functions that are already available in silicon, while retaining a sufficient amount of general-purpose processing power for additional functions provided in software.

2.2.6 Database Algorithms

The final objection of Boral and DeWitt was that the simple hardware implementations in the database machines were not sufficient to support complex database operations such as sorts and joins. As part of the work on database machines after the "time has passed" paper, including work by Boral and DeWitt and their colleagues, many different solutions to this have been proposed and explored. The concept of a shared nothing database "machine" is now a well-established concept within the database community, and much work has gone into developing the parallel algorithms that make this feasible. A survey paper by DeWitt and Gray [DeWitt92] discusses the trends and success of shared nothing systems, including commercial successes from Tandem and Teradata, among others. The authors point to the same objection from [Boral83] and provide compelling evidence that parallel database algorithms have been a success.

Chapter 4 will explore the operations in a relational database system in detail, and illustrate how they can be mapped onto an Active Disk architecture. The basic point is that there are now known algorithms that can operate efficiently in this type of architecture. In addition, many of the data-intensive applications that are becoming popular today rely much more heavily on "simple" scans and require much more brute-force searching of data, because the patterns and relationships in the data are not as well understood as they are in the more traditional relational database systems on transaction processing workloads. Finding patterns in image databases, for example, is a much different task than in structured customer records. The next sections will discuss this change in applications and motivate the benefits to a system that can support more efficient parallel scans than today's systems.

2.2.7 Interconnects

There are several issues raised in the analysis of the database machines that were not specifically addressed in most of the existing work, and are still issues today. The most important one is contention for network bandwidth to the front-end host. This bottleneck persists today and is a primary reason why Active Disk systems, with processing at the "edges" of the network can be successful. Many types of data processing queries reduce a large amount of "raw" data into a much smaller amount of "summary" data that answers a

System	System Bus	Storage Throughput	Mismatch Factor
Compaq ProLiant TPC-C	133 MB/s	1,410 MB/s	10.6 x
Microsoft TerraServer	532 MB/s	3,240 MB/s	6.1 x
Digital AlphaServer TPC-C	266 MB/s	610 MB/s	2.3 x
Digital AlphaServer TPC-D 300	532 MB/s	5,210 MB/s	9.8 x

Table 2-6 Comparison of system and storage throughput in large server systems. If we estimate a modest 10 MB/s for current disk drives on sequential scans, we see that the aggregate storage bandwidth is more than twice the (theoretical) backplane bandwidth of the machine in almost every case.

particular user question and can be easily understood by the human end-user. Queries such as: *How much tea did we sell to outlets in China? How much revenue would we lose if we stopped marketing refrigerators in Alaska? How much money does this customer owe us for goods we sent him more than two months ago?* The use of indices can speed up the search for a particular item of data, but cannot reduce the amount of data that must be returned to the user in answer to a particular query. This is what led to the contention for the output channel in the early processor-per-track database machines, and it is also the bottleneck in many large storage systems today.

Instead of being limited by the bandwidth of reading data from the disk media, modern systems often have limited peripheral interconnect bandwidth, as seen in the system bus column of Table 2-6. We see that many more MB/s can be read into the memory of a large collection of disks than can be delivered to a host processor.

The interconnection "networks" used between storage devices and hosts and those used among hosts have long had somewhat different characteristics. A technology survey paper by Randy Katz [Katz92] breaks the technology into three distinct areas: backplanes, channels, and networks. Where backplanes are short (about 1 m), with bandwidth over 100 MB/s, sub microsecond latencies, and highly reliable; channels are longer (small tens of meters), support up to 100 MB/s bandwidth, have latencies under 100 microseconds, and medium reliability; and networks span kilometers, sustain 1 to 15 MB/s, and have latencies in milliseconds, with the medium considered unreliable, requiring the use of expensive protocols above to ensure reliable messaging. These distinctions are no longer as true as it was at the time of this survey. The need to connect larger numbers of devices and larger numbers of hosts over larger distances, has led to the development of the much more "network-like" Fibre Channel for storage interconnects. The growing popularity of Fibre Channel for storage devices and packet-switched networks for local- and wide-area networks has clouded the boundaries of peripheral-to-host and host-to-host interconnects. Since both Fibre Channel and Fast or Gigabit Ethernet, the storage and networking interconnect technologies of choice respectively are based on packets, switches, and run over the same fiber optic infrastructure. Why continue to artificially separate the two systems?

This is the contention of previous work at Carnegie Mellon on Network-Attached Secure Disks (NASD) [Gibson97, Gibson98] and is coming close to reality by the introduction of Storage Area Networks (SANs) in industry [Clariion99, Seagate98, StorageTek99]. Industry surveys estimate that 18% of storage will be in SANs by the end of 1999, reaching up to 70% within two years [IBM99].

Even though individual point-to-point bandwidths have increased greatly and latency has decreased significantly, the network connectivity in a distributed system will continue to be a bottleneck. It is simply too expensive to connect a large number of devices with a full crossbar network. This means that systems will need to take advantage of locality of reference in order to manage the inherent bottlenecks, but it also means that certain access patterns will always suffer from the bottlenecks among nodes. The cost comparison is clear when one compares the cost of a hierarchical switched system against a full crossbar system as the number of nodes in a system is increased. For a small number of nodes, a local switched fabric is quite effective, but as soon as the number of nodes exceeds the capacity of a single switch, the costs of maintaining a full crossbar increase rapidly. Switches must be deployed in a way that requires most of their ports to be dedicated to switch-to-switch connectivity, rather than to support end nodes. This greatly increases the cost of the system to the point where it becomes prohibitive to provide that level of connectivity in systems of more than one hundred nodes. This means that networks must either be limited to the size of the largest crossbar switch currently available, or must attempt to take advantage of locality in some form or another and live with certain bottlenecks. The ability to move function as well as data throughout different parts of the system (e.g. from hosts to disks) provides additional leverage in most efficiently taking advantage of a particular, limited, network configuration.

2.3 Storage Interfaces

One of the major changes since the time of the database machines is the level of interface between storage devices and the rest of the system. The advent of the SCSI standard [Schmidt95, ANSI93, Shugart87] has enabled much of the progress and performance gains in storage systems over the last fifteen years.

2.3.1 Disk Interfaces

At the time of the database machines, hosts were responsible for much more of the detailed control of the disk drive than they are today. The use of higher-level, more abstract, interfaces to storage have moved much of this detailed control function (e.g. head positioning, track following, and sector layouts) to the on-drive processors discussed above, thereby offloading the host to perform more application-level processing tasks.

The standardization and popularity of SCSI has greatly helped the development of storage devices by providing a fixed interface to which device drivers could be written, while allowing devices to optimize “underneath” this interface. This is true at the individual device level, as well as in groups of devices (such as RAID arrays, for example), which have taken advantage of the standardization of this interface to simply “act like” a single device and provide the same interface to the higher-level filesystem and operating system code that are not aware that they are really dealing with a group of devices. But this interface is specified at a relatively low level, and has not been updated in many years.

One of the contentions of previous work on Network-Attached Secure Disks (NASD) is that the simple block-level interface of SCSI should be replaced by a richer *object interface* that allows the drives to manage more of their own metadata [Gibson97a]. This allows both offloading of the host and provides additional opportunities for optimization at the devices. In this system, a particular filesystem built on top of network-attached disks decides how to map user-level “objects”, such as files, directories, or database tables onto the “objects” provided by the object interface. A particular filesystem may choose to map multiple files or directories to the same object, or it might split a particular file over several objects, depending on particular semantics or performance characteristics, but the most straightforward approach maps a single file to a single *object* at the device. The drive is then responsible for managing the block layout of this object itself. This for the first time gives the drive knowledge of which blocks make up a user-understood unit of access. The drive can now understand and act on information such as “I am going to read this entire file” because it has a notion of what underlying blocks this refers to, whereas before only the filesystem at the host was aware of how variable-length files mapped onto fixed-size SCSI blocks. This also improves the efficiency of the higher-level filesystem, because it must no longer keep track of the usage and allocation of individual disk blocks. It can now reason in terms of objects and depend on the drive to “do the right thing” within this context. In a network-attached environment with shared storage, this means there is much less metadata update traffic that must take place. The mapping of objects to disk blocks is known only to the disks and must not be shared among multiple hosts that may be accessing or updating the same filesystem. Original SCSI disks had thousands of blocks, which were managed directly by the filesystems on the host, today’s drives have millions and tens of millions of blocks, so offloading the management of the lowest level of allocation to the drives is a reasonable step to take.

The object interface also provides a natural way to handle the security required by network-attached disks. It is no longer reasonable for a device to execute any command that is sent to it, as is true with SCSI devices today. The drive must have a way to authenticate that a particular request comes from a trusted party, that the requestor has the right to perform a particular action. Within NASD, this is done through a *capability system* [Gobioff97]. The *file manager* responsible for a particular device or set of devices shares a secret key with each of these devices. The file manager then hands out *capabilities*, which are cryptographically protected from tampering and which identify that a particular permission, or access right, could only have come from this file manager. Clients use these capabilities to access drives directly, without having to resort to the file manager on each access, as would be required with today’s file servers. When combined with the object interface, this means that security is handled on a per-object basis. The file manager does not provide capabilities on a per-block basis, but provides a read or write capability for an entire object, which the clients can then use to read and write at their leisure.

The use of an object interface at the drives aids the development of Active Disks because application-level code operating at the drive can deal with an object as a whole. The on-drive code does not have to resort to a file server for metadata mapping or to provide per-block capabilities. The on-drive code can obtain capabilities just as client code does and these can be shipped as part of the code to the drive. The on-drive code then acts as any other users of the drive and provides the appropriate capability whenever it wishes to access the object. This means that the security system that already protects unauthorized clients from destroying other users' data also operates here. If a particular piece of on-drive code does not have the appropriate capability, then it cannot read or write the object. If it does have the appropriate capability, then it could just as well read or write the object remotely, so no additional security "hole" is opened up.

2.3.2 Storage Optimizations

Processing power inside drives and storage subsystems has already been used to optimize functions behind standardized interfaces such as SCSI. This includes optimizations for storage parallelism, bandwidth and access time, including RAID, TickerTAIP, Iceberg [Patterson88,Drapeau94, Wilkes95, Cao94, StorageTek94] and for distributed file system scalability, including Petal, Derived Virtual Devices, and Network-Attached Secure Disks [Lee96, VanMeter96, Gibson97]. With Active Disks, excess computation power in storage devices is available directly for application-specific function in addition to supporting these existing storage-specific optimizations. Instead of etching database functions into silicon as envisioned 15 years ago, Active Disks are programmed in software and use general purpose microprocessors. This makes possible a much wider range of optimizations as more vendors and users are able to take advantage of on-drive processing. The types of optimizations performed in these systems - informed prefetching, transparent compression, various levels of object storage - can be built on top of the simple infrastructure of Active Disks. The real benefit comes in being able to open up this capability to the much greater number of specific applications (e.g. database systems, mail servers, streaming video servers) that do not alone form a large enough "pull" to change on-drive processing for their own applications individually.

2.4 Workloads

There is considerable variety among the applications that place large demands on storage systems. If we consider the uses to which large storage systems are put, we see a wide range of requirements, with the only common thread a continual increase in demand, leading to the deployment of larger and larger systems as fast as the technology can evolve.

Site	System	Storage	Size	Software	Type of Data
Motley Fool	AlphaServer 1000	2 x StorageWorks 310	2 x 30 GB	SQL Server	message boards, financial data
Atrieva.com		StorageTek	12 TB, 20 GB/week	custom	free Internet storage
Aramark Uniforms	AlphaServer 4100	ESA 10000	1 TB	Oracle	sales & cust info, mining
Northrop Grumman	2 x AlphaServer 8400	StorageWorks	2 TB	SAP, Oracle	100% mirrored
Lycos	n x AlphaServer 8400	StorageWorks	5 TB	custom	web site, catalog
Mirage Resorts	Tandem, NT, AS400, UNIX	StorageTek Powder-Horn	450 GB/night		backups
CERN	various	AIX, Sun storage arrays	1 TB	AFS	15 servers, 3-5,000 active users
Boeing Engineering	various	RS/6000, Sun	50 TB	DFS	3-5,000 seats
Nagano Olympics	48 SP2 web servers	2 x RS/6000 16x9 SSA	144 GB	DFS/Web	4 complete replicas
Goddard SFC	Cray T3E, 128 GB	fibre channel disks	960 GB	Unicos	650 MFLOPS, 1024 nodes
Corbis	Compaq ProLiants	StorageWorks	2 TB	NT, IIS, SQL	high-resolution images
Cathay Pacific	Sun Enterprise 10000	Sun storage arrays	1.5 TB		data warehousing

Table 2-7 Large storage customers and systems. Data from *www.storage.digital.com*, *www.stortek.com*, *www.transarc.com*, and via *www.gapcon.com*.

2.4.1 Large Storage Systems

Disk/Trend reports that the disk drive industry as a whole shipped a total of 145 million disk drives in 1998 [DiskTrend99]. With an average drive size near 5 GB, this is a total of 725 petabytes (10^{15}) of new data storage added in a single year. Of this total, 75% went into desktop personal computers, 13% into server systems, and 12% into portables. This means over 100 petabytes of new storage found its way into data centers and servers around the world.

There is a large variation in the types of workloads for such large data systems. The advantage of Active Disks is that they provide a mechanism whereby a wide variety of applications with a range of characteristics can be supported effectively and take advantage of the same underlying hardware components. The increased flexibility in placing functions allows applications to be structured in novel ways that are simply not possible in systems with “dumb” disks, where processing can only occur after data has been transferred to a host. This provides system designers a new avenue for optimization and planning. There are also benefits in functionality and optimization that may be possible in desktop drives with an Active Disk capability, but the focus of this dissertation is on the benefits of parallelism and offloading in systems with multiple disks, outside of the commodity market for low-end single disks.

Table 2-7 provides a sample of several large storage systems in use today across a range of organizations and applications. We see that it is quite easy to reach several terabytes of data with even a modest number of users. We also see a significant variety among the uses for large data systems, meaning that storage systems must support different access patterns and concerns. A number of the most popular classes of usage are discussed below, along with the general trends in the demands they place on storage systems.

Year	System	Processor	Memory	Storage	Cost	tpmC	$/tpmC
1993	IBM RS/6000 POWERserver 230 c/s	45 MHz RISC	64 MB	10.6 GB	$245,273	115.83	2,118.00
1993	HP3000 Series 957RX	48 MHz PA-RISC	384 MB	32.7 GB	$487,710	253.70	1,923.00
Enterprise Systems							
1995	HP 9000 K410	4 x 120 MHz PA	2 GB	341.0 GB	$1,384,763	3809.46	364.00
1997	IBM RS/6000 Enterprise Server J50 c/s	8 x 200 PPC	3 GB	591.5 GB	$895,035	9,165.13	97.66
1997	HP 9000 V2200 Enterprise Server	16 x 200 MHz PA	16 GB	2,439.0 GB	$3,717,105	39,469.47	94.18
1999	IBM RS/6000 Enterprise Server H70 c/s	4 x 340 MHz RS	8 GB	1,884.4 GB	$1,343,526	17,133.73	78.50
1999	HP 9000 N4000 Enterprise Server	8 x 440 MHz PA	16 GB	3,787.0 GB	$2,794,055	49,308.00	56.67
Commodity Systems							
1995	IBM RS/6000 Workgroup Server E20 c/s	100 MHz PPC	512 MB	56.3 GB	$278,029	735.27	378.00
1997	IBM RS/6000 Workgroup Server F50 c/s	4 x 166 MHz PPC	2.5 GB	495.8 GB	$725,823	7,308.10	99.32
1997	HP NetServer LX Pro	2 x 200 MHz Pent	2 GB	512.4 GB	$584,286	7,351.50	79.48
1997	Dell PowerEdge 6100	4 x 200 MHz PPro	2 GB	451.0 GB	$327,234	7,693.03	42.53
1999	IBM Netfinity 7000 M10 c/s	4 x 450 MHz Xeon	4 GB	1,992.9 GB	$577,117	22,459.80	25.70
1999	HP NetServer LH 4r	4 x 450 MHz PII	4 GB	1,310.0 GB	$440,047	19,050.17	23.10
1999	Dell PowerEdge 6350	4 x 500 MHz Xeon	4 GB	1,703.0 GB	$404,386	23,460.57	17.24

Table 2-8 Comparison of large transaction processing systems over several years. The table compares the size, performance, and cost of large transaction processing systems - as given by TPC-C benchmark results - over the six year period. Data from [TPC93], [TPC97], and [TPC99]. No attempt has been made to adjust the dollar figures for inflation, such an adjustment would only raise the costs of the older systems and make the improvements more striking.

2.4.2 Database

Traditionally, the most important use for large data systems is to store the transaction databases that form the basis of the electronic world - whether in stock markets, banks, or grocery stores. As more and more of the world becomes computerized, more and more of our daily actions (and transactions) are stored and tracked.

The increasing size and performance of large transaction processing systems is illustrated in Table 2-8 which shows the evolution of systems over the six years since the introduction of the TPC-C benchmark. Three manufacturers and two product lines are shown. For IBM and Hewlett-Packard, there is data for “enterprise” class systems and for commodity or “workgroup” class systems. In the later years, data for commodity class systems from Dell is added. We see that over the six years, there is a huge increase in performance and a huge drop in price. From over $2,000 per tpmC to $17 per tpmC for a system that performs 200 times as many transactions. Figure 2-7 graphically illustrates the cost trend, with the commodity machines dropping off more steeply than the high-end systems. We also see that the amount of storage in these systems has increased significantly. In a TPC-C benchmark, the amount of storage required is proportional to the transaction rate, and we see that this has increased 100-fold since the first TPC-C benchmark machines.

A basic requirement embodied in the TPC-C benchmark is that the benchmark systems provide sufficient storage to maintain roughly four months of active data. If the system were going to retain historical data beyond this time, for example to support longer-term trend analysis or decision support queries, the storage requirements would quickly grow. For example, the table shows that a 50,000 tpmC system is able to fill 3 TB

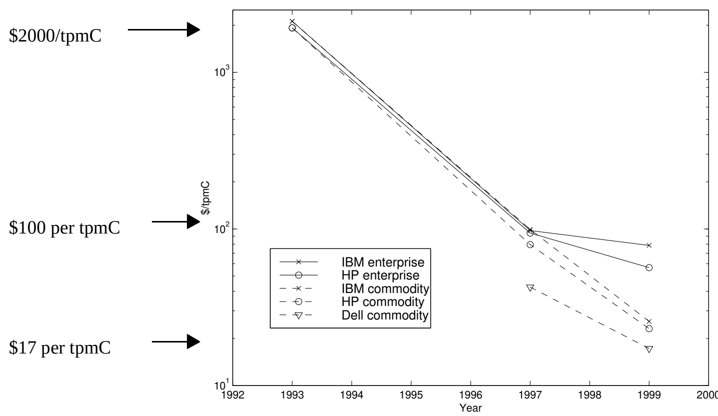


Figure 2-7 Trends in transaction processing performance and cost. The chart shows price/performance ratios for TPC-C machines from the introduction of the TPC-C benchmark in 1993 to 1999. There are two sets of plots, one for enterprise class and one for commodity or workgroup class machines across three different manufacturers. Note the log scale.

of storage in only four months. If a customer were to continue to collect that data and store it for further analysis, this system would grow at a rate of over 10 TB per year. If we look to process data of this size in a decision support system, we find that 3 TB is already the largest scale factor available for a TPC-D decision support benchmark [TPC98]. This means that the fastest transaction systems of today are rapidly swamping the largest decision support systems. Soon it may be necessary to add 30 and 300 TB scale factors to the TPC-D benchmark as such database sizes become commonplace.

2.4.3 Data Mining

In data mining, basic tasks such as association discovery, classification, regression, clustering, and segmentation are all data-intensive. The data sets being processed are often quite large and it is not known *a priori* where in a particular data set the "nuggets" of knowledge may be found [Agrawal96, Fayyad98]. Point-of-sale data in retail organizations is collected over many months and years and grows continually [Agarwal95]. Telecommunication companies maintain tens of terabytes of historical call data that they wish to search for patterns and trends. Financial companies maintain decades of data for risk analysis and fraud detection [Senator95]. Airlines and hotels maintain historical data as input for various types of yield management and targeted marketing [Sun99a]. The potential list of data sources and potential uses is endless.

Many of the statistical and pattern recognition algorithms used in data mining have been developed with small data sets in mind and depend on the ability to operate on the data in memory, often requiring multiple passes over the entire data set to do their work. This means that users must either limit themselves to using only a subset of their data, or must have more efficient ways of operating on them out-of-core.

The primary difference between "traditional" database operations and data mining or data warehousing is that on-line transaction processing (OLTP) systems were designed to automate simple and highly structured tasks that are repeated all day long - e.g. credit card sales or ATM credits and debits. In this case the reliability of the system and consistency of the data are the primary performance factors [Chaudhuri97]. The historical data sets that form the basis for data mining are the accumulated transactions of (often) several OLTP systems that an organization collects from different portions of its daily business. The goal of data mining is to combine these datasets and look for patterns and trends both within and among the different original databases (imagine, for example, combining grocery store receipts with customer demographics and weather reports to determine that few people in Minneapolis buy charcoal briquets in December). As a result, data mining systems will often contain orders of magnitude more data than transaction processing systems. In addition, the queries generated in a data mining system are much more *ad-hoc* than those in an OLTP system. There are only so many ways to debit a bank account when a withdrawal is made, while there are an infinite number of ways to summarize a large collection of these transactions based on branch location, age of the customer, time of day, and so on. This means that the use of static indices is significantly less effective than in OLTP workloads, particularly as the number and variety of attributes and dimensions in the data increases (due to the *curse of dimensionality*, discussed in more detail in the next section). The use of materialized views and pre-computed data cubes [Gray95, Harinarayan96] allows portions of the solution space to be pre-computed in order to answer frequently-asked queries without requiring complete scans, although these mechanisms will only benefit the set of queries and aggregates chosen *a priori* and must still be computed (using scans) in the first place.

These characteristics mean that data mining tasks are not well-supported by the database systems that have been optimized for OLTP workloads over many years [Fayyad98]. There are several efforts underway to identify a basic set of data mining primitives that might be added as extensions to SQL and used as the basis of these more complex queries. The field is relatively new, so many of the basic tasks are still being identified and debated within the community. One of the major factor in determining what these primitives should be is how they can be mapped efficiently to the underlying system architectures. In particular, what sorts of operations will be quick and which more cumbersome. There is as yet no "standard" way to do data mining, and there is great variance across disciplines and data sets. This means there is room for novel architectures that provide significant advantages over existing systems to make inroads before particular ways of doing things are set in stone (or code[1]).

2.4.4 Multimedia

In multimedia, applications such as searching by content [Flickner95, Virage98] place large demands on both storage and database systems. In a typical search, the user might provide a single "desirable" image and requests a set of "similar" images from the

---

1. To further illustrate this point, there are over 200 companies of varying sizes currently developing and providing data mining software [Fayyad99], whereas the number of companies that provide "standard" OLTP database management software can be counted on the fingers of one hand. This means there is much more scope for novel architectures or ways of developing code than there might be in the "traditional" database systems.

database. The general approach to such a search is to extract a set of *feature vectors* from every image, and then search these feature vectors for nearest neighbors in response to a query [Faloutsos96]. Both the extraction and the search are data-intensive operations.

Extracting features requires a range of image processing algorithms. The algorithms used and features extracted are also constantly changing with improvements in processing, or as the understanding of how users classify "similarity" in multimedia content such as images, video, or audio changes. The state of the art is constantly evolving, so workloads will require repeated scans of the entire data sets to re-extract new features. Since the extraction of features represents a lossy "compression" of the data in the original images, it is often necessary to resort to the original images for re-processing. This is true in data sets of static images that may be available for searching on the web [Flickner95], as well as in image databases used to find patterns in the physical world [Szalay99].

Once a fixed set of features has been identified and extracted from an image database, it is no longer necessary to resort to the original images, which may be measured in terabytes or more, for most queries, but the extracted data is still large. The Sloan Digital Sky Survey, for instance, will eventually contain records for several hundred million celestial objects [Szalay99]. The Corbis archive maintains over 2 million online images [Corbis99]. The dimensionality of these vectors will often be high (e.g. moments of inertia for shapes [Faloutsos94] in the tens, colors in histograms for color matching in the hundreds, or Fourier coefficients in the thousands). It is well-known [Yao85], but only recently highlighted in the database literature [Berchtold97], that for high dimensionalities, sequential scanning is competitive with indexing methods because of the *curse of dimensionality.* Conventional database wisdom is that indices always improve performance over scanning. This is true for low dimensionalities, or for queries on only a few attributes. However, in high dimensionality data and with nearest neighbor queries, there is a lot of "room" in the address space and the desired data points are far from each other. The two major indexing methods for this type of data, grid-based and tree-based, both suffer in high dimensionality data. Grid-based methods require exponentially many cells and tree-based methods tend to group similar points close together, resulting in groups with highly overlapping bounds. One way or another, a nearest neighbor query will have to visit a large percentage of the database, effectively reducing the problem to sequential scanning.

There is a good deal of ongoing work in this area to address indexing for this type of data, including X-trees [Berchtold96], but there are some recent theoretic results to indicate that this is actually a structural problem with these types of data and queries, rather than simply due to the fact that no one has found the right indexing scheme "yet".

In addition to requiring support for complex, data-intensive queries, the sheer size of these databases can be daunting. One hour of video requires approximately one gigabtye of storage and storing video databases such as daily news broadcasts can quickly require many terabytes of data [Wactlar96]. Increasingly, users are maintaining such databases that can be searched by content (whether as video, as text, or as audio), using many of the methods discussed above to find a particular piece of old footage or information. Medical image databases also impose similarly heavy data requirements [Arya94].

2.4.5 Scientific

Large scientific databases often include image data, which has already been mentioned, as well as time series data and other forms of sensor data that require extensive and repeated post-processing. These data sets are characterized by huge volumes of data and huge numbers of individual objects or observations. The Sloan Digital Sky Survey projects will collect 40 TB of raw data on several hundred million celestial objects to be processed into several different data products totalling over 3 TB. This data will be made available for scientific use to a large number of organizations, as well as to the public via a web-accessible database [Szalay99]. The dozens of satellites that form NASA's Earth Observing System will generate more than a terabyte of data per day when they become fully operational [NASA99].

2.4.6 File Systems

The workloads for distributed filesystems are likely to be as varied as the number of organizations that use them, but the trend toward an ever-increasing amount of stored data is constant [Locke98].

There have been a variety of published filesystem sizing and performance studies over the years, and each has used a slightly different methodology to illustrate different points. The results of a number of these studies are presented in Table 2-9. These results

Site	Year	Data	Files	Users	(MB/user)	(files/user)	Comments	Reference
Carnegie Mellon	1981	1.6 GB	36,000	200	8	180	single system	[Satya81]
Berkeley	1985			331			three servers	[Ousterhout85]
Carnegie Mellon	1986	12 GB		400	30		100 workstations	[Howard88]
Carnegie Mellon	1987	6 GB		1,000	6		400 clients, 16 servers, *server data only*	[Howard88]
Berkeley	1991			70			four servers, 40 diskless clients	[Baker91]
Western Ontario	1991		304,847	200		1,524	three servers, 45 diskless clients	[Bennett91]
HP Labs	1992	10.5 GB		20	537		single server	[Ruemmler93]
Berkeley	1992	3 GB		200	15		single server, nine dataless clients	[Ruemmler93]
AFS	1994	217 GB		4,750	47		900 clients, 70 servers, *server data only*	[Spasojevic96]
HP	1994	54 GB	2.3 million	527	105	4,363	46 machines	[Sienknecht94]
Harvard	1994	23 GB		75	314		four machines	[Smith94]
Carnegie Mellon	1996	8 GB		25	328		single server, *server data only*	[Riedel96]
Carnegie Mellon	1998	26.5 GB		40	671		single server, *server data only*	
Microsoft	1998	10.5 TB	140 million	4,418	2,492	31,689	4,800 machines	[Douceur99]

Table 2-9 Amount of storage used in an organization. The table compares the amount of data and the total number of users across several years of filesystem studies. All the studies are from university, research or commercial software development environment. These studies may not necessarily be representative in that they are usually the researchers studying themselves, but this does give a rough indication of how things have changed over the years. The figure for Users is the number of active users, this number is an estimate for both of the 1994 results as those papers only give the total number of registered users (based on the ratio of registered to active users in the 1987 AFS study). Note that several of the AFS studies contain data from the shared servers only, so the total amount of data (if users' workstations were included, as they were in some of the older studies) would be significantly higher.

were gathered from a number of different environments, but focussed on university research or software development organizations (as researchers in this area have a tendency to study themselves). The older studies report on entire systems, including servers and clients, while the more recent studies usually focus on shared server usage, ignoring local client storage. The most recent study at Microsoft, on the other hand, reports only client storage. All this means that the numbers among the various studies are not directly comparable, but the intent of the table is simply to show a trend. The storage per user numbers show a definite upwards trend. Looking at the only *very* roughly comparable data among similar systems (comparing the 1998, 1996, 1994 and 1987 AFS systems to each other, and the commercial environments of HP and Microsoft to each other) gives average growth rates of between 30% and 165% per year in megabytes per user. A recent survey on storage consolidation also identifies increased pressure to re-centralize storage, thereby increasing the both the amount of data and the amount and extent of sharing among multiple systems, putting increased pressure on distributed filesystems.

2.4.7 Storage Systems

A recent trend in storage devices is the increased availability of “value added” storage systems that provide a higher level of functionality than the disks themselves. This trend began with disk arrays that combine a set of disks into a single logical “device” and has continued to higher-level protocols as more “intelligence” moves into the storage systems themselves, a trend that bodes well for the acceptance of Active Disk technology. Table 2-10 considers a sampling of such systems, along with the premium that these vendors are able to charge above the cost of the storage itself.

System	Disks	Function	Cost	Premium	Other	Source
Seagate Cheetah 18LP LVD	18 GB	disk only	$900	-	lvd, 10,000 rpm	warehouse.com
Seagate Cheetah 18LP FC	18 GB	disk only	$942	5%	FC, 10,000 rpm	harddisk.com
Dell 200S PowerVault	8 x 18 GB	drive shelves & cabinet	$10,645	48%	lvd disks	dell.com
Dell 650F PowerVault	10 x 18 GB	dual RAID controllers	$32,005	240%	full FC, 2x 64 MB RAID	dell.com
Dell 720N PowerVault	16 x 18 GB	CIFS, NFS, Filer	$52,495	248%	ethernet, 256/8 MB cache	Dell
EMC Symmetrix 3330-18	16 x 18 GB	RAID, management	$160,000	962%	2 GB cache	EMC

Table 2-10 Value-added storage systems. A comparison of several value-added storage systems and their price premium over the cost of the raw storage. Note that the PowerVault 650 is an OEM version of a Clariion array from Data General and the PowerVault 720 is a version of the NetApp Filer from Network Appliance. All the costs shown are street prices as of September 1999.

2.5 Downloading Code

Downloading application code directly into devices has significant implications for language, safety, and resource management. Once there is an execution environment at the drive for user-provided code, it is necessary to provide mechanisms that protect the internal drive processing from the user code, as well as protecting different user “applications” from each other. This is necessary to safeguard the data being processed by the user code, as well as the state of drive operation. Resource management is necessary to ensure reliable operation and fairness among the requests at the drive.

Given the increased sophistication of drive control chips as discussed in Section 2.2.5, it may be possible to simply use the standard memory management hardware at the drive and provide protected address spaces for applications as in standard multiprogrammed systems today. For the cases where efficiency, space or cost constraints require that application code be co-located with “core” drive code, recent research in programming languages offers a range of efficient and safe remote execution facilities that ensure proper execution of code and safeguard the integrity of the drive. Some of these mechanisms also promise a degree of control over the resource usage of remote functions to aid in balancing utilization of the drive between demand requests, opportunistic optimizations such as read-ahead, and demand requests.

There are two issues for code operating at the drive: 1) how is the code specified to the drive in a manner that is portable across manufacturers and operating environments and 2) how is safety and resource utilization of the code managed. The next sections discuss potential solutions in these two areas.

2.5.1 Mobile Code

The popularity of Java (from zero to 750,000 programmers in four years [Levin99]) makes it a promising system for doing mobile code. A survey quoted in the Levin article reports that 79% of large organizations have active projects or plans to pursue Java-based applications [Levin99]. This popularity, and the wide availability of development tools and support, makes Java a compelling choice as a general execution environment. The availability of a common, community-wide interface for specifying and developing mobile code makes it possible for individual device manufacturers to leverage their investment in a single computation environment or “virtual machine” across a wide range of applications. It is no longer necessary to produce a custom device or custom firmware to support a large variety of different higher-level software layers. The device manufacturer can create a single device that is programmed in Java, and that can then be used by Microsoft and Solaris and Oracle and Informix in the same basic way. The development of systems such as Jini [Sun99] for managing and configuring devices builds on this same advantage to address a particular part of the problem, mediating the interaction among heterogeneous devices. There are a number of additional domains where a general-purpose mobile code system would be applicable [Hartman96].

2.5.2 Virtual Machines

The use of a virtual machine provides two complimentary benefits, the first is the ability to use the same program on a variety of underlying machine and processor architectures, the second is the greater degree of controlled provided in a virtual machine, when the code does not have direct access to the hardware. The downside of virtual machines is the performance impact of “virtualized” hardware. The extent of the performance difference across several types of interpreted systems was explored in a study by Romer, et al. [Romer96]. This study concluded that although their measurements showed interpreted Java running roughly 100 times slower than the corresponding C code, that there were a range of optimizations that should improve this performance, particularly if code could be compiled before execution. They also cite the ability to interface with efficient native code implementations of “core” functions as a way to achieve performance while maintaining the flexibility of the virtual machines - essentially taking advantage of the 80/20 rule (20% of the code takes 80% of the execution time).

Since the Romer study, a number of efforts have concentrated on improving the performance of Java, incorporating many of the techniques from traditional compiler optimization [Adl-Tabatabai96], and there are now commercial products that claim parity between the performance of Java and the corresponding C++ code [Mangione98].

Another advantage to Java over more traditional systems languages such as C or C++ is that the stronger typing and lack of pointers make Java code easier to analyze and reason about at the compiler level. This allows compilers to be more efficient and aids efforts in code specialization [Volanschi96, Consel98] that could also significantly benefit Active Disk code, as discussed in Section 6.4.

2.5.3 Address Spaces

The most straightforward approach to providing protection in a multi-programmed drive environment is through the use of hardware-managed address spaces, as found in conventional multi-user workstations. The current crop of drive control chips is already beginning to include this functionality. For example, the ARM7 core shown in Table 2-5 above contains a full memory management unit (MMU) and virtual memory support and is only marginally more complex than the same chip with only a simple memory system [ARM98].

The main tradeoff to this approach is the cost of performing context switches among the drive and user code, and of copying data between the two protection domains [Ousterhout91]. Since the on-drive code will be primary concerned with data-processing (i.e. primarily low cycles/byte computations) this overhead must be low enough to not negate the benefits of on-drive execution.

2.5.4 Fault Isolation

Work in safe operating system extensions, software fault isolation, and proof-carrying code [Bershad95, Small95, Wahbe93, Necula96] provides a variety of options for safely executing untrusted code. The SPIN work depends on a certifying compiler that produces only “safe” object code from the source code provided by the user. The downside is that this requires access to the original source code and depends heavily on maintenance of the compiler infrastructure. Software Fault Isolation (SFI) provides a way to “sandbox” object code and perform safety checks efficiently. Early measurements [Adl-Tabatabai96] indicate that this can be done with 10-20% runtime overhead for simple safety checks, without access to the original source code. Proof-Carrying Code (PCC) takes a different approach and moves the burden of ensuring safety to the original compiler of the code. The system requires that each piece of code be accompanied by a proof of its safety. This means that the runtime system is only responsible for verifying the proof against the provided code (which is a straightforward computation), rather than proving the safety of the code (which is a much more complex computation that must be done by the originator of the code at compilation time).

The common theme that each of these systems stress is that while safety is an important concern for arbitrary, untrusted code, the design of the "operating system" interfaces and APIs by which user code accesses the underlying system resources is the key to ensuring dependable execution [McGraw97]. This design will vary with each system within which code is executed and will require careful effort on the part of the system designers, beyond the choice of a mechanism for ensuring safety.

2.5.5 Resource Management

The primary focus of these methods has been on memory safety - preventing user code from reading or writing memory beyond its own "address space", but some of these mechanisms also promise a degree of control over the resource usage of remote functions. This is important within an Active Disk in order to balance resources (including processor time, memory, and drive bandwidth) among demand requests, opportunistic optimizations such as read-ahead, and remote functions.

The simplest approach is to use scheduling algorithms similar to those currently employed in time-sharing systems that depend on time slices and fairness metrics to allocate resources among concurrent processes, as in traditional multi-user operating systems.

There is also work in the realtime community on scheduling and ensuring resource and performance guarantees. The main difficulty with the scheduling methods in this domain is that they require detailed knowledge of the resource requirements of a particular function in order to set the frequency and periods of execution. They also usually requires that resources be allocated pessimistically in order to ensure that deadlines are met. This generally leads to excess resources going unused, a situation that may not be acceptable in the low resource environment at individual disk drives. There has been some recent work to address this problem by allowing feedback between applications and the operating system to make this tradeoff more easily [Steere99].

All of the technologies discussed allow for control over user-provided code, the main tradeoff among them is the efficient utilization of resources at the drives (in terms of safety and "operating system" overheads) against the amount of infrastructure required external to the drive and in the runtime system to support each method (compilers, proof-checkers, and so on). The availability of mobile code opens a compelling opportunity, and there are a variety of options for managing the code that implementors of an Active Disk infrastructure can choose from.

Chapter 3: Potential Benefits

This chapter introduces a model for determining the potential benefits in the performance of an application in a system using Active Disks. This analytic model compares the performance of a server system with a number of “dumb” disks against the same system with the traditional disks replaced by Active Disks. The model makes a number of simplifying assumptions to keep the analysis straightforward, but the model validation discussed in Chapter 5 will show that the performance of the prototype system closely matches the results predicted by the model.

The intent of this chapter is to outline the potential benefits of using an Active Disk system over a system with traditional disks, the following chapters will describe a set of applications and how they map to Active Disks and show the performance of these applications in a prototype system.

3.1 Basic Approach

The basic characteristics of remote functions that are appropriate for executing on Active Disks are those that:

- can leverage the parallelism available in systems with large numbers of disks,
- operate with a small amount of state, processing data as it “streams past” from the disk, and
- execute a relatively small number of instructions per byte.

The degree to which a particular application matches these three characteristics will determine its performance in an Active Disk system.

This chapter presents an analytic model for the performance of such applications in order to develop an intuition about the behavior of a system with Active Disks relative to a traditional server. To keep the model simple, it assumes that 1) applications have the three characteristics mentioned above, and 2) that disk transfer, disk computation, interconnect transfer and host computation can be pipelined and overlapped with negligible startup and post-processing costs.

We will see that the first assumption simply encompasses the application characteristics that determine the performance of a particular application on Active Disks, these will be the input parameters to the model, and are discussed in more detail below. The second assumption is addressed in the section on Amdahl's Law at the end of the chapter, which discusses the effect of relaxing the requirement of perfect overlap among pipelined phases, as well as how to include the startup overhead of Active Disk processing into the model.

Application Parameters

N_{in} = number of bytes processed

N_{out} = number of bytes produced

w = cycles per byte

t = run time for traditional system

t_{active} = run time for active disk system

System Parameters

s_{cpu} = CPU speed of the host

r_d = disk raw read rate

r_n = disk interconnect rate

Active Disk Parameters

s_{cpu}' = CPU speed of the disk

r_d' = active disk raw read rate

r_n' = active disk interconnect rate

Traditional vs. Active Disk Ratios

$\alpha_N = N_{\text{in}}/N_{\text{out}}$ $\quad\alpha_d = r_d'/r_d$ $\quad\alpha_n = r_n'/r_n$ $\quad\alpha_s = s_{\text{cpu}}'/s_{\text{cpu}}$

Starting with the traditional server, the overall run time for a simple non-interactive data-processing application is the largest of three individual pipeline stages: the time to read data from the disk, the time to transfer the data on the interconnect, and the time to process the data on the server, which gives:

$$t = max\left(\frac{N_{\text{in}}}{d \cdot r_d}, \frac{N_{\text{in}}}{r_n}, \frac{N_{\text{in}} \cdot w}{s_{\text{cpu}}}\right)$$

for the elapsed time, and:

$$\text{throughput} = \frac{N_{\text{in}}}{t} = min\left(d \cdot r_d, r_n, \frac{s_{\text{cpu}}}{w}\right)$$

where each term is parameterized by the number of bytes of data being processed, which can then be factored out to obtain a throughput equation independent of data size. For the

Active Disks system, the comparable times for disk read, interconnect transfer, and on-disk processing are:

$$t_{active} = max\left(\frac{N_{\text{in}}}{d \cdot r_d}, \frac{N_{\text{out}}}{r_n'}, \frac{N_{\text{in}} \cdot w}{d \cdot s_{\text{cpu}}'}\right)$$

for the elapsed time, and:

$$\text{throughput}_{\text{active}} = \frac{N_{\text{in}}}{t_{active}} = min\left(d \cdot r_d', r_n' \cdot \frac{N_{\text{in}}}{N_{\text{out}}}, d \cdot \frac{s_{\text{cpu}}'}{w}\right)$$

for the throughput. Both of the throughput equations are a minimum of the three possible bottleneck factors: the aggregate disk bandwidth, the storage interconnect bandwidth, and the aggregate computation bandwidth.

Rewriting the equation for throughput with Active Disks in terms of the parameters of the traditional server and the ratios between the traditional and the Active Disk parameters - the total data moved (the selectivity α_N), the disk bandwidth (α_d, which should be 1, since the use of Active Disks should not impact the raw disk bandwidth[1]), the interconnect bandwidth (α_n, which should also be 1 in the normal case, as the use of Active Disks does not change the raw networking bandwidth[2]), and the relative CPU power (α_s, which will be the key system parameter when comparing the two types of systems) - we have:

$$\text{throughput}_{\text{active}} = min\left(\alpha_d \cdot (d \cdot r_d), \alpha_N \cdot \alpha_n \cdot (r_n), d \cdot \alpha_s \cdot \left(\frac{s_{\text{cpu}}}{w}\right)\right)$$

This equation captures the basic advantages of Active Disks. Applications with high selectivity (large α_N) make more effective use of limited interconnect bandwidth, and configurations with many disks ($d \cdot \alpha_s > 1$) can achieve effective parallel processing and overcome the processing power disadvantage (small α_s) of the individual Active Disks.

1. the only reason this parameter is included is because the prototype system described in the next chapter will have an α_d larger than one. The raw disk bandwidth in the prototype Active Disk is less than that of the competing "dumb" disks, so the results given in the prototype comparison will always be pessimistic to the Active Disk case.
2. this ratio could also be greater than 1.0 if disk-to-disk communication is used, in which case the interconnect bandwidth of the Active Disk system (r_n') will be the aggregate backplane bandwidth of the network switch connecting the drives, instead of the bandwidth into the single server node.

3.1.1 Estimating System Ratios

The applications discussed in the next chapter exhibit selectivities (α_N) of 100 to 10^8 or more, providing throughput possible only with extremely high interconnect bandwidth in the traditional system[1]. In practical terms, this means that a system can obtain high application-level throughput without requiring the use of the highest bandwidth (and most expensive) interconnects, thereby keeping down the overall system cost.

This effect was discussed in the Interconnects section of the previous chapter to argue that network technology will need to continue to take advantage of locality in network traffic patterns, as full crossbar interconnects across a large number of nodes are simply too expensive. A reduction in the amount of data to be moved directly at the drive, before any bytes are even placed on the interconnect, can be highly effective in maintaining low interconnect requirements "upstream", as we will see. To take into account the use of a less expensive interconnect, the model allows for slower Active Disk interconnects in the range of $0.1 < \alpha_n < 1.0$.

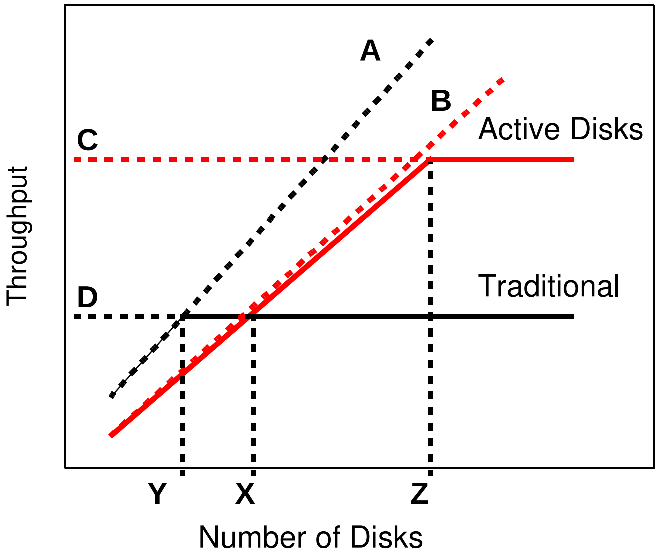
The final and critical system parameter is the ratio of Active Disk to server processor performance. The Silicon section in the previous chapter argued that we can expect processing rates of 100 and 200 MHz microprocessor cores in next generation disk drives. With individual server CPUs of 500 to 1,000 MHz processing rates in the same time frame, a ratio of about $\alpha_s = 1/5$ should be realistic. In this case, the aggregate Active Disk processing power exceeds the server processing power once there are more than five disks ($d > 5$) working in parallel. If there are multiple processors, in an SMP system for example, then the crossover point will shift, as discussed in the Processing section below.

3.1.2 Implications of the Model

Figure 3-1 illustrates the basic trade-offs for Active Disk systems. The slope of line A represents the raw disk limitation in both systems. If we assume that the Active Disk processor will not be able to keep up with the disk transfer rates for many applications ($s_{\text{cpu}}' < w \cdot r_d'$), then the aggregate throughput for these applications will have the somewhat lower slope shown by line B on the chart. Applications with a low enough cycles per byte (w) will be able to keep up with the raw bandwidth of the disk and operate at line A (i.e. the raw disk bandwidth will be the limiting factor, the application cannot process data faster than the disk can provide it), but most applications will operate at the lower line B and no application can operate faster than line A consistently.

With a sufficiently large number of disks, Active Disks saturate their interconnects at line C, with $\text{throughput}_{\text{active}} = r_n' \cdot \alpha_N \le min(d \cdot r_d', d \cdot s_{\text{cpu}}'/w)$. Since $x \ge min(x, y)$ and intercon-

1. The parameter to compare is the total amount of disk bandwidth to get bytes off the disk against the aggregate interconnect bandwidth to deliver those bytes to a host or hosts. As we will see, there are two types of limits here, the interconnect limit of getting to data into any single host, and the aggregate interconnect bandwidth across a fabric containing a number of disks and hosts. In most cases, the throughput possible with Active Disks will exceed both.



To the left of point Y, the traditional system is disk-bound. Below the crossover point X, the Active Disk system is slower than the server system due to its less powerful CPU. Above point Z, even the Active Disk system is network-bottlenecked and no further improvement is possible.

Figure 3-1 Performance model for an application in an Active Disk system. The diagram shows an abstract model of the performance of an Active Disk system compared to traditional single server system. There are several regions of interest, depending on the characteristics of the application and the underlying system.
The raw media rate of the disks in both cases is plotted as line A. The raw computation rate in the Active Disk system is line B, which varies with the cycles/byte cost of each application and the power of the Active Disk processors. Line C shows the saturation of the interconnect between the Active Disks and host, which varies with the selectivity of the application and can easily be at 1000s of MB/s of application-level throughput. Line D represents the saturation of the server CPU or interconnect in the traditional system, above which no further gain is possible as additional disks are added. This limit is often less than 100 MB/s in today's large database systems.

nect bandwidth can be assumed to be greater than a single disk's bandwidth ($r_n' > r_d'$), the number of disks must be larger than the selectivity of the application ($r_n' \cdot \alpha_N < r_n' \cdot d$) before this limit sets in. This is shown to the right of point Z in the figure. With the large selectivities of the applications discussed in the next chapter, the perfect overlap assumption would most likely fail (Amdahl's Law, as discussed below) before this point is reached.

There are two ways in which the traditional server system can be limited, either a network or CPU bottleneck, represented by line D in the figure. The point X in the figure, at which the Active Disk throughput exceeds the traditional server system is determined by $X \cdot s_{cpu}'/w = min(r_n, s_{cpu}/w)$, so $X \le s_{cpu}/s_{cpu}' = 1/\alpha_s$.

Combining all of the above analysis and defining *speedup* as Active Disk throughput over server throughput, we find that for $d < 1/\alpha_s$, the traditional server is faster and at the other points in the chart, the speedup is:

No Speedup $\qquad d < 1/\alpha_s$

$$S = \frac{d \cdot (s_{\text{cpu}}'/w)}{min(r_n, s_{\text{cpu}}/w)} \qquad 1/\alpha_s < d < \alpha_N$$

$$\geq d \cdot \alpha_s$$

$$S = \frac{(r_n' \cdot \alpha_N)}{min\left(r_n, \frac{s_{\text{cpu}}}{w}\right)} \qquad d > \alpha_N$$

$$= max\left(\alpha_N \cdot \alpha_n, \alpha_N \cdot \alpha_s \cdot \left(\frac{w \cdot r_n'}{s_{\text{cpu}}'}\right)\right)$$

$$> \alpha_N \cdot max(\alpha_n, \alpha_s)$$

which should hold for at least the first generation of Active Disks.

Considering for a moment the "slowdown" due to using Active Disks when $d < 1/\alpha_s$ (the area to the left of point X in the figure), we see that this condition is independent of the application parameters, so a query optimizer or runtime system can determine *a priori* when to prefer traditional execution of the scan for a particular system configuration, rather than executing at the drives. This parameter will be determined at the time the system is built, and would only vary across different applications if the declustering of different objects in the system were allowed to vary.

The charts in Figure 3-2 show concrete numbers for a number of real systems and a particular set of application parameters. The chart shows the performance of two of the large database systems from the previous chapter. The Compaq ProLiant system is a TPC-C benchmark system designed for a transaction processing workload [TPC98b]. The top charts show the performance of this system on a Data Mining application representative of the ones discussed in the next chapter. The first chart compares the predicted performance of the server system and a hypothetical system that could take advantage of the processing power already available on today's disk drives. The chart assumes a disk processor of 25 MHz ($s_{\text{cpu}}' = 25$), a raw disk rate of 15 MB/s ($r_d = 1/5$), and a computation requirement of 10 cycles/byte ($w = 1/5$) for the Data Mining application. The server contains four 400 MHz processors ($s_{\text{cpu}} = 1600$) and a total of 141 disk drives ($d = 141$). We see

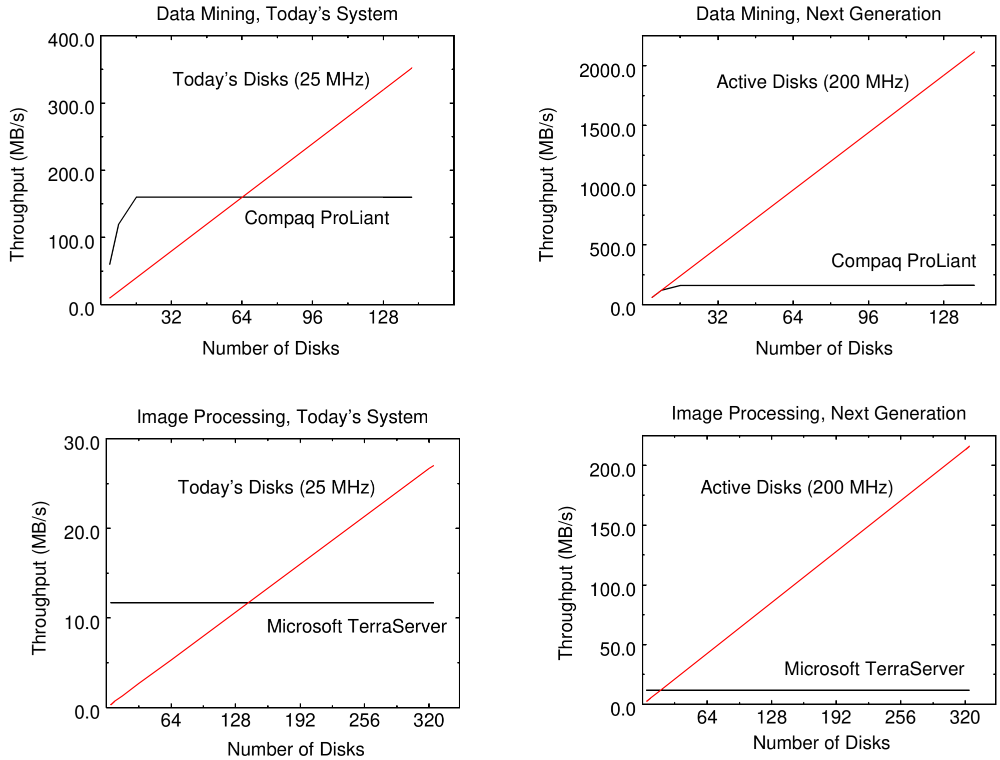


Figure 3-2 Predicted performance of several real systems. The charts show the performance predicted by the model for two of the large database systems introduced at the beginning of the previous chapter. The application shown for Data Mining is an average-cost function from those described in the next chapter and has a computation requirement of 10 cycles/byte. The Image Processing application is similar to those discussed in the next chapter and has a computation requirement of 300 cycles/byte. Note that with a cycles/byte of 10 on a 200 MHz disk processor and a 15 MB/s disk, the next generation Data Mining system is disk-bound, while the Image Processing at 300 cycles/byte is CPU-bound in both cases, as is the ProLiant system in both cases.

that with less than 64 disks, the server system is faster than the system that allows on-disk processing, but that when all 141 disks are used, performance when using the on-disk processing is about 2.5 times that of running the computation at the host. The chart on the right extends this comparison to a next generation system with Active Disks of the power suggested in the previous chapter ($s_{cpu} = 200$). In this case, as soon as there are ten disks in the system, the combined processing power of the disks exceeds the host. When using all 141 disks, the Active Disk system exceeds the performance of the traditional system by close to a factor of 15.

The lower charts show the same comparison using the details of the Microsoft TerraServer system described in the previous chapter. It contains eight 440 MHz processors ($s_{cpu} = 3520$) and a total of 324 disk drives ($d = 324$). Using the hypothetical system with today's disk drives, we see that with less than about 140 disks, the server system is again faster than the system that allows on-disk processing. When all 324 disks are used, the system with on-disk processing is again more than twice as fast as the server system. The chart on the right again extends this comparison to a next generation system and shows that as soon as there are about 20 disks in the system, the combined power of the Active Disks exceeds the host. At 324 disks, the Active Disk system is almost 20 times faster than the traditional system. This clearly demonstrates the potential of Active Disks in large systems, even with relatively low-powered Active Disks.

3.1.3 Trends

Taking a look at Figure 3-1 and considering the prevailing technology trends, we know that the processor performance (line B) improves by 60% per year and disk bandwidth (line A) by 20% to 40% per year [Grochowski96]. This will cause the ratio of processing power to disk bandwidth in both systems to increase by 15% per year. This will continue to narrow the gap between line A and B and bring the performance of Active Disks closer to the maximum possible storage bandwidth (the raw disk limitation) as the processing power available on the disks catches up to the raw disk bandwidth. This means that more and more applications, with higher cycles/byte, can be supported effectively.

3.1.4 Application Properties

The basic characteristics of an application that determine its performance in an Active Disk system is the cycles per byte cost of its basic computation and the selectivity of its processing. A secondary parameter is the memory requirements of the computation, although we will see that this can often be folded into a change in the selectivity, with proper partitioning and choice of the algorithms. Table 3-1 shows the values for several of

Application	Input	Computation (instr/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Bandwidth (KB/s)
Select	m=1%	7	28.6	-	100	290
Search	k=10	7	28.6	72	80,500	0.4
Frequent Sets	s=0.25%	16	12.5	620	15,000	0.8
Edge Detection	t=75	303	0.67	1776	110	6.1
Image Registration	-	4740	0.04	672	180	0.2

Table 3-1 Costs of the applications presented in the text. Computation time per byte of data, memory required at each Active Disk, and the selectivity factor in the network. The parameter values are variable inputs to each of the applications.

the applications discussed in the next chapter. The Throughput column shows the maximum possible throughput of the application based on the cycles per byte in the Computation column and assuming a 133 MHz Active Disk processor. In the cases where this throughput is higher than the raw bandwidth of the disk, the performance will be disk-limited. If this throughput is lower than the disk bandwidth, then the performance is compute-limited at the drives, although it may still be network limited at the server. The Bandwidth column shows the bandwidth required per Active Disk assuming a 10 MB/s raw disk bandwidth, the computation rate shown in the Throughput column and the selectivity factor given in the Selectivity column. As we will see in Chapter 5, an Active Disk system with sufficient parallelism can outperform a server system even if the drive processors cannot keep up with the raw disk bandwidth. This is not the most efficient regime for the disks, but still provides better overall performance than the server system with its more limited aggregate computational resources.

We see that the data mining operations have very low cycle per byte costs and high selectivities. The multimedia applications are significantly more costly in cycles, lowering the aggregate throughput possible with these applications. The details of these costs and how they might vary across applications, or across data sets, are discussed further in the next chapter. This table only outlines the basic parameters to provide a level set of the values expected in practice.

3.1.5 System Properties

The basic properties of the system include the raw disk data rate of the Active Disks, the processing power of the individual Active Disks, the processing power of the host to which they are attached, and the network speeds between the disks and the host. Table 3-2 shows the system parameters used in the model and gives realistic values for what is available in current generation systems, the details of what the prototype system in the following chapters has, and what is expected for next generation systems.

Parameter	Symbol	Today	Prototype	Next Generation
host processor	s_{cpu}	500 MHz	500 MHz	750 MHz
disk processor	s_{cpu}'	25 MHz	133 MHz	200 MHz
disk rate	r_d	10.0 MB/s	11.0 MB/s	20.0 MB/s
network rate	r_n	40.0 MB/s	45.0 MB/s	200.0 MB/s
active disk rate	r_d'	-	7.5 MB/s	20.0 MB/s
active net rate	r_d'	-	12.0 MB/s	100.0 MB/s

Table 3-2 System parameters in today's, tomorrow's, and the prototype system. The chart shows the setting of the system parameters in the prototype, as well as for a typical system in use today and a prediction for the first generation of Active Disk sytems. Today's system assumes a SCSI disk drive of average performance and two Ultra Wide SCSI adapters at the host (20.0 MB/s each). The next generation system assumes a first generation Active Disk with 200 MHz of processing and a Fibre Channel storage interconnect. The host again has two Fibre Channel adapters (100.0 MB/s each). Values for today's and the next generation system are rated maximums, while the values for the prototype are measured achievable maximums.

3.2 Bottlenecks

Depending on the characteristics of the application, there are three areas where a system might become bottlenecked. The throughput of the disks, the network, or the processing elements may all be the limiting factor in overall performance. Active Disks address each of these areas.

3.2.1 Disk Bandwidth

In an application with a low cycle per byte cost, the overall throughput is indeed limited by the throughput of the underlying disks. This is true in both the Active Disk and server case, as shown in Figure 3-1. In a sense, this is the best possible situation for a storage system. If the raw disk bandwidth (i.e. physical performance of the disk assembly and density of the media) is the limiting performance factor, this means the disk is essentially operating at maximum efficiency. In the limit, it is simply not possible to process data faster than the disk media can provide it.

In this case, the benefit of Active Disks is in providing the possibility of better scheduling of requests before they go to the disk internals, to make more efficient use of the underlying bandwidth. With additional higher-level knowledge [Patterson95, Mowry96], overall throughput of the disk can be increased. The work of Worthington and Ganger [Worthington94] shows that with more sophisticated scheduling, the performance for random requests can be increased by up to 20%. These types of benefits will be less dramatic for large, sequential requests, which already use the disk very efficiently. There may be a benefit with the use of extended interfaces that allow more flexible reordering of requests. One possible way to take advantage of this is by allowing a “background” workload that can take advantage of idleness in a “foreground” workload to opportunistically improve its performance, as illustrated in Section 5.4.

The far bigger effect on disk throughput comes from the addition of extra disks across which data is partitioned. If the data rates of particular applications are known, then disks can be added to provide the appropriate level of performance [Golding95]. Active Disks will not help this directly, but will allow more efficient use of the disk resources that are available. Active Disks can also aid in the collection of statistics and performance metrics of individual devices and workloads. This information can then be used by a higher-level management system to optimize the layout and placement of the workload [Borowsky96, Borowsky98]. This makes possible systems with a much greater amount of self-tuning and self-management that typical storage systems today. In order to scalably perform such monitoring and control, it is necessary to have control and computation at the end devices, rather than attempting to monitor everything centrally.

3.2.2 Processing

An application that is limited by the CPU processing rate, such as the multimedia applications discussed in the next chapter, benefits from the inherent parallelism in the Active Disk system. Where the server is limited by it’s single CPU, the processing power of the Active Disks scales with the number of disks available.

Of course, it is also possible to add additional processing capability to a server system, for example by using an SMP architecture rather than a single-processor machine. This is not precluded by an Active Disk system, nor does it change the basic model. This simply replaces the processing rate (s_{cpu}) with a higher value. For example, Figure 3-3

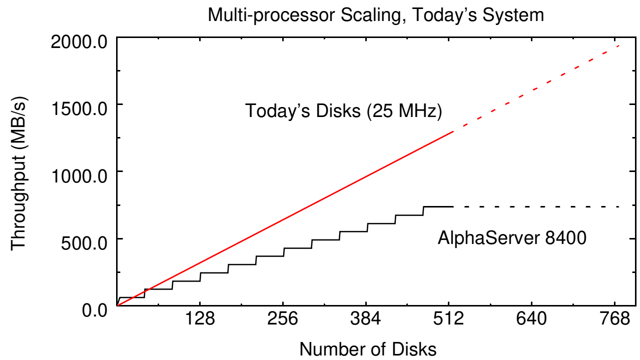


Figure 3-3 Performance of an SMP system. The chart shows the performance of a multi-processor system as processors and disks are added. We see a coarse step function as additional processors are added, but a much smoother increase with additional Active Disks. This is the model prediction for the AlphaServer 8400 system introduced in the previous chapter. The application parameters assume an average-cost Data Mining application with a computation requirement of 10 cycles/byte. The disks have 25 MHz processors and the host has up to twelve 612 MHz processors. Note that the Active Disk system would continue to scale as additional disks are added, while the SMP system cannot support more than 12 processors.

shows the expected performance if the number of processors in an SMP is scaled as additional disks are added. The chart considers the details of the AlphaServer 8400 system from the previous chapter, and assumes that the number of disks is balanced with the number of processors in that system, i.e. both increase linearly. One processor is added for approximately every 45 disks. This shows a step function in the host processing power, with a large boost whenever another processor is added. This still does not scale nearly as far as the Active Disk system at the high end, because there are still many more disks and processors. The chart assumes the comparison in today's system, using the 25 MHz value for on-disk processing power. The benefit would be even greater with the 200 MHz on-disk processors. The line for Active Disks is much smoother because processing power is added in much smaller increments - for each 4 GB of additional storage, another 25 MHz of processing power is added.

Also note the much bigger performance gap at the very high end. It is simply not possible to add processors beyond twelve in this AlphaServer system, which is still one of the largest SMP systems available. This is true in most SMP systems sold today. This limitation comes primarily from physical limitations of building a system of that size, including the basic speed of the memory bus connecting all the processors and the single shared memory. The Active Disk system, on the other hand, will continue to scale as additional disks are added (to over 10,000 disks for this particular application, at which point the system is network bottlenecked).

3.2.3 Interconnect

Applications that are network-limited benefit from the filtering of Active Disks (leaving data on the disks, if it doesn't have to move) and the scheduling that can be done with intelligence at the edges of the network. In a traditional disk system, a set of bytes must be moved from the disk, across the interconnect, and through the host memory system before the CPU can operate on it and make a decision (e.g. “take or leave”, “where to route”). In an Active Disk, these decisions can be offloaded to the devices and bytes will never have to leave the device if they will not be used in further processing.

Additional pressure is placed on storage interconnects by the introduction of Storage Area Networks (SANs) and the increased sharing demanded by today’s applications and customers [Locke98].

The data in Table 3-3 shows the interconnect limitations in a number of today’s

System	Disks	SCSI	PCI	Actual (Q6)
AlphaServer 8400	5,210 MB/s	120x20=2,400 MB/s	266x3=798 MB/s	446 MB/s
AlphaServer GS140	5,640 MB/s	96x40=3,840 MB/s	12x266=3,192 MB/s	684 MB/s
Sun Enterprise 4500	990 MB/s	6x100=600 MB/s	2x1000=2,000 MB/s	180 MB/s

Table 3-3 Interconnect limitations in today’s database systems. The table shows the theoretical and achieved bandwidth of a number of large database systems executing the TPC-D decision support benchmark. Query 6 is a simple aggregation, so the primary cost should be the reading of the data from disk. The query is a scan based on the shipdate attribute in the table. All of the system use a layout that range-partitions the table based on shipdate. This optimizes performance for this particular scan on shipdate, but would be useless for another query that used orderdata, for example. It does provide a surrogate for determining the raw disk performance possible with the system.

TPC-D systems. The table lists the theoretical and the actual throughput of delivering data to the processors in these large SMP systems. The throughput values are obtained by using the time to complete Query 6 in the benchmark, which must sequentially scan 1/7 of the lineitem table, and uses very little cycles/byte (so it should be interconnect limited in all cases).

3.3 Amdahl’s Law

The model presented above assumes that computation, network transfer, and disk access can be completely overlapped. This ignores Amdahl’s Law, which can be expressed using the parameters of the model as:

$$\text{speedup} = \frac{(1-\boldsymbol{p}) \cdot \frac{s_{\text{cpu}}}{w} + \boldsymbol{p} \cdot \frac{d \cdot s_{\text{cpu}}'}{w}}{\frac{s_{\text{cpu}}}{w}}$$

where p is the parallel fraction of the computation, the portion that can be performed in parallel at the Active Disks. This equation also assumes that CPU processing is the bottleneck, although a similar calculation would apply for an interconnect bottleneck as well.

We see that even if there is no parallel fraction ($p = 0$), the system with Active Disks is never slower than the system without. On the other hand, for applications such as the example Data Mining application shown for the AlphaServer 8400 system in the previous section, the parallel fraction is close to 100% ($p = 0.98$), meaning a speedup of 1.75, close to twice as fast even with the low processing power of today's disks. With next generation Active Disks at 200 MHz, the ratio would be 13.9. Even if the parallel fraction were only 50% ($p = 0.50$), the speedup would still be 7.6.

We see that the non-parallel fraction of a computation will definitely affect an application's performance in an Active Disk system, but if we view the Active Disks as simply an "accelerator" on the host system, overall system performance will never be worse with Active Disks than without, while in many cases it will be many times better.

3.3.1 Startup Overhead

The serial fraction of the computation ($1 - p$) can be an inherent property of the computation, but it may be due simply to the overhead of starting up the parallel computation at the disks. The Validation section in Chapter 5 discusses the startup overheads seen in the prototype system and their impact on performance. In general, this will be the time to send the necessary code to the drives, initialize the execution environment on each disk, and begin execution on a particular data object or set of objects. In applications that operate on very small data sets spread across a large number of disks, this could become a significant fraction of the overall execution time. However, given the applications and data sizes discussed in the previous chapter and the prototype applications illustrated in the next chapter, this overhead should easily be overcome by the amount of data being processed, resulting in a very low serial fraction and good speedups. In addition, many of the factors that contribute to the startup overhead will be static properties of the application or the Active Disk system, meaning that a query optimizer or runtime system could take this overhead into account and not initiate an Active Disk computation if it would be overhead-dominated. It could them proceed simply with the host processor and not take advantage of the extra power available at the drives.

3.3.2 Phases of Computation

The final property of an application that will work against the fully overlapped assumption of the model is synchronization between different phases of a computation. For example, the frequent sets application discussed in the next chapter proceeds in several stages and requires synchronization among all the disks and the host at the end of each stage, as illustrated in Figure 3-4. In this computation, the host sets the initial parameters for the computation and starts parallel execution at the disks. The disks then perform their computation locally and determine the results for their own data. These results are passed to the host and combined for the start of the second phase. This process is repeated through several more phases, until the host determines that the results obtained are complete and computation ends.

This type of synchronization among processors operating in parallel is the bane of all parallel programmers and system designers. There are several reasons to believe that this effect will be less severe in the case of Active Disk computations than in general par-

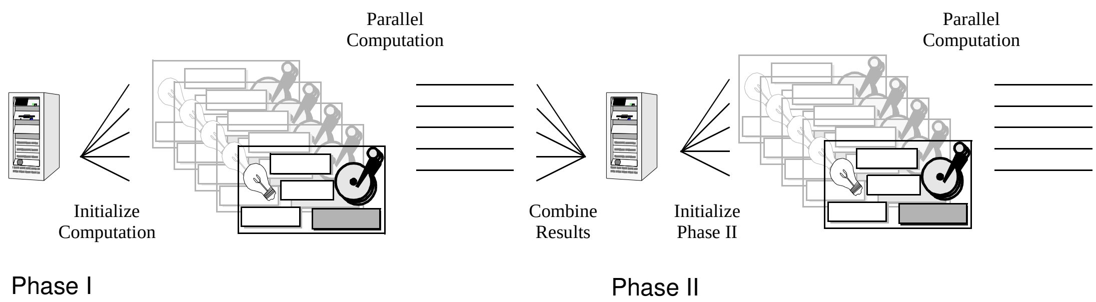


Figure 3-4 Synchronization in a multiple phase computation. The diagram shows several stages of the frequent sets application introduced in the next chapter. The host initiates the computation at all the disks, the disks proceed in parallel computing their local results. The results are then gathered at the host which combines the individual disk results and prepares the parameters for the second phase. This continues through several phases, requiring synchronization among all the drives and the host at the end of each.

allel programs. For one, the types of computations performed at the Active Disks will usually be *data parallel*, since the basic point of executing function at the disks is to move function and processing power where the data is - and distribute it in the same way that data is distributed. In addition, the disks will be largely homogeneous, eliminating some of the imbalances seen in general parallel systems.

In a sense, one of the degrees of freedom available in a general parallel programming system - the ability to move data to the place where there are available computing resources - is removed with Active Disks. The most successful Active Disk applications will operate on the data at the disk where it already resides. By computing on the data before it is placed on the network, Active Disks eliminates one of the phases of parallel computation that proceeds in three steps:

1) read data into the memories of the processing elements (whether into distributed memories or into a single, shared memory)
2) rearrange the data to the most appropriate node for processing, and
3) perform the processing

With perhaps a fourth phase:

4) rebalance the data (and work) among the processors

This process is simplified for Active Disks because the basic tenet is to compute on the data where it is stored, and then send it onto the network. The most effective Active Disks applications will perform the largest portion of their processing on the disks, before data is ever put onto the network. This does not mean that it is not possible to move data among computation elements, but it does lead to a different cost/benefit tradeoff for doing such a move, when compared to a traditional parallel processing system. More details on this, and a further discussion of the differences between Active Disks and general parallel programming are provided in Chapter 7.

3.4 Modified Model

Combining the discussion of the previous sections with the original performance model, gives a modified model that takes into account both Amdahl's Law and multiple-phase computations.

From before, we have:

$$t_{active} = max\left(\frac{N_{\text{in}}}{d \cdot r_d'}, \frac{N_{\text{out}}}{r_n'}, \frac{N_{\text{in}} \cdot w}{d \cdot s_{\text{cpu}}'}\right) \qquad (1)$$

which applies for the parallel portion of the computation, with the assumption of a 100% parallel fraction ($p = 1.0$). If we now add a serial fraction as:

$$t_i = (1 - \boldsymbol{p}) \cdot t_{serial} + t_{active} \qquad (2)$$

then we have an equation that holds for an arbitrary computation step and takes into account any startup overhead and any computation that cannot be performed in parallel as the serial fraction ($(1 - \boldsymbol{p})$).

If we then also take into account multiple phases of computation, each of which is subject to the same equation, we have:

$$t_{overall} = \sum_i t_i \qquad (3)$$

as the time for the entire application, which gives the throughput of the Active Disk system as:

$$\text{throughput}_{\text{active}} = \frac{N_{\text{in}}}{t_{overall}} \qquad (4)$$

for the total amount of data processed.

For an application with only a single phase, this simplifies to:

$$\text{throughput}_{\text{active}} = \frac{N_{\text{in}}}{(1-\boldsymbol{p})\cdot t_{serial} + max\left(\frac{N_{\text{in}}}{d\cdot r_d}, \frac{N_{\text{out}}}{r_n'}, \frac{N_{\text{in}}\cdot w}{d\cdot s_{\text{cpu}}'}\right)} \qquad (5)$$

where we still see the basic benefit of Active Disks. If the serial fraction is sufficiently small, then the terms on the right will scale linearly with the number of disks until it becomes network bottlenecked (the N_{out}/r_n' term) at some point.

As a modification to the original model, this means that the Active Disk line in Figure 3-1 will have a lower slope than the linear scaling shown there, with the amount of reduction proportional to the size of the serial fraction.

Chapter 4: Applications and Algorithms

This chapter describes a number of data-intensive database, data mining, and multimedia applications and details the changes required for them to efficiently leverage an Active Disk system. It discusses the partitioning of function between the host and the Active Disks for standalone applications from data mining and multimedia, and for all the core functions of a relational database systems. The chapter introduces the structure of these applications, and the following chapter presents the measured performance of these applications in a traditional system, and in an Active Disk system.

4.1 Scans

The most compelling applications for Active Disks are the "embarrassingly parallel" scan operations that can be easily split among a set of drives and that perform highly selective processing before the data is placed on the interconnection network.

The basic processing in an Active Disk system compared to a traditional host-based system is illustrated in Figure 4-1. Instead of running all application processing at the host, and forcing all the raw data to move from the disk drives, through the storage interconnect, and to the host before processing, Active Disk applications execute on both the host and the disks. The "core" portion of the applications' data-intensive processing is extracted and executed in parallel across all the disks in the system. The diagram in Figure 4-1 shows the Active Disks as an extension of NASD (Network-Attached Secure Disks), which means they are network-attached, present an object- instead of a block-level interface, and contain provisions for a robust security system enforced directly at the drives. These three characteristics are not strictly necessary for Active Disks, one could imagine providing execution functionality in more traditional SCSI disks, but there are several advantages, discussed in Section 2.3.1 and Section 6.2.1 to building on the NASD model. The addition required for Active Disks, then, is the presence of an execution environment that can run portions of the applications' code - these are the "bright ideas" (application-level execution) in the diagram.

The second basic type of Active Disk application applies a filter to data as it moves from the disk to the host. This both reduces the amount of data on the interconnect, and offloads the host processor. The simplest example of this is a database select operation,

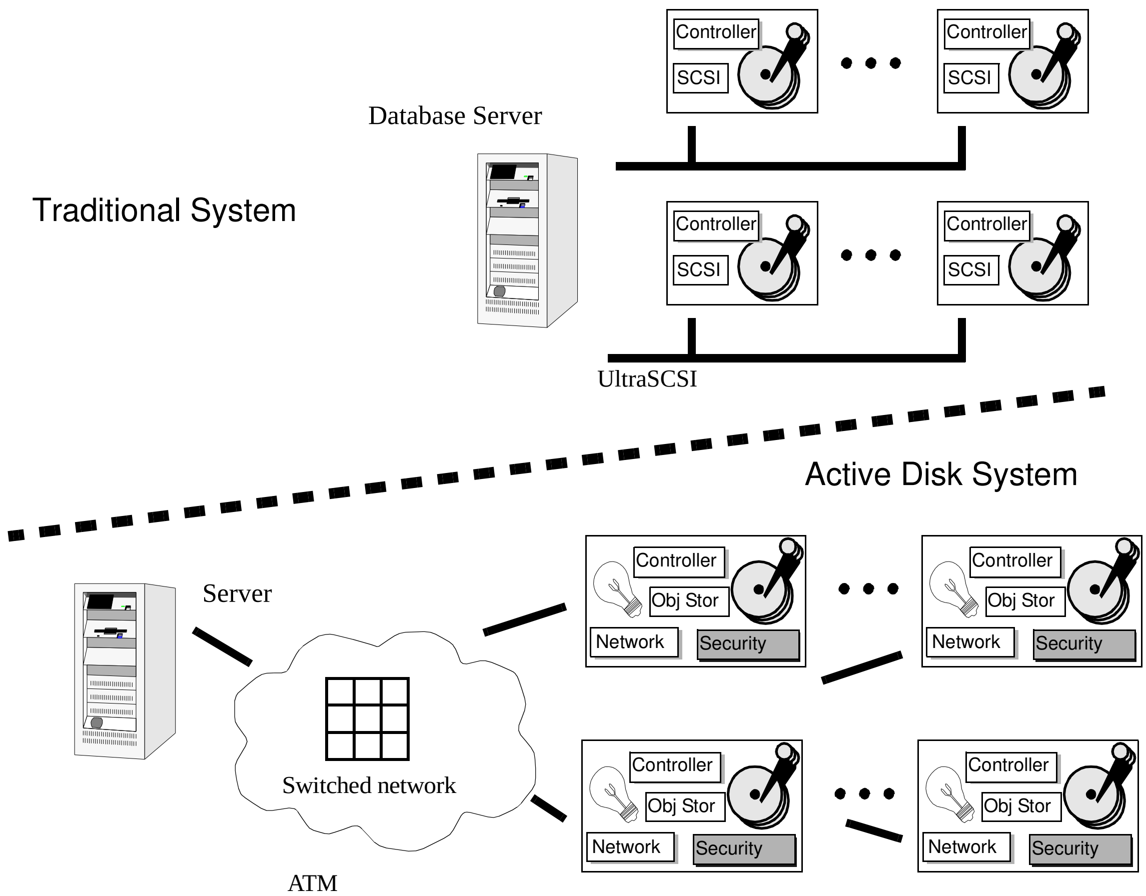


Figure 4-1 Architecture of an Active Disk system vs. a traditional server. The top diagram shows a traditional server system with directly-attached SCSI disks. The lower picture shows an Active Disk system using network-attached disks, and including the object interface and security system at the drives.

which will be discussed in detail in the Database section below. The individual disks can apply the filter to their local data pages, and return only the records that match a given condition. Similar applications are also possible in the area of multimedia, where image processing algorithms can be performed directly at the disks, before data is transferred to the host.

4.1.1 Data Mining - Nearest Neighbor Search

The first data mining application examined is a variation on a standard database search that determines the *k* items in a database of attributes that are closest to a particular input item. This is used for queries that wish to find records in a database that are most similar to a particular, desirable record. For example, in a profile of risk for loan applications there are a number of determining factors and the desire is to be "close" on as many of them as possible. Someone who is close in age may be far in terms of salary, or level of education, and so on. This means that the "nearest neighbor" when all the attributes are considered together, will be the best match. This also means that standard indexing techniques, that allow access to records based on a small subset of the "key" attributes, are not sufficient, all attributes in the record must be considered.

The nearest neighbor application uses synthetic data created by a program from the Quest data mining group at IBM Almaden [Quest97] that contains records of individuals applying for loans and includes information on nine independent attributes: <age>, <education>, <salary>, <commission>, <zip code>, <make of car>, <cost of house>, <loan amount>, and <years owned>. In searches such as this across a large number of attributes, it has been shown that a scan of the entire database is as efficient as building extensive indices [Berchtold97, Berchtold98]. Therefore, an Active Disk scan using the "brute force" approach is appropriate. The user provides a target record as input and the application processes records from the database, always keeping a list of the k closest matches so far and adding the current record to the list if it is closer than any already in the list. Distance, for the purpose of comparison, is the sum of the simple cartesian distance across the range of each attribute. For categorical attributes the Hamming distance between two values is used: a distance of 0.0 is assigned if the values match exactly, otherwise 1.0 is assigned.

For the Active Disk system, each disk contains an integral number of records and the comparisons are performed directly at the drives. The host sends the target record to each of the disks which determine the k closest records in their portions of the database. These lists are returned to the server which combines them to determine the overall k closest records. Because the application reduces the records in a database of arbitrary size to a constant-sized list of k records, the selectivity as defined in the previous chapter is arbitrarily large. The memory state required at each disk is simply the storage for the current list of k closest records.

4.1.2 Data Mining - Frequent Sets

The second data mining application is an implementation of the *Apriori* algorithm for discovering association rules in sales transactions [Agrawal95]. Again, synthetic data is generated using a tool from the Quest group to create databases containing transactions from hypothetical point-of-sale information. Each record contains a <transaction id>, a <customer id>, and a list of <items> purchased. The purpose of the application is to extract rules of the form "if a customer purchases items A and B, then they are also likely to purchase item X" which can be used for store layout or inventory decisions. This is a popular type of analysis in retail settings where "baskets" of a particular purchase give clues to what types of items people purchase together on a particular trip to the store. One of the more famous results from this type of analysis is a basket that included diapers and beer and was attributed to young fathers sent to the grocery store on Sunday evenings.

The computation is done in several passes, first determining the items that occur most often in the transactions (the *1-itemsets*) and then using this information to generate pairs of items that occur often (*2-itemsets*) and then larger groupings (*k-itemsets*). The threshold of "often" is called the *support* for a particular itemset and is an input parameter

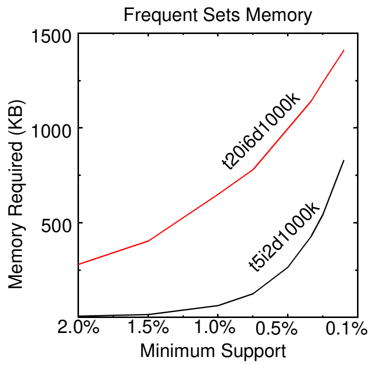


Figure 4-2 Memory required for frequent sets. The amount of memory necessary for the frequent sets application increases as the level of support required for a particular rule decreases. Very low support values may require multiple megabytes of memory at each Active Disk.

to the application (e.g. requiring support of 1% for a rule means that 1% of the transactions in the database contain a particular itemset). Itemsets are determined by successive scans over the data, at each phase using the result of the *k*-itemset counts to create a list of candidate *(k+1)*-itemsets, until there are no *k*-itemsets above the desired support.

In the Active Disks system, the counting portion of each phase is performed directly at the drives. The host produces the list of candidate *k*-itemsets and provides this list to each of the disks. Each disk counts its portion of the transactions locally, and returns the local counts to the host. The host then combines these counts and produces a list of candidate *(k+1)*-itemsets that are sent back to the disks. The application reduces the arbitrarily large number of transactions in a database into a single, variably-sized set of summary statistics - the itemset counts - that can be used to determine relationships in the database. The memory state required at the disk is the storage for the candidate *k*-itemsets and their counts at each stage.

There can be significant variation in the size of these counts, determined largely by the value of the support parameter. The chart in Figure 4-2 shows the memory requirements across a range of support values on two different databases. The lower a support value, the more potential itemsets are generated in successive phases of the algorithm and the larger the state that must be held on each disk. In normal use, the support value will tend toward the higher values since it is difficult for a human analyst to deal with the large number of rules generated with a low support value, and because the lower the support, the less compelling the generated rules will be in terms of their relative frequency and overall relevance for decision-making. For very low support values, however, limited memory at the Active Disks may become an issue.

4.1.3 Data Mining - Classification

The use of data mining for classification to elicit patterns from large databases is becoming popular over a wide range of application domains and datasets [Fayyad98, Chaudhuri97, Widom95]. Many data mining operations including nearest neighbor search, association rules, ratio and singular value decomposition [Korn98], and clustering [Zhang97, Guha98] eventually translate into a few large sequential scans of the entire

Figure 4-3 Edge detection in a scene outside the IBM Almaden Research Center. On the left is the raw image and on the right are the edges detected with a brightness threshold of 75. The data is a set of snapshots from IBM Almaden's CattleCam [Almaden97] and the application attempts to detect cows in the landscape above San Jose.

data. These algorithms can be mapped to Active Disks in much the same way as the nearest neighbor search and association rules described above, with the basic computation "core" operating directly at the drives and only final results being combined at the host. This could also be generalized to a particular set of primitives that might provide, for example, a mechanism to evaluate a neural network, specified in some standard way, across all the items in a data set in parallel.

4.1.4 Multimedia - Edge Detection

The first multimedia application is an image processing algorithm, specifically an application that detects the edges or corners of "objects" in a scene [Smith95]. The application processes a database of 256 KB grayscale images and returns the edges found in the data using a fixed 37 pixel mask. A sample image is shown in Figure 4-3 with the original image on the left and the extracted edges on the right. This models a class of image processing applications where only a particular set of features (e.g. the edges of "objects") is important, rather than the entire image. This includes any tracking, feature extraction, or positioning application that operates on only a small subset of *derived attributes* extracted from the original image. These attributes may include features such as detected edges, objects or color vectors. When used with Active Disks, the processing at the host can make use of the edges directly, rather than having to perform the expensive feature extraction algorithms on the raw image. The expensive pre-processing is done at the drives, and the host can operate on the coordinates of the objects directly.

Using the Active Disk system, edge detection for each image is performed directly at the drives and only the edges are returned to the central server. A request for the raw image in Figure 4-3 returns only the data on the right, which can be represented much more compactly as a simple list of coordinates. The amount of data transferred is reduced by a factor of almost 30, from 256 KB for the image to 9 KB for the edges in the sample image. The memory state required on each drive is enough memory for a single image that must be buffered and processed as a whole.

4.1.5 Multimedia - Image Registration

The second image processing application examined performs the image registration portion in the processing of an MRI brain scan [Welling98]. Image registration determines a set of parameters necessary to register (rotate and translate) an image with respect to a reference image in order to compensate for movement of the subject during the scanning. The application processes a database of 384 KB images and returns a set of registration parameters for each image. This application is the most computationally intensive of the ones studied here. The algorithm performs a Fast Fourier Transform (FFT), determines the parameters in Fourier space and computes an inverse-FFT on the resulting parameters. In addition to higher total computation, the algorithm may also require a variable amount of computation, depending on the image being processed, since it is solving an optimization problem using a variable number of iterations to converge to the correct parameters. This means that, unlike the applications discussed so far, the per byte cost of this algorithm varies significantly with the data being processed.

For the Active Disk system, this application operates similarly to the edge detection. The reference image is provided to all the drives and the registration computation for each processed image is performed directly at the drives with only the extracted parameters (about 1500 bytes for each image) returned to the host. The application reduces the amount of data transferred to the server by a large, fixed fraction as shown in the table. The memory state required at each drive is the storage for the reference image being compared and for the entire image currrently being processed.

Data Parameters

$|S|$ = size of relation S (pages)

k_{bytes} = size of key (bytes)

i_{bytes} = size of tuple (bytes)

p_{bytes} = size of page (bytes)

p_{tuples} = size of page (tuples)

Host Parameters

M = memory size of the host

r_d = disk raw read rate

r_w = disk raw write rate

s_{cpu} = CPU speed of the host

r_n = host network rate

Active Disk Parameters

m = memory size of the disk

s_{cpu}' = CPU speed of the disk

r_n' = active disk network rate

r_a = aggregate network fabric rate

Application Parameters

w_{sort} = cycles per byte of sorting

w_{merge} = cycles per byte of merging

4.2 Sorting

There are many reasons for sorting a set of records, and sort is a popular system benchmark. Sorting is most often done in the context of database systems, where it is usually combined with another operation, as discussed in the next sections. First, the basics of sorting in an Active Disk system.

There are several ways to partition a sorting algorithm on Active Disks, depending on the available functionality at the disks. The primary resource constraints for sorting are the disk bandwidth (each item of data has to be read and written at least once), network bandwidth (how often the data must move between the source disks, the host, and the destination disks on the interconnect), and memory size (which determines how large individual runs in a multiple-pass sort will be). As we will see, the main determinant of performance for large data sets turns out to be the interconnect bottleneck.

4.2.1 Merge Sort

The most common method for out-of-core sorting is Merge Sort [Knuth79, vonNeumann63] which performs a sequence of in-memory sorts on small subsets of the data, followed by a series of merge phases that combine the sorted subsets until the entire dataset is in order. In a normal merge sort, a host performs the following series of steps:

	Sort Phase	*Transfer*
A	read data from disk	-
B	transfer data across network to host	disk -> host
C	sort and create sorted runs	-
D	transfer sorted runs across network to disks	host -> disk
E	write sorted runs back to disk, average run length of *2M*[a]	-

a. assuming replacement selection is used as the local sort algorithm, this would be only *M* if quicksort is used, the difference between these two local sorting algorithms are discussed in Section 4.2.3.

	Merge Phase	*Transfer*
F	read sorted runs from disk, save *M* data still in memory	-
G	transfer data across network to host, save *M* data still in memory	disk -> host
H	merge sorted runs	-
I	transfer merged data across network to disks	host -> disk
J	write merged data back to disk	-

This algorithm requires four complete transfers of the data set across the network, in steps B, D, G, and I[1] and two complete reads and writes of the data to and from disk (with data of size *M* that can be retained in the memory between phases and must not be re-written or re-read).[2]

In an Active Disk algorithm, we save two of these transfers by having each drive perform the sorting of its own data locally, and performing only the final merge step at the host. Instead of providing the raw data to the host, the drives provide already sorted runs that must simply be merged by the host. This leads to a modified algorithm as follows:

1. in practice, *2M* bytes (minus a bit required for the merge buffers) of this traffic can be saved by retaining as much data from the last run as will fit before starting the merge, this saves *M* bytes of write and re-reading. The equations that follow will assume this optimization.

2. note that this is not the fully general algorithm for MergeSort. If the data size is sufficiently large, or the number of buffers available for merging is sufficiently small, then multiple merge phases will be required to generate the final sorted output. The number of merge phases required is $\log_{B-1}(N/B)$ where N is the size of the data in pages and B is the number of buffers available for merging [Ramakrishnan98]. This means that in a system with 1 GB of memory (131,072 pages of 8 KB each), the data would have to be larger than 128 TB before a second merge phase is required. In order to keep the formulas readable, this analysis assumes that the data is smaller than this or that M is sufficiently large to require only a single merge phase.

	Sort Phase (Active Disks)	*Transfer*
A	read data from disk	-
C1	sort and create sorted runs locally	-
C2	write sorted runs back to disk, average run length of *2m*	-
C3	read sorted runs from disk	-
C4	merge locally	-
E	write merged runs back to disk[a], average run length of $\|S\|/n$	-

a. note that this leads to a different number of runs than the host-based algorithm. The primary effect of this is the amount of memory required at the host for merging. If enough memory is available, it is possible to make step C2 be step E and drop steps C3 and C4 altogether. This will lead to average runs of length *2m* instead of $|S|/n$.

	Merge Phase (Active Disks)	*Transfer*
F	read sorted runs from disk	-
G	transfer data across network to host	disk -> host
H	merge sorted runs	-
I	transfer merged data across network to disks	host -> disks
J	write merged data back to disk	-

We now have only two complete network transfers at G and I and we have taken advantage of the ability of the disks to compute locally. In a network-limited system, this saves half the traffic on the network and will reduce the runtime of the sort by nearly one half.

The performance of these two methods can be modeled by a simple set of equations using the parameters listed above. Starting as follows:

$$t_{read + sort} = max\left(\frac{|S|}{d \cdot r_d}, \frac{|S|}{r_n}, \frac{|S| \cdot p_{\text{tuples}} \cdot w_{\text{sort}}}{s_{\text{cpu}}}\right)$$

where the time for the first half of the Sort Phase is the largest of three parts, the time to read the data off the disks, the time to transfer it to the host, and the time to sort all the tuples at the host, then:

$$t_{write + runs} = max\left(\frac{|S| - M}{r_n}, \frac{|S| - M}{d \cdot r_w}\right)$$

to transfer the data back to the drives in runs (except for one memory-full which is retained at the host), and to write it back to the disks, then the Merge Phase in two steps:

$$t_{read+merge} = max\left(\frac{|S|-M}{d \cdot r_d}, \frac{|S|-M}{r_n}, \frac{|S| \cdot p_{\text{tuples}} \cdot w_{\text{merge}}}{s_{\text{cpu}}}\right)$$

to read the runs back from the disks (except for the memory-full which we kept at the host), transfer them to the host, and merge them into the final output, and finally:

$$t_{write} = max\left(\frac{|S|}{r_n}, \frac{|S|}{d \cdot r_w}\right)$$

to transfer the final output back to the drives, and write the sorted output back to the disks. This gives an overall time of:

$$t = t_{read+sort} + t_{write+runs} + t_{read+merge} + t_{write}$$

and a throughput of:

$$\text{throughput} = \frac{|S| \cdot p_{\text{bytes}}}{t}$$

for the traditional server.

For the Active Disk system, we have a similar set of equations as:

$$t_{read+sort} = max\left(\frac{(|S|/d)}{r_d}, \frac{(|S|/d) \cdot p_{\text{tuples}} \cdot w_{\text{sort}}}{s_{\text{cpu}}},\right)$$

to read the local part of the data at each drive, and sort it (note that there is no network transfer in the Sort Phase), then:

$$t_{write+runs} = \frac{(|S|/d)-m}{r_w}$$

to write the sorted runs back to the disk, then:

$$t_{read+merge} = max\left(\frac{(|S|/d)-m}{r_d}, \frac{(|S|/d) \cdot p_{\text{tuples}} \cdot w_{\text{merge}}}{s_{\text{cpu}}},, \frac{|S|}{r_n}, \frac{|S| \cdot p_{\text{tuples}} \cdot w_{\text{merge}}}{s_{\text{cpu}}}\right)$$

to read the runs back off the disk, merge them locally (using the drive CPU), transfer the merged runs to the host, and merge the runs from all the drives at the host (using the host CPU), and finally:

$$t_{write} = max\left(\frac{|S|}{r_n}, \frac{(|S|/d)}{r_w}\right)$$

to transfer the data back across the network to the drives, and write the final output, which gives an overall time of:

$$t_{active} = t_{read+sort} + t_{write+runs} + t_{read+merge} + t_{write}$$

and a throughput of:

$$\text{throughput}_{\text{active}} = \frac{|S| \cdot p_{\text{bytes}}}{t_{active}}$$

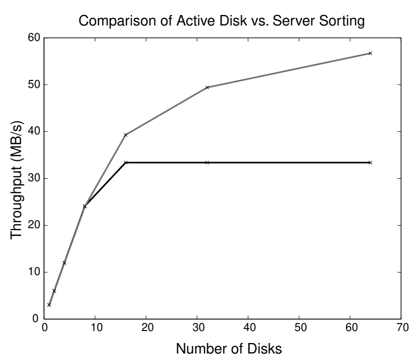
and captures the more efficient processing at the disks.

In terms of the equations of Chapter 3, this is a two-phase computation, with $t_{read+sort} + t_{write+runs}$ as the first phase, followed by a synchronization point where all the drives must catch up, followed by $t_{read+merge} + t_{write}$ as the second phase. As described here, the sort phase has virtually no serial fractions, so $p = 1.0$ for the sort phase. For the merge phase, the serial fraction is the portion of $t_{read+merge}$ that is bottlenecked on the merging of runs in the CPU of the host, and may be a significant factor, depending on the relative performance of the CPU and network.

Figure 4-4 shows the performance of merge sort on a large data set using a traditional server system and a system with an equivalent number of Active Disks. We see that up to a certain number of disks, performance is disk-bound, so the Active Disk system has no advantage. It is not until the traditional server hits its network bottleneck that the Active Disk system begins to outperform the server, scaling linearly as additional disks are added.

4.2.2 Key Sort

A further extension that uses the ability to compute at the disks is to perform the sort using only the keys and introduce a "shuffling" phase once the keys are in sorted order. In practice, most in-core sorting is done on keys only, rather than entire records, since keys are usually small relative to whole records. If we apply this in an Active Disk system, we can save several network transfers and one write of the entire data to disk by doing run formation using only the keys, instead of the full records. This means that, in the host-based algorithm shown above, we can reduce the amount of data transferred by moving



Parameter	Host	Active Disk (per disk)
Input Size	128 GB	128 GB
Number of Disks	64	64
Disk Read Rate	15 MB/s	15 MB/s
Disk Write Rate	10 MB/s	10 MB/s
CPU	500 MHz	25 MHz
CPUsort	15 cycles/tuple	15 cycles/tuple
CPUmerge	10 cycles/tuple	10 cycles/tuple
Memory	1 GB	8 MB
Network	133 MB/s	agr 1000 MB/s

Figure 4-4 Performance of sorting in Active Disk vs. a traditional server system. Results from analytic modeling via the equations presented above. Parameters as detailed on the right. we see that with a small number of disks, both system are disk-bound retrieving and writing data. Once sufficient disk bandwidth is available, the server system next bottlenecks on it's network link. Since it performs approximately four full transfers of the data on the network, it bottlenecks at 133 MB/s / 4 transfers. Since the Active Disk system performs only two transfers, it will plateau and finally bottleneck near 66 MB/s, but continues to scale until the disk bandwidth nears that point.

only the keys in the initial Sort and Merge Phases, and moving the balance of the data only in the Shuffle Phase, as follows:

Sort Phase (as above, but using only the keys, a data reduction of $k_{\text{bytes}} / i_{\text{bytes}}$)

Merge Phase (as above, but using only keys)

	Shuffle Phase	*Transfer*
K	read data and keys from disk, save *M* keys still in memory	-
L	transfer data & keys across network to host, save *M* keys	disk -> host
M	shuffle data to the appropriate destination disk based on keys	-
N	transfer shuffled data across network to disks	host -> disks
O	write sorted data back to disk	-

This algorithm transfers all the data on the network only twice, in steps L and N and requires only two reads and one write of the data. The (much smaller) keys are transferred on the network five times, at steps B, D, G, I and L[1].

This means the equations above can be modified as follows:

$$t_{read+sort} = max\left(\frac{|S|}{d \cdot r_d}, \frac{|S|}{r_n}, \frac{\left(|S| \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right)\right) \cdot p_{\text{tuples}} \cdot w_{\text{sort}}}{s_{\text{cpu}}}\right)$$

which captures the smaller amount of data to be sorted (although it must still all be read from the drives and sent to the host), then:

$$t_{write+runs} = max\left(\frac{\left(|S| \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right)\right) - M}{r_n}, \frac{\left(|S| \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right)\right) - M}{d \cdot r_w}\right)$$

which transfers only the keys back to the drives in sorted runs, and writes only the keys back to the disks (note that this does require additional storage on disk for the keys, while the full-record sort could always re-use the space occupied by the unsorted records), then:

1. note that the fraction of keys that can be retained in the *M* memory at the host (and which must not be repeatedly transferred on the network) is significantly more than the amount of data that can be retained in the same memory. With even a reasonably small memory and small key sizes, it should easily be possible to avoid the Merge Phase for the key sorting altogether, although it would still be required in the Active Disk case.

$$t_{read + merge} = max\left(\frac{|S| + |S| \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right) - M}{d \cdot r_d}, \frac{|S| + |S| \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right) - M}{r_n}, \frac{|S| \cdot p_{\text{tuples}} \cdot w_{\text{merge}}}{s_{\text{cpu}}}\right)$$

to read the sorted runs of keys and the records themselves back from the disks, transfer everything to the host, and merge the records into the final output, and finally:

$$t_{write} = max\left(\frac{|S|}{r_n}, \frac{|S|}{d \cdot r_w}\right)$$

which remains the same as before, to transfer the final sorted records back to the drives and write them to disk.

For the Active Disk system, the changes give:

$$t_{read + sort} = max\left(\frac{\left(\frac{|S|}{d}\right)}{r_d}, \frac{\left(\frac{|S|}{d} \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right)\right) \cdot p_{\text{tuples}} \cdot w_{\text{sort}}}{s_{\text{cpu}}},\right)$$

to read the local part of the data at each drive, and sort it (again, only the keys need to be sorted, and since the computation is done locally, nothing needs to be transferred on the network), then:

$$t_{write + runs} = \frac{\left(\frac{|S|}{d} \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right)\right) - m}{r_w}$$

to write the sorted runs back to the disk, then:

$$t_{read + merge} = max\left(\frac{|S|/d - m}{r_d}, \frac{|S|/d \cdot p_{\text{tuples}} \cdot w_{\text{merge}}}{s_{\text{cpu}}},\right.,$$

$$\left.\frac{|S| + |S| \cdot \left(\frac{k_{\text{bytes}}}{i_{\text{bytes}}}\right)}{r_n}, \frac{|S| \cdot p_{\text{tuples}} \cdot w_{\text{merge}}}{s_{\text{cpu}}}\right)$$

to read the runs from the disk, merge them locally, transfer the merged runs to the host, and merge the runs from all the drives at the host, and finally:

$$t_{write} = max\left(\frac{|S|}{r_n}, \frac{(|S|/d)}{r_w}\right)$$

which remains the same as before, to write the final sorted output.

Furthermore, if the Active Disks provide a disk-to-disk transfer mechanism (without host intervention on each transfer), then the shuffling of steps M and N can be performed at the full cross-section bandwidth of the disk interconnect fabric, rather than at the (much lower) rate of the network interface into the host.

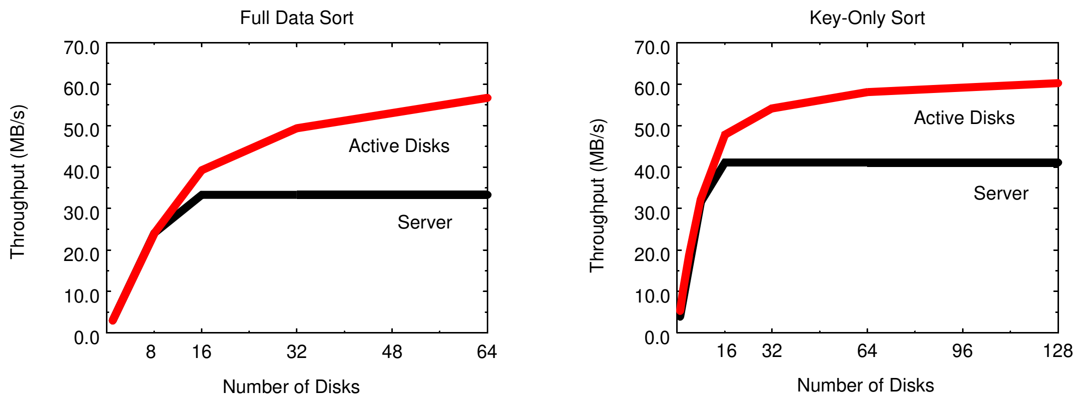
In terms of the equations of Chapter 3, this is a three-phase computation, with sort as the first phase, followed by a synchronization point where all the drives must catch up, followed by merge. and then shuffle. As described, the sort phase again has virtually no serial fractions, so $p = 1.0$. For the merge phase, the serial fraction is the portion of $t_{read + merge}$ that is bottlenecked on the merging of runs in the CPU of the host as before, but only for the merging of keys, which should be significantly less costly than the merging of full records. When disk-to-disk transfer is used, the shuffle phase is also fully parallel, with $p = 1.0$.

With a disk-to-disk transfer mechanism, this algorithm also outperforms the Active Disk system in the simple Merge Sort because the full data must never pass through the host (only the keys go to the host during the Key Merging phase). Without direct disk-to-disk transfers, this algorithm will perform slightly worse than the simple Merge Sort because of the extra transfers of all the keys on the network (since the data never left the disks until the final merge phase, sending the keys to the host for merging and then the entire data increases the total amount of data transferred[1]).

Figure 4-5 shows the performance of the key-only sorting algorithm, again comparing the traditional server with an Active Disk system. We see that the throughput of both the server and the Active Disk system increases in direct proportion to the reduction in network traffic. The Active Disk system benefits more because it is able to perform the initial "filtering" of keys directly at the disks, while the server system must first transfer all the data to the host, at which point it can also drop everything except the keys for the remainder of the sort and merge phases. The full power of Active Disks is shown in the chart on the right of Figure 4-5, where we allow direct disk-to-disk data transfers. These transfers proceed at the full aggregate bandwidth of the network fabric, rather than being limited by the network interface into the host.

One variation on this basic algorithm that would be more efficient than the above description is an algorithm where the initial Key Sort and Merge phases are replaced with a Sampling phase that examines only a subset of the keys and makes a "guess" about the expected distribution of the result data, instead of sorting the keys to determine the exact distribution [Blelloch97]. This is the same as the Key Sort algorithm with Active Disks except that only a subset (say 10%) of the keys are initially transferred to the host for sorting and merging and only a listing of quantile boundaries must be returned to the disks (rather than the entire, exact lists of keys). In this case, it would also be possible to move

1. unless the records in the final shuffle phase are sent without the key portion and are re-united with their keys before being written at the host, but this seems needlessly complex (and of low value for small keys).



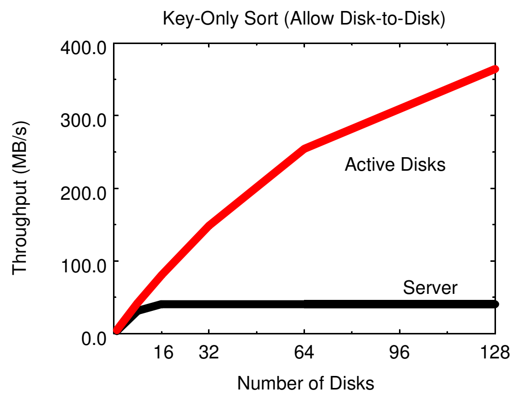


Figure 4-5 Comparison of sorting in Active Disk vs. a traditional system. The first chart compares the server against the Active Disk system using the algorithm that sorts the entire data set. We see that the Active Disk system is comparable at low numbers of disks, and just under twice as fast as the server system bottlenecks due to the additional network transfers. The second chart shows the simple key-only sorting. The basic key-only sort improves performance by 10-15% in both the Active Disk and traditional case. But the real benefit of Active Disks can be seen in the chart on the right, which allows direct disk-to-disk data transfers. In this case, the main body of the data never moves across the network to the host, only the keys are sent to the host for merging, and then back to the disks which can operate independently to transfer the data to its final destination. There are some difference in the disk performance in the two cases, and that is discussed in more detail in the next section. All numbers are analytic results based on the formulas of the preceding section.

the local Sort phase to the end of the algorithm. The same Sort and Merge phases must still be done locally, either at the source drive before the records as sent or at the destination drive before they are written back to the disk.

This is essentially the algorithm proposed by the NowSort group at Berkeley [Arpaci-Dusseau97] and will work well if the sample taken closely matches the final data distribution[1]. Whatever mismatch occurs will require a final Fixup phase where drives exchange “overflow” records with each other to rebalance the data.

To get an idea of the applicability of these optimizations, Table 4-1 shows the data sizes for both normal and key-only sort using several queries from the TPC-D benchmark [TPC98] and a data set from the Datamation sort benchmark [Gray97a]. We see significant savings for the sorts within the database queries, and a factor of exactly ten for the Datamation sort, which specifies 100 byte records and 10 byte keys.

1. the algorithm described in [Arpaci-Dusseau97] actually assumes a uniform key distribution, but they mention the need for a Sampling phase if the data is known to be non-uniform.

Query	Input Data (KB)	Sort - Full Data (KB)	Sort - Keys Only (KB)	Savings (selectivity)
Q1	126,440	33,935	1,131	30.0
Q4	29,272	145	9	16.1
Q6	126,440	-	-	-
Datamation	32,768	32,768	3,276	10.0

Table 4-1 Sort size and data reduction. Sizes for sorts within TPC-D queries using plans as chosen by the default PostgreSQL optimizer, and the Datamation benchmark for a 32 GB input file.

4.2.3 Local Sorting Algorithms

The local sort algorithm used in all of the above examples has a significant impact on the overall performance and flexibility of the computation. The two main choices of local sort algorithms for general data types is *quicksort*, or some variant of *replacement selection*.

The advantage of using replacement selection is that it produces longer runs than quicksort (with an average size of *2M* - twice the size of the available memory - and even higher when the data is already partially sorted, as is often true in real data sets). It is straightforward to extend replacement selection to do aggregation or duplicate elimination, where the amount of memory required will be proportional to the output size, rather than the input size, a considerable advantage for highly selective aggregates or projections. There has also been considerable work on developing memory-adaptive versions of replacement selection that can easily adapt to changing memory conditions [Pang93a]. This variant of replacement selection can give up memory pages as it sorts (resulting in smaller average run size, but not inhibiting the overall progress of the sort) and can make use of additional pages as they become available (to increase average run size). Replacement selection can take advantage of as much or as little memory as is available at any given time, with a consequent variation in run size and in the total number of runs to be merged later.

The major disadvantage of replacement selection is its poor cache performance on modern processors [Nyberg94] due to the non-local access pattern for insertions into the sorted heap. Quicksort, on the other hand, has better cache behavior, but is limited to smaller runs (with a size of exactly *M* - the amount of available memory). It is also more difficult to adapt quicksort to perform aggregation or duplicate elimination as it sorts, or to make it adaptive to changes in the available memory as described above for replacement selection. In order for quicksort to proceed, there must be enough memory to sort a memory-full of data, and this memory must remain allocated until the sort is complete.

The use of radix sort, or other, more sophisticated schemes can yield improved performance if that data types of the keys is known and can be optimized for in advance [Arpaci-Dusseau97].

Algorithm	Instructions (inst/byte)	User (cycles/byte)	System (cycles/byte)	Unique Instructions (KB)	Executed More Than Once (KB)	Other
quicksort	15.77	28.6	17.8	13.0	10.1	
replacement selection	23.23	78.0	18.0	9.7	6.4	
merge	7.90	8.8	15.4	16.9	12.1	96 KB memory

Table 4-2 Performance of local sort algorithms. The table compares the performance of quicksort and replacement selection, showing instructions and cycles per byte, the total size of the code executed, and the size of the instructions executed more than once.

A comparison of the two algorithms for sorting is shown in Table 4-2 where we see the overall cycles/byte and code sizes for the two algorithms. We see that quicksort has a somewhat lower instructions/byte than replacement selection - about 25% less - but the real impact is in the cycles/byte cost, where the cache effects are clear - in this case, quicksort is about a factor of three less than replacement selection. A further comparison is shown in Table 4-3 which gives the performance of the two algorithms on a StrongARM

Algorithm	User (cycles/byte)	System (cycles/byte)	Other
quicksort	31.6	19.2	
replacement selection	110.8	18.0	

Table 4-3 Performance of sort on an embedded processor. The table compares the performance of quicksort and replacement selection on a 200 MHz StrongARM processor.

processor such as the one proposed in Chapter 2 for future Active Disk processors. It may be possible to further optimize replacement selection for execution in an embedded platform such as the StrongARM, but the overall performance benefit of quicksort will continue to be at least a factor of two. This means that the longer runs and greater flexibility of replacement selection must overcome the performance drop in order to be competitive. For processing on Active Disks, the flexibility to adapt to changing memory conditions, and the ability to perform duplicate elimination and aggregation during the sort still make replacement selection a compelling choice.

4.3 Database

There are three core operations in a relational database system: select, project, and join [DeWitt90]. This section uses the PostgreSQL relational database system to show how Active Disks can benefit by partitioning each of these operations between the Active Disks and the host. PostgreSQL was chosen as a platform for this examination because it has been extensively reported on in the database literature [Stonebraker86], and because on open-source version was available that could be examined and modified in detail without trade secret or publishing restrictions [PostgreSQL99].

l_orderkey	l_shipdate	l_qty	l_price
7010	10-05-98	23	29356.3
36070	11-27-98	31	34167.9

relation S

l_orderkey	l_shipdate	l_qty	l_price	l_disc
1730	01-25-93	6	11051.6	0.02
3713	04-12-96	32	29600.3	0.07
7010	10-05-98	23	29356.3	0.09

32742	05-05-95	8	9281.9	0.01
36070	11-27-98	31	34167.9	0.04

Figure 4-6 Illustration of basic select operation in a database system. A single relation is searched for attributes that match the given value. Note that there are two types of data reduction going on here. The first is the selection of matching records (5% in this example from TPC-D) and the second is the elimination of unneeded columns from the relation (3/4 of the data in this example).

4.3.1 Select

The **`select`** operation is an obvious candidate for an Active Disk function. The **`where`** clause in a SQL query can be performed directly at the drives, returning only the matching records to the host. This is the operation that was implemented by many of the early database machines, and is the basic function performed by the SCAFS search accelerator [Martin94]. The query is parsed by the host and the select condition is provided to all the drives. The drives then search all the records in a particular table in parallel and return only the records that match the search condition to the host. If the condition is highly selective, this greatly reduces the amount of data that must traverse the interconnect, as records that will not be part of the result must never leave the disks. The selectivity is simply the fraction of records that match the given search condition. This type of search requires very little state information at the drives, since only the search condition and the page currently being operated on must be stored in drive memory.

A basic select operation is illustrated in Figure 4-6 which shows the query:

```
select * from lineitem where l_shipdate > '1998-09-02'
```

using tables and test data from the TPC-D decision support benchmark.

4.3.2 Project - Aggregation

The purpose of sorting in database systems is almost always as an input step to another operation, either a join, an aggregation, or as the final step in a projection (duplicate elimination). The biggest benefit of performing sorting at Active Disks comes when it is combined with one of these other steps directly at the disks. Aggregation combines a set

l_disc	sum_revenue	sum_qty
0.02	39599.7	29
0.07	67936.6	71

relation S

l_orderkey	l_shipdate	l_qty	l_price	l_disc
1730	01-25-93	6	11051.6	0.02
3713	04-12-96	32	29600.3	0.07
7010	10-05-98	23	29356.3	0.02
32742	05-05-95	8	9281.9	0.07
36070	11-27-98	31	34167.9	0.07

Figure 4-7 Illustration of basic aggregation operation in a database system. A single relation is processed and values in the requested columns are summed together. Other operations include min, max, count, and average.

of records to compute a final sum, average, or count of groups of records with particular key values. If this summing or counting can be done at the disks as records are sorted on the group by columns (particularly attractive if replacement selection is used as the sorting algorithm, as discussed above) then the network traffic can be greatly reduced by returning only the sums or counts from the individual disks for final aggregation at the host. Similarly, if duplicate elimination can be done while sorting locally at the Active Disks, then the duplicate records must not be needlessly transferred across the network simply to be discarded at the host.

A basic aggregation operation is shown in Figure 4-6 which illustrates the query:

```
select sum(l_qty), sum(l_price*(1-l_disc)) group by l_disc
```

that totals up the number of items sold and the total amount of money taken in combined based on the `l_disc` attribute. All items with the same discount value are combined and total quantity and revenue reported.

4.3.2.1 Aggregation via Hashing

In most cases, an even more efficient way to do aggregation is via hashing [Graefe95]. Since sorted order isn't strictly required to aggregate groups of records, it is only necessary to combine records with the same key values, not completely sort the records. The primary difficulty with hashing is that it cannot easily output partial results if the amount of memory available is less than what is required for the entire output. The replacement selection algorithm, on the other hand, can output partial results in sorted order, so they can be easily combined in a final merge step, requiring only memory proportional to the number of runs created, rather than proportional to the size of the output. Basically, replacement selection provides a more adaptive and memory-efficient way to do aggregation than either hashing or full sorting followed by aggregation does.

4.3.2.2 Aggregation Filtering

There are several steps in a typical aggregation query where Active Disk filtering can be profitably applied. The first step is to remove the columns in a relation that are not relevant to a particular query result, for example, the address field of a customer record is not needed if we are totalling the total amount of money they owe, and the receipt date of particular order is not important if we are trying to determine how many of a given part have left our factory (in database terms, this is performing a projection on R to obtain R'). Figure 4-8 shows a few rows from the input table to Query 1. For this particular query, only 7 of the 16 columns in this table are required. In an Active Disk system, the rest of

"Report the amount of business that was billed, shipped, and returned. Summarize for for all items shipped up to 90 days from the last date in the database and include total price, total discounted price, total price plus tax, average quantity, average price, and average discount, grouped by order status and return status."

```
select l_returnflag, l_linestatus,
sum(l_quantity), sum(l_price),
sum(l_price*(1-l_disc)),
sum(l_price*(1-l_disc)*(1+l_tax)),
avg(l_quantity), avg(l_price),
avg(l_disc), count(*)
from lineitem
where l_shipdate <= '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

Figure 4-9 Business question and query text for Query 1 from TPC-D. The purpose of this query is to summarize about 95% of the item records in the database across price, discount, and quantity.

the data never needs to leave the drive and consume interconnect bandwidth. Figure 4-9 gives the business question being answered by Query 1 as well as the SQL query text. From looking at the query text, we can determine that only a subset of the columns in the table are needed to answer this query, the comment and address fields, for example, are

```
l_okey|l_quantity| l_price|l_disc|l_tax|l_rf|l_ls|l_shipdate|l_commitdate|l_receiptdate|l_shipmode|l_comment
------+----------+--------+------+-----+----+----+----------+------------+-------------+----------+---------
  1730|         6|11051.58|  0.02|    0|N   |O   |09-02-1998|  10-10-1998|   09-13-1998|TRUCK     |wSRnnCx2
  3713|        32|29600.32|  0.07| 0.03|N   |O   |09-02-1998|  06-11-1998|   09-28-1998|TRUCK     |MOgnCO1
  7010|        23|29356.28|  0.09| 0.06|N   |O   |09-02-1998|  08-01-1998|   09-14-1998|MAIL      |jPNQlx3i
 19876|         4| 6867.24|  0.09| 0.08|N   |O   |09-02-1998|  09-06-1998|   09-29-1998|AIR       |3nRkNn4
 24839|         8|12845.52|  0.05| 0.02|N   |O   |09-02-1998|  10-14-1998|   09-06-1998|REG AIR   |jlw61g3
 25217|        10| 18289.1|  0.05| 0.07|N   |O   |09-02-1998|  08-12-1998|   09-26-1998|TRUCK     |SQ7xS5
 29348|        29|41688.08|  0.05| 0.02|N   |O   |09-02-1998|  07-04-1998|   09-18-1998|FOB       |C0NxhzM
 32742|         8| 9281.92|  0.01| 0.03|N   |O   |09-02-1998|  07-17-1998|   09-19-1998|FOB       |N3MO1C
 36070|        31|34167.89|  0.04|    0|N   |O   |09-02-1998|  07-11-1998|   09-21-1998|REG AIR   |k10wyR
[...more...]
(600752 rows)
```

Figure 4-8 Format of the `lineitem` table, which is the largest in the TPC-D benchmark. The table serves as input to Query 1. Note that a few of the columns have been removed or shortened for presentation purposes. The full schema is provided in Appendix A and in the TPC-D benchmark specification [TPC98].

Query	Input Data (KB)	SeqScan Result (KB)	Scan Savings (selectivity)	Aggregate Result (bytes)	Aggregate Savings (selectivity)
Q1	126,440	34,687	3.6	240	147,997.9
Q4	29,272	86	340.4	80	1100.8
Q6	126,440	177	714.4	8	22,656.0

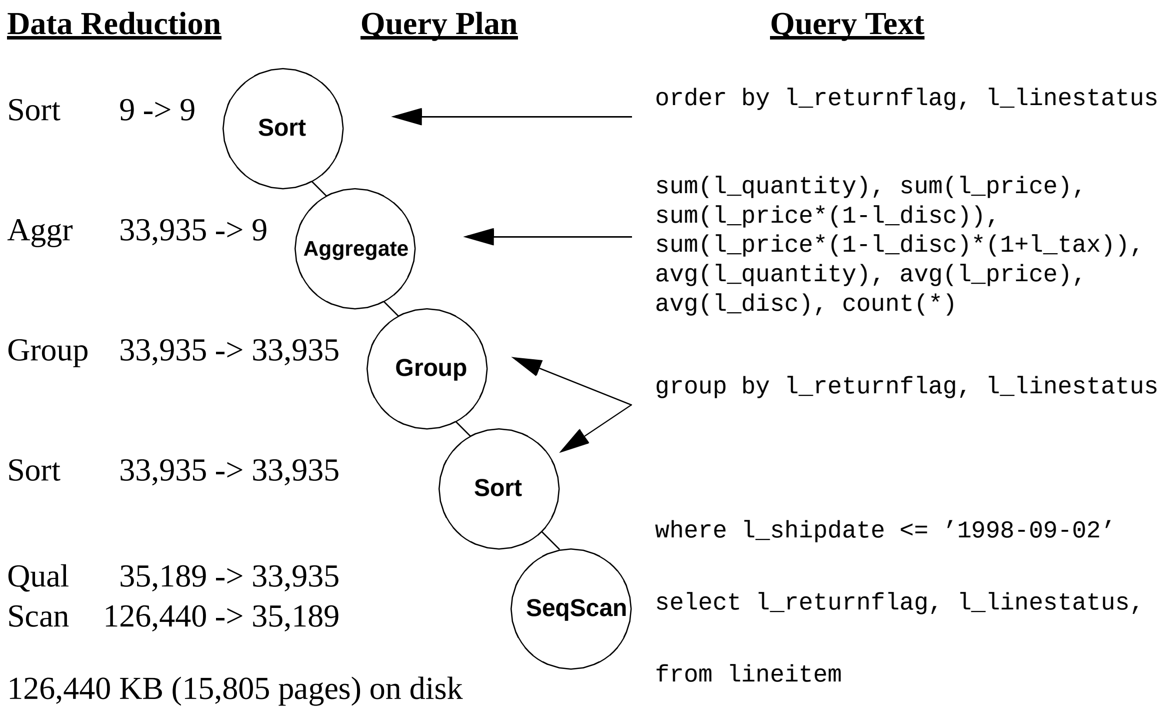
Table 4-4 Sizes and selectivities of several TPC-D queries. The table gives the data sizes and selectivities at intermediate stages of several TPC-D queries as executed by PostgreSQL. Note that since aggregation produces only one (or a few) values - a sum of values or a count of tuples for example - the selectivity is essentially infinite as the input size increases, i.e. it reduces an arbitrary-sized input to a fixed (and small) number of bytes.

never used by this query. In fact, the subset of columns that remain after the initial scan of the table as it comes off the disk, provide a close to four times reduction in the amount of data moved from the disks to the host. The Scan Savings column of Table 4-4 shows how much data reduction is possible simply by eliminating the unecessary columns from the largest table in a subset of the TPC-D queries.

Note that the negative impact of pre-scan at the drives is that the full records are not returned to the client for later caching. In the case of small relations, this decrease in cache efficiency could well outweigh the benefits of Active Disk processing. This requires the query optimizer to choose plans where small relations (for which caching will likely be benefitial) are returned to the host in their entirety, while large relations (which would "blow out" the cache at any rate) are processed at the disks and marked "uncacheable" at the host. This also avoids the cache coherence problem that would arise if Active Disks were able to process pages that may be dirty in the host's buffer pool. There are some additional concerns about locking, particularly in the presence of the UF1 and UF2 update functions in the TPC-D benchmark. These are important functions to consider in the design of an Active Disk database system, but there are a number of possible methods to minimize the impact of the updates on the decision support queries [Mohan92, Merchant92] and these issues are mentioned again in Section 7.5.2. The discussion that follows will assume that the relations being processed at the disks are uncacheable at the hosts, and that cursor stability is sufficient for the decision support queries.[1]

Figure 4-10 shows the entire plan for Query 1 as determined by the PostgreSQL optimizer, along with the amount of data reduction at each step in the query. We see that there is a factor of four reduction in data moved at the initial scan phase, a further 5% reduction in the qualification phase (the `where` clause) and then the final reduction of 5 orders of magnitude in the aggregation step, when everything is summarized down to only four result rows. The values shown in the figure are the size estimates made by the query

1. Note that this assumption, as well as the fact that our system has been in no way audited or approved by the normal TPC guidelines, means that are TPC-D results should be considered illustrative only of the types of benefits that might be possible in a fully audited and benchmarked system.



Query Result

l_rf	l_ls	sum_qty	sum_base_price	sum_disc_price	sum_charge	avg_qty	price	disc	count
A	F	3773034	5319329289.67	5053976845.78	5256336547.67	25.509	35964.01	0.049	147907
N	F	100245	141459686.10	134380852.77	139710306.87	25.625	36160.45	0.050	3912
N	O	7464940	10518546073.97	9992072944.46	10392414192.06	25.541	35990.12	0.050	292262
R	F	3779140	5328886172.98	5062370635.93	5265431221.82	25.548	36025.46	0.050	147920

(4 rows)

Figure 4-10 Text, execution plan, and result for Query 1 from the TPC-D benchmark. The right column shows the text of the query, the center diagram shows the final plan chosen by the optimizer for this execution, and the left column shows the amount of data reduction at each node in the plan. The query result is shown in the table at the bottom. This is the entire result, note the large data reduction, from 125 MB on disk to several hundred bytes in the final result.

optimizer, the actual sizes vary somewhat from this, as discussed in more detail in the Optimizer section of Chapter 6, but are the same order of magnitude.

The qualification condition does not provide a significant reduction in data moved for Query 1, but the conditions on Query 4 reduce the data to a tiny fraction of their original size. The Qualification Savings column of Table 4-4 shows the data reduction by qualification on the largest relation in the TPC-D queries. The Optimizer section in Chapter 6 provides additional details on how an Active Disk system could estimate the costs and benefits for a particular query, that can then be used to determine the appropriate partitioning across disks and hosts.

Table 4-4 shows several queries from the TPC-D benchmark and the savings in data transfer if filtering is performed at different steps in the query execution. We see that the

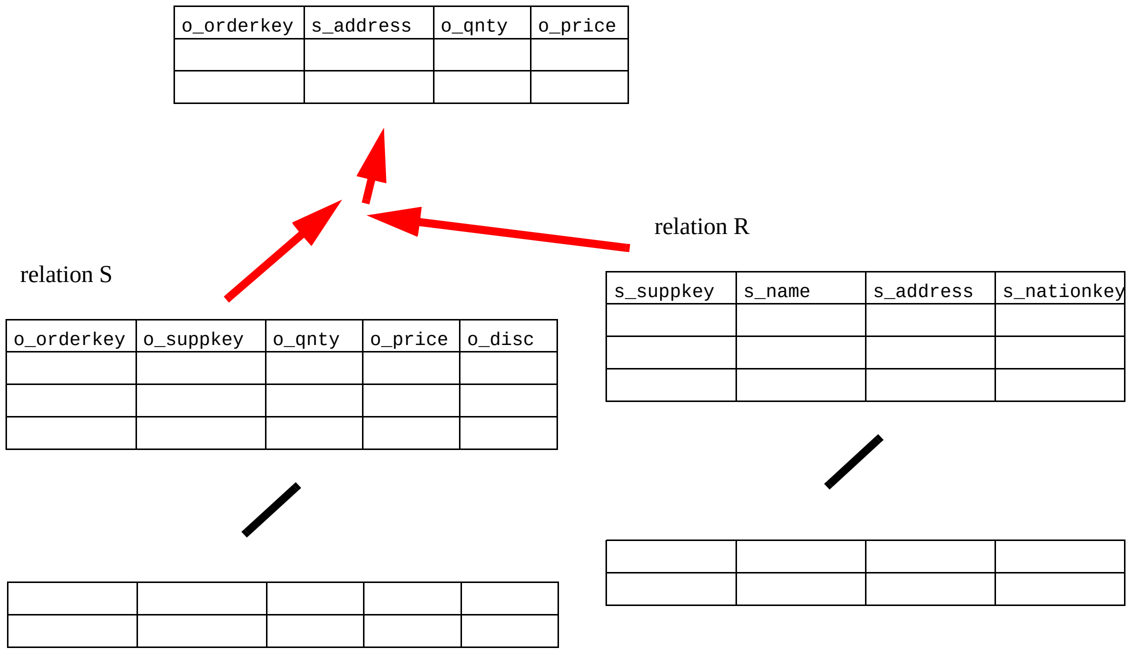


Figure 4-10 Illustration of basic join operation in a database system. Two relations are combined on a particular join attribute. Records from relation S that much the keys in relation R are selected and combined with the corresponding records from R. By definition, R is the smaller of the two relations and is also referred to as the inner relation, and S is the outer relation.

benefits of the initial table scan are significant, and the benefits to performing the entire aggregation at the disks are several orders of magnitude.

4.3.3 Join

Highly selective joins will benefit significantly from the reduction in data transfer by operating directly at the Active Disks. Joins will also benefit from the greater amount of CPU power provided by the Active Disks in aggregate vs. that available at the host. A join combines data from two (or more) relations, so it is more complex than the select or project, which operate on only a single relation at a time. Figure 4-10 illustrates the basic computation of a join on two relations.

4.3.3.1 Join Algorithms

There are a number of possible algorithms for performing joins, depending on the sizes of the relations being joined, the relative size of the inner and outer relation, and the sort order of the source relations. The purpose of a join is to combine two input relations, R and S on a single *join attribute*. If the value of the attribute for a particular record in R matches any record in S, then the combined record is output. The relation R is, by definition, the smaller of the two relations. It is also possible to perform *n-way* joins among a larger number of relations, but these are simply done as a series of 2-way joins (although the choice of join order can greatly affect the overall performance).

Nested-Loops is the most straightforward algorithm for performing a join, but is efficient only if R is extremely small. The name itself explains the basic algorithm, which proceeds as a nested loop (or a series of nested loops, for an *n*-way join), choosing one record from S and looping through R looking for matches, then choosing the next record from S and repeating the process. In this sense, R is the inner loop, and the iteration through S forms the outer loop, which is why R and S are often referred to as the inner and outer relations in a join, respectively. The basic advantage of this algorithm is that it requires very little memory, essentially only enough buffering for the two tuples currently being compared. This algorithm is used only for extrememly small relations when the overhead of building the hash table required for a hash-join would overcome the performance gain.

Hash-Join [Kitsuregawa83] uses a hash table as the basic data structure and has been shown to be the best algorithm choice except in the case of already sorted input relations [Schneider89, Schneider90]. Hash-Join first builds a hash table of R in memory and then reads S sequentially, hashing the join key for each record, probing the hash table for a match with R, and outputting a joined record when a match is found. This is more effective than Nested-Loops as each of the relation needs to be read only once, and is preferable whenever there is sufficient memory to hold the hash table of R. The amount of memory required will be proportional to the size of R, with some amount of overhead for the hash table structures.

Merge-Join takes advantage of the fact that the two input relations are already sorted on the join attribute and simply joins by merging the two lists of ordered records. It does not require repeated passes as in Nested-Loops because the records are known to be sorted, so the algorithm can process both R and S in parallel, without ever having to “look back” in either traversal, as in the merge phase in a Mergesort. This algorithm also has the memory advantage of Nested-Loops because only the tuples currently being examined need to be in memory. When only one of the relations is already sorted, the query optimizer must decide whether it is less expensive to sort the second relation and perform a Merge-Join, or simply revert to Hash-Join as if both were unsorted.

Hybrid Hash-Join is an extension of the hash-join [DeWitt84] that is used when the inner relation is larger than the amount of memory available. The Hybrid algorithm operates in two phases. A hash function is chosen that divides the space of join keys into several partitions. On the first pass, both R and S are read and records are output into per-partition buckets. The partitions are chosen in such a way that a hash table of R_i will fill the memory available. In the second phase, these buckets are read back in pairs, a hash table is built using the records of R_i and this table is probed with all the records of S_i. The first pass of this algorithm does not require a significant amount of memory because records are simply being divided into some number of output buckets. This fact makes possible a straightforward extension to the basic algorithm that partially combines the two phases and performs both the paritioning and the hashing for the first bucket at the same time. This means the memory is always kept full and only *n-1* buckets worth of data need to be written back to the disk. This extension also makes the algorithm very attractive for use as an *adaptive* algorithm. Whereas the basic Hash-Join requires that sufficient memory be available for the entire hash table of R before the operation can proceed, the Hybrid algorithm can adapt to changing memory conditions [Zeller90,Pang93]. It is straightforward for the Hybrid join to give up memory pages when they are needed elsewhere in the system or make use of additional pages that become available during its processing. It simply places more of the buckets onto disk, or loads additional buckets into memory. This algorithm requires temporary disk space equal to the combined size of the two relations (minus one memory-full of data that need never be written to disk). It can make use of any amount of memory available, up to the size of the complete hash table for R (at which point it simply becomes the basic Hash-Join algorithm). For both the basic Hash-Join and Hybrid Hash-Join, care must be taken in the choice of hash functions to ensure even-sized partitions and efficient use of the memory that is available at any given point.

4.3.3.2 Size of Relations

The primary cost of performing joins at the Active Disks is the cost of transferring all the parts of the inner relation that the disk does not already have (generally $|R| / (n-1)$ pages) before the join can begin.

The second main cost of doing joins at Active Disks is the additional time required due to limited memory at the disks. This may require a join that can proceed in one pass at the host (using basic Hash-Join) to use multiple passes (using Hybrid Hash-Join) at the disks. The overall result can still be benefitial if the amount of savings (selectivity) in transfer of tuples from S outweighs the additional disk cost of writing and re-reading S tuples for a multi-pass join (recall that the additional disk cost is accumulated in parallel at all the disks, while the transfer of S tuples loads the single bottleneck host interface).

Data Parameters

$|S|$ = size of relation S (pages)

$|R|$ = size of relation R (pages)

p_{bytes} = size of page (bytes)

p_{tuples} = size of page (tuples)

Application Parameters

α_N = selectivity of join

w_{hash} = cycles to hash a record

w_{probe} = cycles to probe a record

w_{insert} = cycles to insert a record

Host Parameters

M = memory size of the host

r_d = disk raw read rate

r_w = disk raw write rate

s_{cpu} = CPU speed of the host

r_n = host network rate

Active Disk Parameters

m = memory size of the disk

s_{cpu}' = CPU speed of the disk

r_n' = active disk network rate

r_a = aggregate network fabric rate

The basic performance tradeoff is embodied in the equations below:

$$k = \frac{|R|}{M}$$

defines the number of passes that Hybrid Hash-Join will need to make on the server system, then:

$$t_{prepare} = max\left(\frac{|R|}{d \cdot r_d}, \frac{|R| \cdot p_{\text{tuples}} \cdot w_{\text{hash}}}{s_{\text{cpu}}} + \frac{\frac{|R|}{k} \cdot p_{\text{tuples}} \cdot w_{\text{insert}}}{s_{\text{cpu}}}, \frac{|R| \cdot \left(\frac{k-1}{k}\right)}{d \cdot r_w}, \frac{|R| + \quad |R| \cdot \left(\frac{k-1}{k}\right)}{r_n}\right)$$

to read relation R off the disks, transfer it to the host, and create a hash table of all the tuple of R that fit into the available memory (M), then transfer the balance of R back to the drives and write it out to the disks, then:

$$t_{first} = max\left(\frac{|S|}{d \cdot r_d}, \frac{|S| \cdot p_{\text{tuples}} \cdot w_{\text{hash}}}{s_{\text{cpu}}} + \frac{\frac{|S|}{k} \cdot p_{\text{tuples}} \cdot w_{\text{probe}}}{s_{\text{cpu}}}, \frac{|S| \cdot \left(\frac{k-1}{k}\right) + \frac{|S|}{k} \cdot \alpha_N}{d \cdot r_w}, \frac{|S| + |S| \cdot \left(\frac{k-1}{k}\right) + \frac{|S|}{k} \cdot \alpha_N}{r_n}\right)$$

to read relation S off the disks, transfer it to the host, probe the portion of R that currently resides in memory, send the balance of S (that cannot be probed until a later pass, $|S| \cdot \left(\frac{k-1}{k}\right)$) and the successfully matched tuples ($\frac{(|S|)}{k} \cdot \alpha_N$) back to the drives, and write them all to the disks, then subsequent passes in two parts as:

$$t_{moreprepare} = max\left(\frac{\frac{|R|}{k}}{d \cdot r_d}, \frac{\frac{|R|}{k} \cdot p_{\text{tuples}} \cdot w_{\text{insert}}}{s_{\text{cpu}}}, \frac{\frac{|R|}{k}}{r_n}\right)$$

to read a partition of R from the disks, transfer it to the host, and insert the tuples into a hash tabled, and process from S as:

$$t_{subsequent} = max\left(\frac{\frac{|S|}{k}}{d \cdot r_d}, \frac{\frac{|S|}{k} \cdot p_{\text{tuples}} \cdot w_{\text{probe}}}{s_{\text{cpu}}}, \frac{\frac{|S|}{k} \cdot \alpha_N}{d \cdot r_w}, \frac{\frac{|S|}{k} + \frac{|S|}{k} \cdot \alpha_N}{r_n}\right)$$

to read a partition of S from the disks, transfer it to the host, probe all the tuples in the hash table, transfer the matching tuples back to the drives, and write them to the disks. This gives an overall time of:

$$t = t_{prepare} + t_{first} + (k-1) \cdot (t_{moreprepare} + t_{subsequent})$$

and a throughput of:

$$\text{throughput} = \frac{(|R| + |S|) \cdot p_{\text{bytes}}}{t}$$

for the traditional server.

For the Active Disk system, we have a similar set of equations as:

$$k' = \frac{|R|}{m}$$

the number of passes required by the smaller Active Disk memory (m), then:

$$t_{prepare} = max\left(\frac{|R|}{d \cdot r_d}, \frac{|R| \cdot p_{\text{tuples}} \cdot w_{\text{hash}}}{s_{\text{cpu}}'} + \frac{\frac{|R|}{k'} \cdot p_{\text{tuples}} \cdot w_{\text{insert}}}{s_{\text{cpu}}'}, \frac{|R| \cdot \left(\frac{k'-1}{k'}\right)}{r_w}, \frac{|R|}{r_n}\right)$$

to read relation R from all the disks, broadcast the entire relation to all the disks, build an in-memory hash table for the portion of R that will fit in m ($(|R|)/k'$) and hash the rest into partitions, and write all the partitions except the current one ($|R| \cdot \left(\frac{k'-1}{k'}\right)$)to the disk locally, then:

$$t_{first} = max\left(\frac{\frac{|S|}{d}}{r_d}, \frac{\frac{|S|}{d} \cdot p_{\text{tuples}} \cdot w_{\text{hash}}}{s_{\text{cpu}}}, + \frac{\frac{\frac{|S|}{d}}{k'} \cdot p_{\text{tuples}} \cdot w_{\text{probe}}}{s_{\text{cpu}}}, \frac{\left(\frac{|S|}{d}\right) \cdot \left(\frac{k'-1}{k'}\right) + \frac{\left(\frac{|S|}{d}\right)}{k'} \cdot \alpha_N}{r_w}\right)$$

to read the local portion of relation S at each disk, hash the portion into partitions and probe the in-memory portion of R, and write the remaining partitions back to the disk locally, then several passes of:

$$t_{moreprepare} = max\left(\frac{\frac{\left(\frac{|R|}{d}\right)}{k'}}{r_d}, \frac{\frac{\left(\frac{|R|}{d}\right)}{k'} \cdot p_{\text{tuples}} \cdot w_{\text{insert}}}{s_{\text{cpu}}},\right)$$

to read a partition of relation R and insert the tuples into an in-memory hash table locally, then:

$$t_{subsequent} = max\left(\frac{\frac{\left(\frac{|S|}{d}\right)}{k'}}{r_d}, \frac{\frac{\left(\frac{|S|}{d}\right)}{k'} \cdot p_{\text{tuples}} \cdot w_{\text{probe}}}{s_{\text{cpu}}}, \frac{\frac{\left(\frac{|S|}{d}\right)}{k'} \cdot \alpha_N}{r_w}\right)$$

to read a partition of relation S, probe the hash table, and write the matching tuples back to the disk. Note that all passes of these two steps are done locally, after the initial broadcast of relation R to all the disks, processing is done strictly locally with each disk operating independently. All this gives an overall time of:

$$t_{active} = t_{prepare} + t_{first} + (k'-1) \cdot (t_{moreprepare} + t_{subsequent})$$

and a throughput of:

$$\text{throughput}_{\text{active}} = \frac{(|R| + |S|) \cdot p_{\text{bytes}}}{t_{active}}$$

and captures the more efficient processing at the disks.

In terms of the equations of Chapter 3, this is a k'-phase computation, with $t_{prepare} + t_{first}$ as the first phase, followed by a synchronization point, followed by some number of phases of $t_{moreprepare} + t_{subsequent}$. As described above, the first phase has a serial fractions equivalent to the time required to broadcast R to all of the disks. The subsequent phases can then be performed completely in parallel with $p = 1.0$.

Figure 4-11 shows the relative performance of a traditional system against an Active Disk system for two different join sizes. For a join where R and S are relatively close in size, the Active Disk algorithm will perform worse than the host-based algorithm. The disks will be required to use the Hybrid Hash-Join algorithm and make multiple passes across the data, whereas the host-based system can fit the entire hash table for R in memory and proceed in a single pass through S. The benefit of Active Disks becomes obvious when S is significantly larger than R. In this case, the savings in not transferring non-matching tuples of S on the network far outweigh even the repeated passes through S on the disk. Because each of the disks can operate in parallel and independently, the tuples of S that do not match any tuple in R will never leave the disk that they are initially read

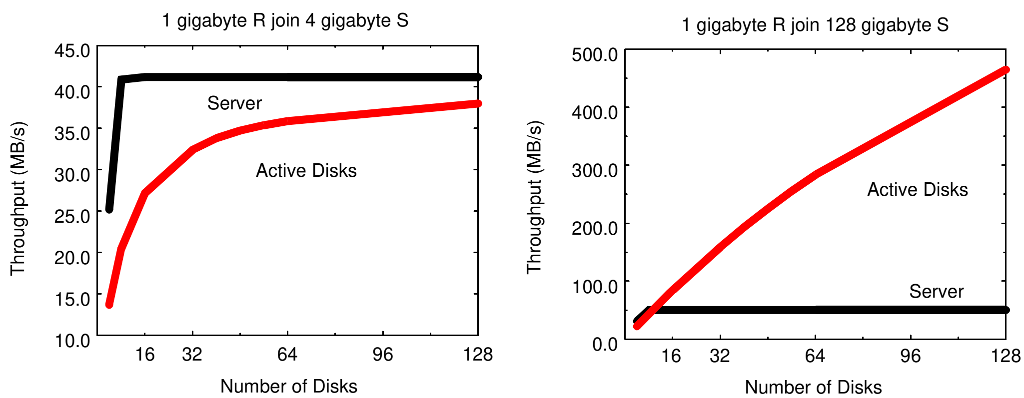


Figure 4-11 Comparison of server and Active Disk across varying sizes of R and S. The basic tradeoff is the amount of network transfer saved by not sending non-contributing tuples of S to the host in the first place vs. the additional cost of doing multiple passes through the data if the entire inner relation does not fit into the Active Disk memory. This downside of Active Disk join is eliminated by the use of a semi-join with Bloom filters as described below. These numbers are analytic results as calculated from the formulas provided in the preceding section.

from, whereas the server algorithm must move all tuples of S across the network to the host in order to perform the probes.

4.3.3.3 Semi-Join

The basic difficulty with performing the processing in this way is that each disk must maintain a complete copy of R in order to perform the join correctly (otherwise it will “miss” records that should be part of the result). This is a problem when R does not fit into the memory available at the individual drives, thereby requiring use of the multiple pass algorithm described above.

The total amount of memory required at each drive can be reduced by not retaining a copy of the entire R relation at the disks, but sending only the join keys necessary for determining whether a particular record from S should be included in the result or not. If the disks perform only this *semi-join* [Bernstein81] they achieve the full savings of the selectivity of the join, without requiring memory for all of R.

The semi-join is similar to the key-only option described in the last section for sorting. The join is performed in two phases. In the first phase, the keys of R are used to filter S and extract only the records that match the keys of R. These records from S are then used to probe R as in the normal hash-join, but with each record guaranteed to find a match in R. The algorithm requires significantly less memory than a full join in the first phase, because only the keys of R must be kept in memory. This makes it a particularly attractive algorithm for use with Active Disks. The disks can individually (and in parallel) perform a semi-join using only the keys of R and send the matching records of S to the host for the final join. The tuples returned must still be joined with R at the host before they are output, but these tuples would have been sent to the host anyway for output.

This “split” version of join also avoids the size explosion possible if a particular record from S matches multiple records in R. Performing only semi-join guarantees that the selectivity of the operation at the drive will not exceed 1.0 (i.e. it never *increases* the amount of data moved off the disk from the traditional case). In the case of a full join, the size of the resulting table can range from no tuples (in the extreme) to the cross product of both relations (assuming each tuple in S matches all the tuples in R, leading to $|S| * |R|$ tuples each of size $s + r$). Using the semi-join, the amount of data returned from the drives is never more than $|S|$, which is the amount of data that the traditional system must transfer in all cases.

The only downside is the additional work performed at the host in again probing R with all the successfully matched S tuples. This cost is mitigated by several factors: 1) the S tuple will be guaranteed to find a match in R, because it already matched a join key at one of the disks, 2) if the S tuple matches multiple tuples in R, only one of them will be a repeated test (since the disk will send the tuple after a single match, without probing for additional matches, thereby leaving this work to the host) and 3) the selectvity benefits of pre-computing at the drives.

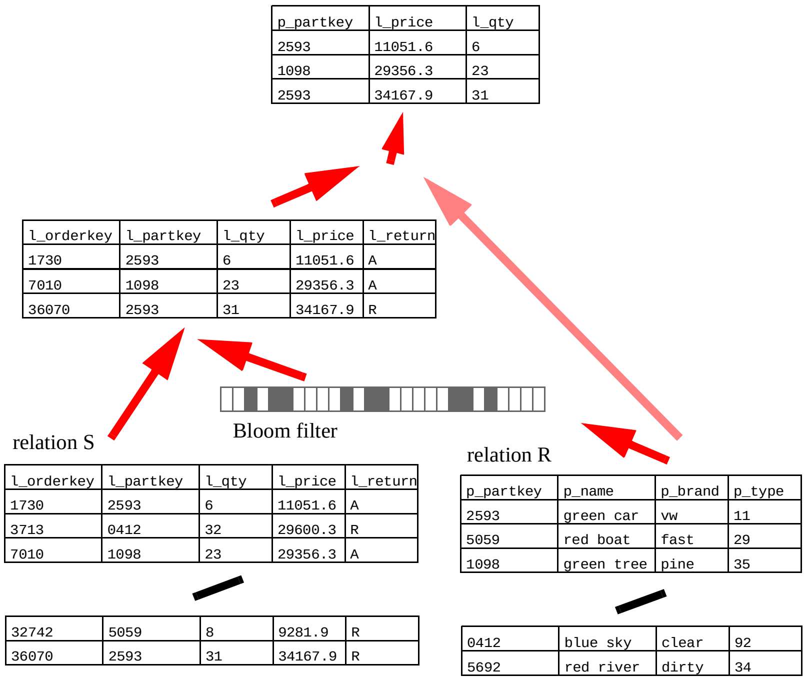


Figure 4-12 Illustration of the Bloom join algorithm. Keys from the inner relation (R) are used to form a hash-based bit-vector that is broadcast to all the disks. The disks use this bit vector to filter records from the outer relation (S) before records are returned to the host. The filter will return some number of "false positives" since the bit vector cannot represent all the keys exactly (multiple keys may hash to the same bit position), but it will always return all the necessary records from S. This provides most of the selectivity benefit of a highly selective join, while requiring only constant memory at the drives - the size of the Bloom filter can be chosen based on the memory available, rather than requiring memory proportional to the size of the relations in a particular query.

4.3.3.4 Bloom Join

If even the keys of R necessary for a semi-join exceed the memory capacity of the disks, a Bloom Join algorithm [Mackert86] can be used to perform the first phase of a semi-join at the drives. This algorithm uses a hash-based bit vector built up from the set of inner tuples at the host and copied to each of the drives to eliminate tuples from the outer relation before they are sent to the host, as illustrated in Figure 4-12.

The goal, as with semi-join, is to exclude tuples from S that will not find a match in R and therefore will not be part of the final result (non-contributing tuples). Instead of broadcasting all the distinct values of the join attribute from R to the drives, we create a bit vector b[*1...n*], initially set to all '0's. For each value of the join attribute in R, we hash it to a value in the range *1* to *n* and set the corresponding bit to '1'. We then use this bit vector when processing tuples from S. If we apply the same hash function to the join attribute in S, then any tuple for which the bit is set to '0' can be excluded from the result, since it will not match any tuples from R. This still allows some number of "false positives" from S to be sent back to the host, but it will give us the selectivity benefits of semi-join while using only a constant amount of memory. A Bloom filter reduces an arbitrarily large list of keys to a fixed-size bit vector. As a result, the memory required from doing a Bloom join at the drives is independent of the size of the relations and can often achieve large selectivity benefits with only a small amount of memory, as shown in Table 4-5 for a number of queries from the TPC-D benchmark.

More recent work in this area has proposed an alternative algorithm that encodes the keys in scan order, rather than using hashing, with some promising improvements in filter size and effective selectivity [Li95].

Query	Join	Size of Bloom filter						Keys	Table
		128 bits	1 kilobyte	8 kilobytes	64 kilobytes	1 megabyte	ideal	KB	MB
Q3	1.1	1.00	0.54	0.33	0.33	0.33	0.21	12.4	4.2
Q5	1.4	1.00	1.00	1.00	1.00	1.00	0.04	58.6	4.2
Q5	2.1	1.00	0.94	0.75	0.55	0.55	0.15	89.7	28.6
Q5	4.1	0.90	0.22	0.22	0.22	0.22	0.22	0.9	0.3
Q5	5.1	0.23	0.23	0.23	0.23	0.23	0.23	0.1	0.01
Q9	1.1	1.00	0.11	0.11	0.11	0.11	0.05	4.0	4.7
Q10	2.1			0.33	0.21	0.21	0.08	21.9	28.6

Table 4-5 Sizes and selectivities of joins using Bloom filters of a particular size. Note that these measurements are based on a particular choice of execution plan for each query, the sizes required for the different joins would be different if the join orders were changed (and the order might well be changed based on the choice of Active Disk function placement).

In order to take advantage of Bloom filters, join processing at the drives provides a semi-join function, rather than a full join as discussed above. The Bloom filter representing the keys is sent to all storage locations of the outer relation. The processing then returns all the tuples of the outer relation that may match - with false positives allowed - the inner relation. As in the semi-join, these tuples are then used to probe a full hash table of R at the host and the matching records are joined and output. This saves the transfer time of returning non-contributing tuples to the host, as well as the processing time required at the to look up and reject a non-contributing outer tuple. There will still be some non-contributing tuples that must be eliminated at the host, but the selectivity benefit should usually overcome this additional CPU work.

Using a Bloom filter at the Active Disks is particularly attractive because it has the selectivity benefits of semi-join mentioned above, but requires only a constant amount of memory, rather than depending on the size of the inner relation.

Chapter 5: Performance and Scalability

This chapter describes a prototype system and a performance evaluation of running the applications described in the previous chapter in an Active Disk setting. The first section details the experiments performed to illustrate the benefits and tradeoffs in using an Active Disk system compared to a traditional server with “dumb” disks. Performance is measured against a running prototype system using six-year-old workstations to emulate Active Disks. Some preliminary results using an embedded version of the prototype are also discussed.

The next section discusses the implementation of a relational database system on Active Disks and the performance of this system on a subset of the queries from a decision support benchmark. The intent of these two sections is to show that the benefits promised by the performance model of Chapter 3 are achieveable in a realistic system.

Finally, the last section discusses the balancing of Active Disk functions with an existing foreground workload and explores a mechanism whereby closer integration of application knowledge with on-drive scheduling can provide considerable performance gains. The intent of this section is to illustrate a particular class of optimizations that are possible only when application-level knowledge is combined with scheduling information at the individual disks.

5.1 Prototype and Experimental Setup

The testbed used for all the experiments consists of ten prototype Active Disks, each one a six-year-old Digital Alpha AXP 3000/400 (133 MHz, 64 MB, Digital UNIX 3.2g) with two 2.0 GB Seagate ST52160 Medalist disks. For the server case, a single Digital AlphaServer 500/500 (500 MHz, 256 MB, Digital UNIX 3.2g) with four 4.5 GB Seagate ST34501W Cheetah disks on two Ultra-Wide SCSI busses is used[1]. All these machines are connected by an Ethernet switch and a 155 Mb/s OC-3 ATM switch. This setup is illustrated in Figure 5-1 showing the details of both systems.

1. note that these four disks on two busses represents more bandwidth than the single CPU server can handle when performing sequential accesses, so adding additional disks would give no benefit for sequential bandwidth.

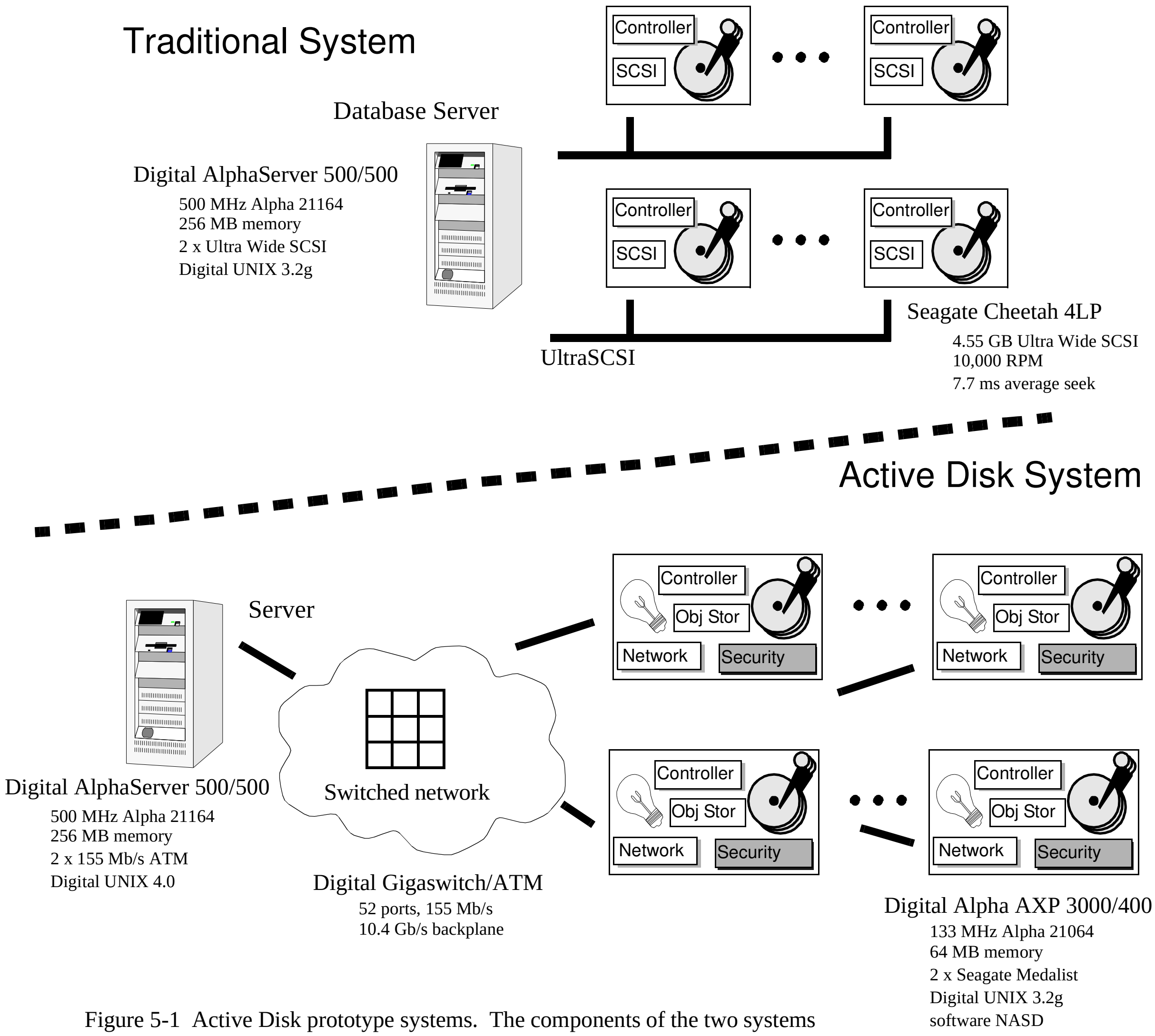


Figure 5-1 Active Disk prototype systems. The components of the two systems compared in the prototype numbers to follow. The diagram shows a traditional server system with directly-attached SCSI disks. The lower picture shows an Active Disk system using network-attached disks.

The experiments compare the performance of the single server machine with directly-attached SCSI disks against the same machine with network-attached Active Disks, each of which is a workstation with two directly-attached SCSI disks[1]. All the results reported give the throughput (MB/s) of both systems, and the amount of data processed is scaled with the number of disks used. The results will show dramatic improvements with Active Disks and confirm the intuition given by the model of Chapter 3.

1. the need to use two actual disks on each single “active disk” is an artifact of using old workstations not explicitly designed for this purpose. The 3000/400 contains two narrow SCSI busses, with a maximum bandwidth of 5 MB/s each. The Seagate Medalist disks used are capable of 7 MB/s each, but the use of the narrow SCSI busses limits sequential throughput to a total of 7 MB/s when used in combination, as shown in the Microbenchmarks section. The text will clearly identify any results where the use of two disks instead of a single, faster disk would impact the system performance and affect the comparison.

5.1.1 Microbenchmarks

This section presents basic measurements from the systems under test, to give an idea of the underlying performance characteristics of each. The two critical parameters are the disk bandwidth available to read bytes from the media and the "network" bandwidth to move these bytes from the disk to the host. In the traditional server system, the disks are simple disk drives using a SCSI interconnect. In the Active Disk system, the disks are old workstations and use an ATM interconnect to the host, as shown in Figure 5-1.

5.1.1.1 Disk Bandwidth

The experiments in this section measure the total raw disk bandwidth available from the two systems. Table 5-1 shows the performance of the disks for a single Active Disk

Drive	Disks	Read (MB/s)	CPU Idle (%)	Configuration
Active Disk	2	6.5	-	two Medalist drives
Cheetah	1	17.0	-	
Cheetah	2	26.9	60.1	
Cheetah	4	42.9	7.3	

Table 5-1 Performance of the disks in the prototype. The table shows the performance and the overhead of reading from the raw disks on the prototype drive and host.

and for the prototype host with a varying number of attached disks. The host disks perform signficantly better than the those on the prototype "disk" because they are two generations newer than those used in the individual Active Disks. This will make any comparison on raw disk bandwidth in the subsequent sections pessimistic to the Active Disk system.

5.1.1.2 Interconnect Bandwidth

The second primary factor for comparison among the two systems is the network bandwidth available between the Active Disks and the host. Table 5-2 shows the perfor-

Drives	Read from drive			Write to drive			Configuration
	Throughput (MB/s)	Drive Idle (%)	Host Idle (%)	Throughput (MB/s)	Drive Idle (%)	Host Idle (%)	
1	10.4	13	-	9.4	6	-	UDP, 256K buf, 32K frag
1	11.9	21	-	-	-	-	UDP, 256K buf, 32K frag, no checksum
1	11.4	12	-	10.4	10	-	TCP, 256 K buf, 32K frag
2	15.1	39	-	15.3	40	-	TCP, 256K buf, 32K frag
4	24.7	54	5	30.3	39	30	TCP, 256K buf, 32K frag

Table 5-2 Performance of network processing in the prototype. The table shows the performance and the overhead of network processing on the prototype drive and host for large requests. We see that TCP performs slightly better due to a superior flow control mechanism. We also see that turning off the checksum in UDP gives a significant reduction in processor utilization. It was not possible to easily turn off TCP checksumming in the prototype (to simulate a hardware-assisted checksum calculation, or a reliable network fabric, for example), but we would expect this to lower the overhead of the TCP processing as it does for UDP.

mance of the network for a single Active Disk and the host using ATM. This chart shows that there is a great deal of inefficiency in using a general-purpose network stack for storage traffic, which is again pessimistic to the performance of the Active Disk prototype. The CPU utilization of transferring data on the ATM network is significantly higher than an equivalent level of performance in the SCSI interconnect "network" used in the host system. This means that, for example, in the case of a four disk system, only a bit more than half of the processing power on the drive is available for Active Disk processing, the balance of the processor is busy sending data. The "network" for the host system is simply the SCSI bus, so the values in Table 5-1 also give the interconnect bandwidth for the traditional system.

5.2 Results

This section gives the results from running experiments with each of the applications described in Chapter 4. Each section provides the results for a single application, comparing the server system with traditional "dumb" disks and the same system with an equal number of Active Disks. All the experiments show the scaling of performance as the number of disks increases from one to ten, the maximum number of Active Disks available in the prototype setup.

5.2.1 Data Mining - Nearest Neighbor Search

Figure 5-2a compares the performance of the server system against the same system with Active Disks for the nearest-neighbor search. As predicted by the model of Chapter 3, we see that for a small number of disks, the server system performs better. The server processor ($s_{cpu} = 500$) is four times as powerful as a single Active Disk processor ($s_{cpu}' = 133$) and can perform the computation at full disk rate. The server system CPU saturates at 25.7 MB/s with two disks and performance does not improve as additional disks are added, while the Active Disks system continues to scale linearly to 58.0 MB/s with 10 disks. The prototype system was limited to ten Active Disks by the amount of hardware available, and four traditional disks by the length limitations of the Ultra SCSI bus, but extrapolating the data from the prototype to a larger system with 60 disks, the smallest of

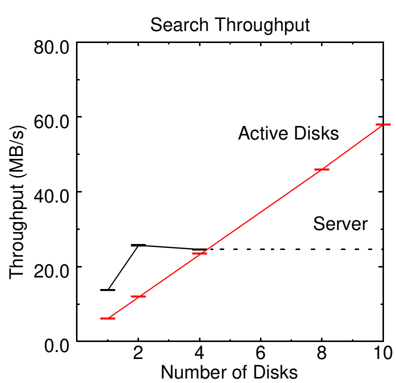
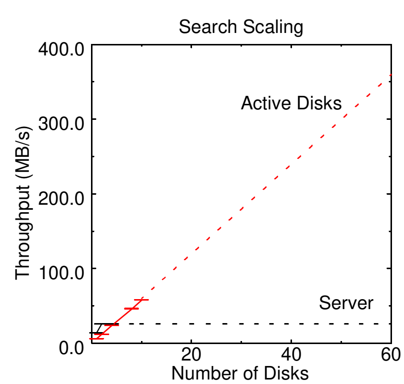


Figure 5-2a Performance of search. The search application shows linear scaling with number of disks up to 58 MB/s, while the server system bottlenecks at 26 MB/s.

Figure 5-2b Scaling of search performance. Because of the high selectivity of this search, we would not expect the Active Disks system to saturate for at least a few hundred disks.



the real systems introduced in Table 2-1, we would expect throughput, as shown in Figure 5-2b, near the 360 MB/s that the model predicts for this configuration.

The details of the computation in the search application are shown in Table 5-3.

Application	Input	Computation (instr/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Bandwidth (KB/s)
Search	k=10	7	28.6	72	80,500	0.4

Table 5-3 Costs of the search application. Computation requirement, memory required, and the selectivity factor in the network. The parameter value is a variable input to the application and specifies the number of neighbors to search for, which directly determines the memory size required at each disk.

5.2.2 Data Mining - Frequent Sets

In Figure 5-3a, we see the results for the first two passes of the frequent sets application (the *1-itemsets* and *2-itemsets*). We again see the crossover point at four drives, where the server system bottlenecks at 8.4 MB/s and performance no longer improves, while the Active Disks system continues to scale linearly to 18.9 MB/s. Figure 5-3b illustrates an important property of the frequent sets application that affects whether or not a particular analysis is appropriate for running on Active Disks. The chart shows the memory requirements across a range of input support values on two different data sets. The lower a support value, the more itemsets are generated in successive phases of the algorithm and the larger the state that must be held on disk. We expect that the support will tend toward the higher values since it is difficult to deal with a large number of rules, and the lower the support, the less compelling the generated rules will be. For very low values of the support, though, the limited memory at Active Disk may become an issue. Modern disk drives today contain between 1 MB and 4 MB of cache memory, and we might expect 16 to 64 MB in the timeframe in which Active Disks would become commercially available [Anderson98]. This means that care must be taken in designing algorithms and in choosing when to take advantage of execution at the disks. The details of the basic computation in the frequent sets application are shown in Table 5-4.

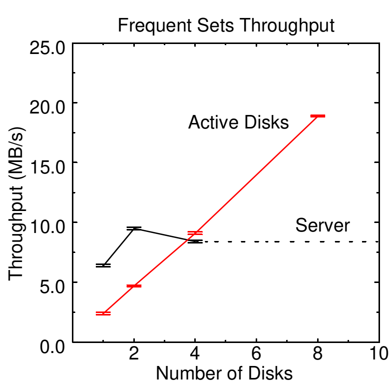
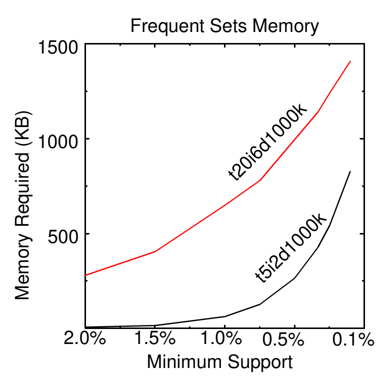


Figure 5-3a Performance of frequent sets. The frequent sets application shows linear scaling to 18.9 MB/s with eight Active Disks, while the server system bottlenecks at 8.4 MB/s.

Figure 5-3b Memory required for frequent sets. The amount of memory necessary for the frequent sets application increases as the level of support required for a particular rule decreases. Very low support values may require multiple megabytes of memory at each disk.



Application	Input	Computation (instr/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Bandwidth (KB/s)
Frequent Sets	s=0.25%	16	12.5	620	15,000	0.8

Table 5-4 Costs of the frequent sets application. Computation requirement, memory required, and the selectivity factor in the network. The parameter value specifies the minimum support required for an itemset to be included in the final count, which affects the amount of memory required at each drive.

5.2.3 Multimedia - Edge Detection

Figure 5-4 shows the results for the edge detection application. As we see in Table 5-5, the image processing applications require much more CPU time than search or

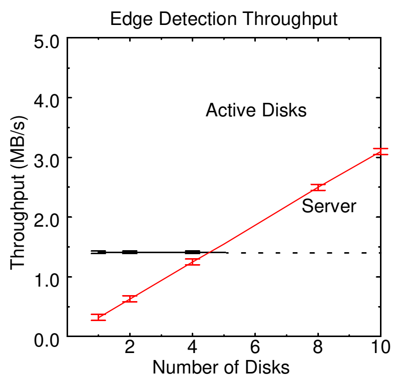


Figure 5-4 Performance of edge detection. The edge detection application shows linear scaling with number of disks while the server system bottlenecks at about 1.4 MB/s.

frequent sets do, leading to much lower throughputs on both systems. The edge detection bottlenecks the server CPU at 1.4 MB/s, while the Active Disk system scales to 3.2 MB/s with 10 disks. The brightness threshold provided as an input to the application determines how many objects are identified in a particular scene by setting a threshold for the contrast between nearby pixels to determine an “edge”. The setting shown is appropriate for properly identifying the cows (as well as a small number of rocks) in the sample images.

Application	Input	Computation (instr/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Bandwidth (KB/s)
Edge Detection	t=75	303	0.67	1776	110	6.1

Table 5-5 Costs of the edge detection application. Computation requirement, memory required, and the selectivity factor in the network. The parameter value is the brightness threshold of the objects to be detected, and affects the selectivity of the overall computation.

5.2.4 Multimedia - Image Registration

Figure 5-4 shows the results for the image registration application. Image registration is the most CPU-intensive of the applications we have considered, as shown in Table 5-6. It achieves only 225 KB/s on the server system, and scales to 650 KB/s with ten

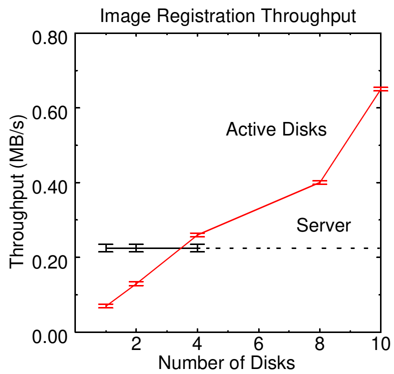


Figure 5-5 Performance of image registration. The image registration application also scales linearly, but requires almost a factor of ten more CPU cycles, reducing throughput in both systems.

Active Disks. Due to the iterative nature of this computation, the amount of processing required can vary significantly from image to image, a factor that the Active Disk runtime system would have to take into account when scheduling this particular computation.

Application	Input	Computation (instr/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Bandwidth (KB/s)
Image Registration	-	4740	0.04	672	180	0.2

Table 5-6 Costs of the image registration application. Computation requirement, memory required, and the selectivity factor in the network. The amount of computation required is highly dependent on the image being processed and the value shown is for an average image.

5.2.5 Database - Select (subset of Query 1)

Figure 5-6 compares the performance of a database server with traditional disks against a server with an equivalent number of Active Disks for a simple select query. The query being performed is:

```
select * from lineitem where l_shipdate > '1998-09-02'
```

using tables and test data from the TPC-D decision support benchmark. The records in the database cover dates from 1992 through the end of 1998, so this query returns about 4% of

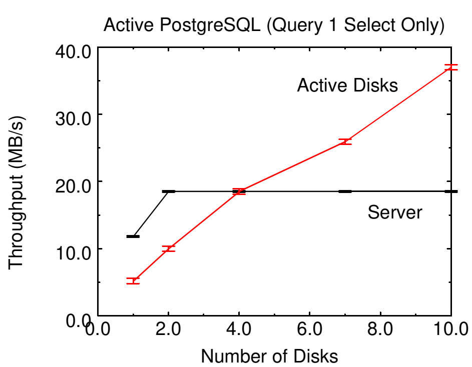


Figure 5-6 Performance of PostgreSQL select. The PostgreSQL select operation shows linear scaling with number of disks up to 25.5 MB/s with 7 disks, while the server system bottlenecks at 18 MB/s.

the total records in the **`lineitem`** table. This query performs the qualification at the disks and returns a record to the host only if the condition matches.

As usual, the server performs better than the Active Disk system for small numbers of disks, since each individual disk is much less powerful than the 500 MHz host. Once the aggregate compute power of the disks passes that of the host, the Active Disk system continues to scale while the server performance remains flat, no matter how much aggregate disk bandwidth is available. Notice that the performance increase in the Active Disk system is somewhat less than linear. This is due to the sequential overhead of performing the query - primarily the startup overhead of initiating the query and beginning the Active Disk processing. This overhead is amortized over the entire size of the table processed. For the experiments in the chart, the table is only 125 MB in size, so the overhead is significant and noticable in the results. A real TPC-D system sized for a 300 GB benchmark, would have a **`lineitem`** table of over 100 GB [TPC98].

The code executed at the host is a version of the PostgreSQL 6.5 modified to handle Active Disks. Changes were made in the storage layer to provide striping and use a NASD interface for disk access, rather than a traditional fileystem, and in the “scan” function to provide a way to ship the qualifcation condition to the drives and start the Active Disk processing. Additional changes to support aggregations and joins are discussed in a later section, as are the changes to allow the query optimizer to make decisions on the most appropriate location to execute a particular part of the query. Further details of the code modifications necessary to support Active Disks are provided in Chapter 6.

The details of the basic computation in the select are shown in Table 5-7. We see that the select is the least expensive of all the applications discussed so far, using less than four instructions per byte of data processed. It also uses very little memory since only enough memory to evaluate one page of tuples at a time is required.

Application	**Computation** (instr/byte)	**Computation** (cycles/byte)	**Throughput** (MB/s)	**Memory** (KB)	**Selectivity** (factor)	**Code** (KB)
Database Select	3.75	6.5	19.5	88	52.0	20.5 (13.3)

Table 5-7 Costs of the database select application. Computation requirement, memory required, and the selectivity factor in the network. The computation requirement is shown in both instructions per byte and cycles per byte. The last column also gives the total size of the code executed at the drives (and the total size of the code that is executed more than once).

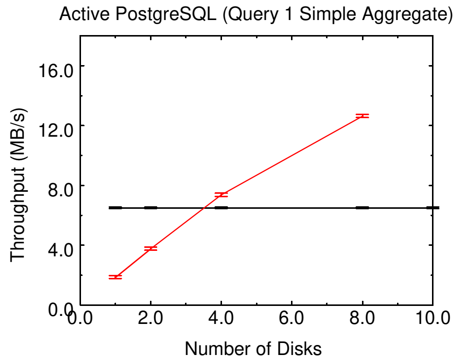


Figure 5-7 Performance of PostgreSQL aggregation. The PostgreSQL aggregation shows linear scaling with the number of Active Disks and reaches 13 MB/s with eight disks, while the server bottlenecks on the CPU at 6.5 MB/s.

5.2.6 Database - Aggregation (Query 1)

Figure 5-7 compares the performance of a database server with traditional disks against a server with an equivalent number of Active Disks for a simple aggregation query. The query being performed is:

```
select l_returnflag, l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from lineitem
where l_shipdate <= '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

using tables and test data from the TPC-D decision support benchmark. The records in the database cover dates from 1992 through the end of 1998, so this query summarizes about 95% of the records in the `lineitem` table. This query performs the qualification at the disks and examines a record only if the condition matches. It also performs the aggregation calculations in parallel at the disks, and returns per-disk summaries that are then aggregated into a single set of results at the host.

The details of the basic computation in the aggregation are shown in Table 5-8. The computation required for aggregation is significantly more than for the select. The same comparison as in the select is performed to identify records that match the qualification condition. Matching records are then sorted and combined using the `group by` keys and aggregated into the sums and averages specified by the query. Each disk returns the aggregation values for its portion of the relation, and these results are then combined at the host.

Application	Computation (instr/byte)	Computation (cycles/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Code (KB)
Database Aggregation	15.0	31.1		120	31.9	26.7 (18.4)

Table 5-8 Costs of the database aggregation application. Computation requirement, memory required, and the selectivity factor in the network. The computation requirement is shown in both instructions per byte and cycles per byte. The last column also gives the total size of the code executed at the drives (and the total size of the code that is executed more than once).

5.2.7 Database - Join (Query 9)

Figure 5-9 compares the performance of a database server with traditional disks against a server with an equivalent number of Active Disks for a simple two-way join. The query being performed is:

```
select sum(l_quantity), count(*)
from part, lineitem
where p_partkey = l_partkey
and p_name like '%green%'
group by n_name, t_year
order by n_name, t_year desc
```

using tables and test data from the TPC-D decision support benchmark. The records in the database cover 1,000 different items, so this query matches about 10% of the unique part numbers in the database. This query performs a semijoin at the disks and returns a record to the host only if the join key matches the filter created from the inner relation. The returned records from the `part` table are then used to probe the `lineitem` tables.

The details of the basic computation in the join are shown in Table 5-9. The operation performed at the drives is the filtering phase of a Bloomjoin as discussed in Chapter 4. The drives perform only the initial filtering based on keys of the inner relation, essentially performing a semijoin of the outer relation with the keys from the inner relation. Only matching tuples from the outer relation are returned to the host, where they are joined with the necessary fields from the inner relation.

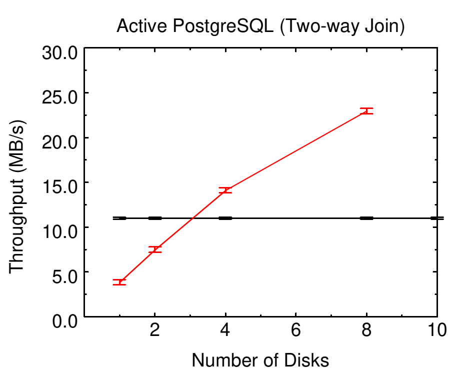


Figure 5-8 Performance of PostgreSQL join. The PostgreSQL join scales nearly linearly to 24 MB/s with Active Disks, and is limited to 11 MB/s in the server system.

Application	Computation (instr/byte)	Computation (cycles/byte)	Throughput (MB/s)	Memory (KB)	Selectivity (factor)	Code (KB)
Database Join	3.4	6.2	20.0	88	4.3	19.8 (14.4)

Table 5-9 Costs of the database join application. Computation requirement, memory required, and the selectivity factor in the network. The computation requirement is shown in both instructions per byte and cycles per byte. The last column also gives the total size of the code executed at the drives (and the total size of the code that is executed more than once).

The results in Figure 5-8 show the performance of a more complex join, executing the full 5-way join given by Query 9 from TPC-D. The query being performed is:

```
select n_name, t_year,
sum(l_extprice*(1-l_disc)-ps_supplycost*l_quantity) as sum_profit
from part, supplier, lineitem, partsupp, order, nation, time
where s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and t_alpha = o_orderdate
and s_nationkey = n_nationkey
and p_name like '%green%'
group by n_name, t_year
order by n_name, t_year desc
```

again using tables and test data from TPC-D.

This query has a much higher serial fraction than the two-way join, and shows the performance limitation in the Active Disk much sooner than the simple join. The serial fraction of this entire query is close to 30%, so the maximum speedup possible with Active Disks is a factor of 3x, even with perfect parallel scaling. The results here show a 11% improvement in performance with a total of eight disks.

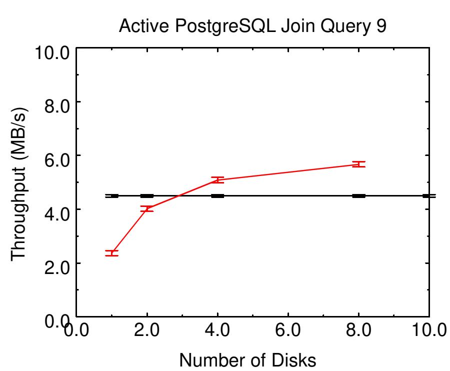


Figure 5-9 Performance of PostgreSQL join. This query has a significantly higher serial fraction than the previous applications, so the scaling with Active Disks drops off relatively early. The performance improvement is about 11% with eight disks.

5.2.8 Database - Summary

Table 5-10 summarizes the results of the last several sections and compares the performance of the server system and the Active Disk prototype on several of the most expensive queries from the TPC-D benchmark. We see that the scan-intensive applications

Query	Type	Input (MB)	Output (KB)	Disks	Host (s)	Throughput (MB/s)	Active Disk (s)	Throughput (MB/s)	
Q1	scan	494	0.2	8	76.0	6.5	38.0	12.6	100%
Q5	join (6)	494	0.1	8	219.0	2.2	186.5	2.6	17%
Q6	select	494	5057	8	27.2	18.8	17.0	29.0	60%
Q9	join (6)	494	0.5	8	95.0	4.5	85.3	5.78	11%

Table 5-10 Summary of TPC-D results using PostgreSQL. The table compares the performance of a selected set of queries from the TPC-D benchmark running on the PostgreSQL database system using a single host, and in a system modified to use Active Disks.

show roughly linear scalability, while the more complex join operations have considerably higher serial overheads, but still show significant speedups with Active Disk processing. These results are with a small prototype system of only eight disks, which is much smaller than the system that are built in practice for this type of workload.

5.2.9 Database - Extrapolation

Table 5-11 extends the results from the previous table to estimate the performance of a more realistically sized TPC-D system compared to a similar system using Active Disks. The system modelled is a Digital AlphaServer 8400 with 520 disks running

Query	Type	Input (GB)	Output (KB)	Disks	Host (s)	Throughput (MB/s)	Active Disk (s)	Throughput (MB/s)	
Q1	scan	192.3	0.2	520	4,357.1	45	*307.7*	*640*	*1,320%*
Q5	join (6)	245.1	0.1	520	1,988.2	126	*1,803.4*	*139*	*10%*
Q6	select	27.5	0.1	520	63.1	446	*6.1*	*4,636*	*900%*
Q9	join (6)	279.2	6.5	520	2,710.8	105	*2,232.1*	*128*	*22%*

Table 5-11 Extension of TPC-D results to a larger system. This table extends the results of the summary table in the previous section to a larger system with a total of 520 disks. The system is the Digital AlphaServer 8400 originally presented in Chapter 2, with performance numbers from a TPC-D 300 GB benchmark reported in May 1998 [TPC98a]. The numbers in this table predict the performance benefit of replacing the disks in that system with Active Disks and using an appropriately modified database system. The change in cost assumes that each Active Disk costs twice as much as a traditional disk, which is a very conservative estimate.

Oracle 8 on a TPC-D benchmark with a scale factor of 300 GB [TPC98a]. The numbers are estimates for this larger system based on the results for the eight disk prototype presented in the previous section. The table shows the improvements on four of the most expensive queries in the benchmark. We see that better than order of magnitude improvements are possible in the scan-intensive queries with less dramatic, but still significant, benefits for even the most complex join operations.

5.3 Model Validation

This section compares the performance results measured in the prototype system against the predictions made by the model of Chapter 3 and finds generally good agreement across all the applications measured.

5.3.1 Data Mining & Multimedia

The graphs of Figure 5-2, 3, 4, and 5 match the basic shape of the model predictions in Chapter 4. To confirm the values, we need the specific parameters of this testbed. We have $\alpha_s = 133/500 = 1/3.8$ for relative processing rates between the Active Disks and the host, the host processor is about four times as powerful as a single Active Disk processor.[1] Ideally, the prototype would have $\alpha_d = \alpha_n = 1$ for these tests, with disks and networks being equal, but this was not possible with the testbed hardware available. Instead, the parameters are $r_d = 14\ \mathrm{MB/s}$, $r_d' = 7.5\ \mathrm{MB/s}$, $r_n = 60\ \mathrm{MB/s}$ and $r_n' = 10\ \mathrm{MB/s}$ for the host and Active Disks respectively.

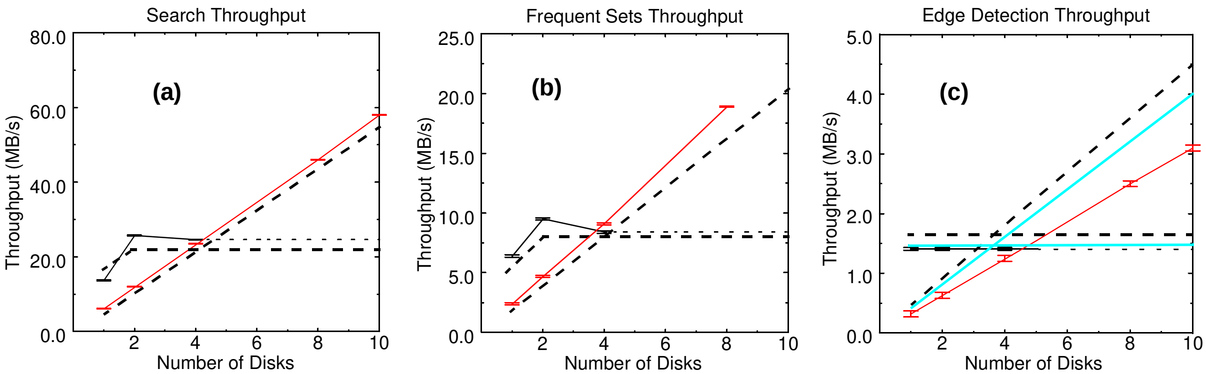
Estimating the applications' selectivity is a straightforward exercise of counting bytes and these are shown in Table 5-12. Estimating the number of cycles per byte is not so straightforward. The analysis began by instrumenting the server implementation of each application to determine the total number of cycles spent for the entire computation when all code and data are locally cached, and dividing this by the total number of bytes processed. This ignores the cost of forming, issuing and completing the physical SCSI disk operations, measured in a previous study as 0.58 microseconds on a 150 MHz Alpha or 10.6 cycles per byte [Patterson95]. Adding this to the "hot cache" numbers gives an estimate of the cycles per byte required by each application in Table 5-12.

Figure 5-10 combines the results for all four applications and superimposes the predictions of the model based on these system and application parameters. The search and frequent sets applications show strong agreement between the model and the measure-

Application	Computation (cycles/byte)	Memory (KB)	Selectivity	Parameter
Search	23.1	72	80,500	k=10
Frequent Sets	61.1	620	15,000	s=0.25%
Edge Detection	288	1776	110	t=75
Image Registration	1495	672	150	-

Table 5-12 Parameters of the applications for validation of the analytic model. The table gives computation time per byte of data, memory required at each Active Disk, and the selectivity factor in the network.

1. this ratio can be estimated directly from the clock rates because the processors use the same basic chip, and the code is identical for both cases. Normally one would need to compare the relative performance on the particular benchmark application - e.g. the embedded processor on the disk may be less adept at some functions than a more complex, superscalar host processor, although the simple integer computations used as the basis for most of the applications discussed here should perform nearly as well on the embedded processor



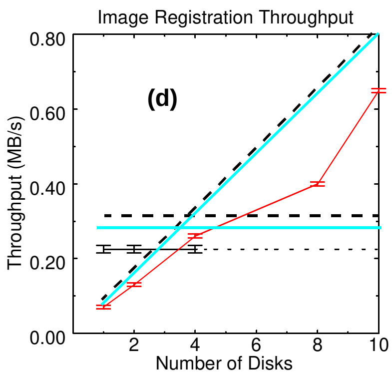


Figure 5-10 Validation of the analytical model against the prototype. The values predicted by the model using the system and application parameters in Section 5.4 are superimposed as dashed lines on the measurements from the prototype systems. The differences are within 15% for search and frequent sets. Edge detection and image registration do not precisely fit the assumptions of our model. Both applications suffer additional, unmodelled I/O stall time because they read from the disks in image-sized chunks, rather than streaming in sequential accesses as search and frequent sets are able to do. This means that there is disk access time that cannot be overlapped with computation, reducing the throughput. There is serial overhead in all of the applications. This is largely amortized by the large size of the databases used in search and frequent sets, but shows up in the image processing applications that process less total data. The gray lines on the figures estimate the expected performance when these factors are accounted for. This estimated performance is now within 15%.

ments. The largest error, a 14% disagreement between the server model and implementation of the search may reflect an overestimate of the cycles per byte devoted to disk processing because the estimate is based on an older machine with a less aggressive superscalar processor. The other two applications, however, differ significantly from the model predictions. The problem with these applications is that they do not yet overlap all disk accesses with computation, as the model assumes. For example, the edge detection application reads 256 KB images as a single request and, since the operating system read-ahead is not deep enough, causes additional stall time as each image is fetched. Using asynchronous requests or more aggressive prefetching in the application should correct this inefficiency. An additional contributor to this error is the serial portion of the applications which affects the image processing applications more seriously since they process less total data than the other two. To estimate the performance of these applications if the overlapping were improved, the model validation results estimate the total stall time experienced by each application and subtract it from the application run time. These "improved" prototype estimates are shown as additional lines in Figure 5-10c and d. With this modification, the model predicts performance within 15% for all the applications shown. Given the goal of using the model to develop intuition about the performance of Active Disks applications, these are strong results.

5.3.2 Database

The results in Figure 5-11 show the validation of the model for the three basic database operations. The chart shows very close agreement between the performance predicted by the model and that observed in the prototype. For the select, the primary limitation is the interconnect bandwidth into the host. The traditional system is faster at a low number of disks because the SCSI busses can deliver the aggregate bandwidth of the disks. Once this limit is exceeded, the server system no longer improves as additional disks are added. The Active Disk system starts out lower because of the mismatch in the underlying physical bandwidths in the prototype ($r_d \neq r_d'$) but quickly overtakes the host due to the much smaller amount of data being transferred. The aggregation is cpu-limited, so the host system bottlenecks immediately as its processor is completely occupied. Once the aggregate processing power of the Active Disks exceeds that of the host, the Active Disks are faster and continue to scale as disks are added. The selectivity of this computation is over 600, so the Active Disk system would not bottleneck on the interconnect until well over 100 disks. The two-way join shows a much lower selectivity - only 8 for the chosen subjoin from Query 9 - so even the Active Disk system will bottleneck at a certain point. In this case, we can see the plateau already with 16 disks, although this still repre-

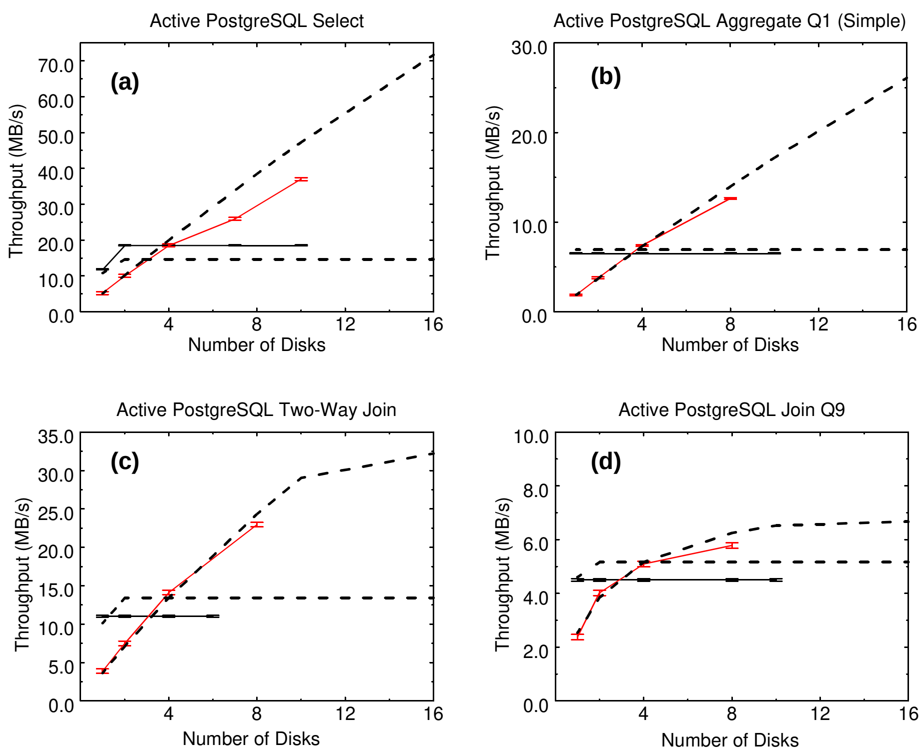


Figure 5-11 Validation of the model against the database operations. The values predicted by the model using the system and application parameters in Section 5.4 are superimposed as dashed lines on the measurements from the prototype systems.

sents a factor of 4x performance improvement over the host at the same point. Finally, the power of Amdahl's Law is shown for the full Query 9 join - the most complex of the TPC-D queries. This computation, the way it is partitioned, has a serial fraction of 30%, which causes the Active Disk system to quickly plateau. There are more complicated ways to partitioned this computation which might improve the Active Disk performance, but the gain would likely never be more than 2x the current performance.

5.4 Extension - Data Mining and OLTP

One of the major benefits of performing processing directly at the drives is the ability to more efficiently schedule application work. This is particularly promising in the ability to balance the application processing done by the Active Disks with the existing "demand" work at the drive.

The previous sections have focussed on the direct speedups of applications running in an Active Disk system. It is possible, however, to use the additional resources provided by Active Disks in another way: to perform additional work at the drives without impacting the applications already using the storage system. This can improve total system thoughput in an existing system, or reduce the cost of a new system that handles an equivalent workload.

This section explores one example of this type of scheduling, where an application operating directly at the drive takes advantage of a particular foreground workload to aid its own processing by doing additional work in the "background" using resources that would otherwise be wasted.

5.4.1 Idleness in OLTP

Query processing in a database system requires several resources, including 1) memory, 2) processor cycles, 3) interconnect bandwidth, and 4) disk bandwidth. Performing additional tasks, such as for example data mining, on a transaction processing system without impacting the existing workload would require there to be "idle" resources in each of these four categories.

Active Disks provide the additional memory and compute resources that are not utilized by the existing transaction processing workload. Using Active Disks to perform scans, aggregations, and joins directly at the drives keeps the interconnect requirements low. This leaves the disk arm and media as the critical resources. The following sections discuss a scheduling scheme at the disks that allows a background sequential workload to be satisfied essentially for free while servicing foreground requests.

The discussion starts with a simple priority-based scheme that allows the background workload to proceed with only a small impact on the foreground work. The following sections extend this system to read additional blocks completely "for free" and show that these benefits are consistent at high foreground transaction loads and as data is striped over a larger number of disks.

5.4.2 Motivation

The use of data mining to elicit patterns from large databases is becoming increasingly popular over a wide range of application domains and datasets [Fayyad98]. One of the major obstacles to starting a data mining project within an organization is the high initial cost of purchasing the necessary hardware. This means that someone must "take a chance" on the up front investment simply on the suspicion that there may be interesting "nuggets" to be mined from the organizations existing databases. Many data mining operations translate into large sequential scans of the entire data set. If these selective, parallel scans can be performed directly at the individual disks, then the only limiting factor will be the bandwidth available for reading data from the disk media (i.e. the application performance will scale directly with the number of disks available).

The most common strategy for data mining on a set of transaction data is to purchase a second database system, copy the transaction records from the OLTP system to the decision support system each evening, and perform mining tasks only on the second system, i.e. to use a "data warehouse" separate from the production system. This strategy not only requires the expense of a second system, but requires the management cost of maintaining two complete copies of the data. Table 5-13 compares a transaction system and a decision support system from the same manufacturer. The decision support system contains a larger amount of compute power, and higher aggregate I/O bandwidth, even for a significantly smaller amount of live data. This section argues that the ability to operate close to the disk makes it possible for a significant amount of data mining to be performed using the transaction processing system, without requiring a second system at all. This provides an effective way for an organization to "bootstrap" its mining activities.

system	# of CPUs	memory (GB)	# of disks	storage (GB)	live data (GB)	cost ($)
NCR WorldMark 4400 (TPC-C)	4	4	203	1,822	1,400	$839,284
NCR TeraData 5120 (TPC-D 300)	104	26	624	2,690	300	$12,269,156

Table 5-13 Comparison of an OLTP and a DSS system from the same vendor. We see that the DSS system requires much greater processing power and bandwidth than the OLTP system. Data from *www.tpc.org*, May and June 1998.

The idea of this section is to use Active Disks to support an existing transaction processing workload as well as a background data mining workload. Previous sections have argued that the memory and processing power is available on the disks to perform additional functions. This section will show that, for a particular choice of workloads, two applications can be combined and can take advantage of the ability to schedule directly at the disk drives to operate more efficiently together. Figure 5-12 illustrates the architecture of such a system.

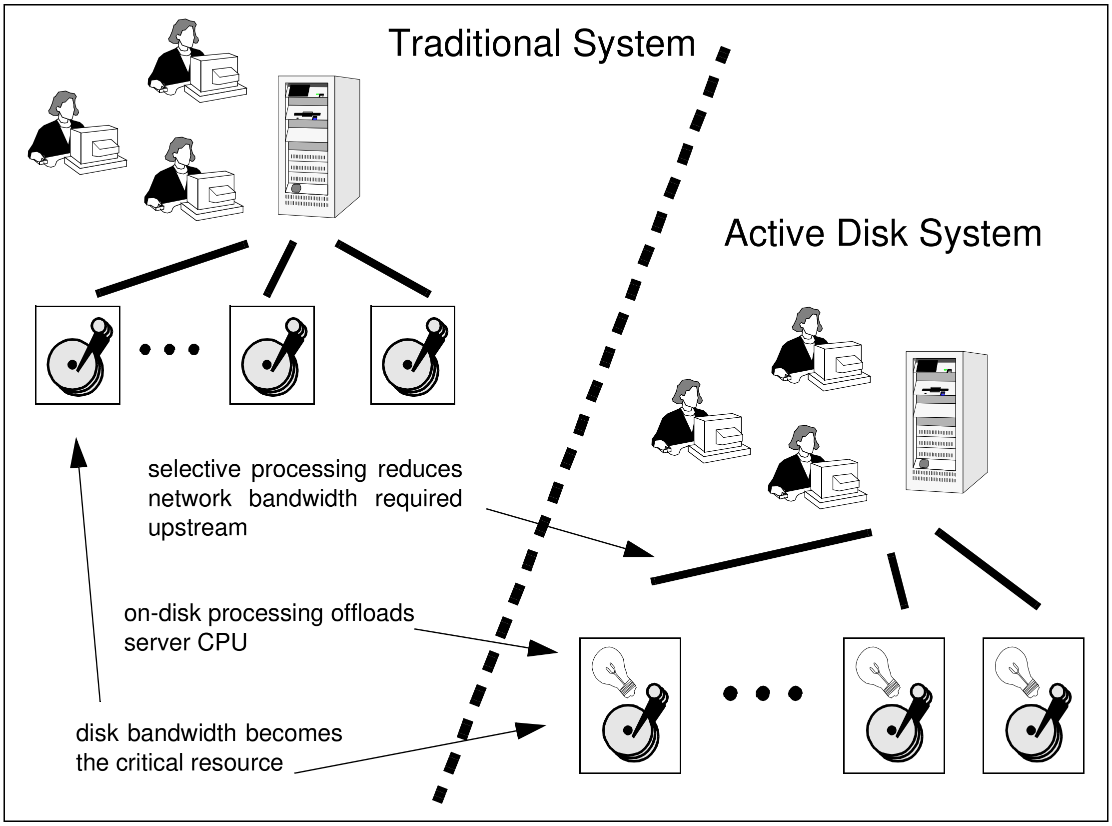


Figure 5-12 Diagram of a traditional server and an Active Disk architecture. By moving processing to the disks, the amount of data transferred on the network is reduced, the computation can take advantage of the parallelism provided by the disks and benefit from closer integration with on-disk scheduling. This allows the system to continue to support the same transaction workload with additional mining functions operating at the disks. In this case, the primary performance bottleneck becomes the bandwidth of the individual disks.

5.4.3 Proposed System

The performance benefits of Active Disks are most dramatic with the highly-selective parallel scans that form a core part of many data mining applications. The scheduling system proposed here assumes that a mining application can be specified abstractly as:

```
(1)  foreach block(B) in relation(X)
(2)      filter(B) -> B’
(3)  combine(B’) -> result(Y)
```

assumption: ordering of blocks does not affect the result of the computation

where steps (1) and (2) can be performed directly at the disk drives in parallel, and step (3) combines the results from all the disks at the host once the individual computations complete.

The performance of an application that fits this model and has a low computation cost for the `filter` function and high selectivity (data reduction from B to B’) will be limited by the raw bandwidth available for sequential reads from the disk media. In a dedicated mining system, this bandwidth would be the full sequential bandwidth of the individual disks. However, even in a system running a transaction processing workload, a significant amount of the necessary bandwidth is available in the "idle" time between and during disk seek and rotational latency for the transaction workload.

The key insight is that during disk seeks for a foreground transaction processing (OLTP) workload, disk blocks passing under the disk head can be read "for free". If the blocks are useful to a background application, they can be read without any impact on the OLTP response time by completely hiding the read within the request's rotational delay. In other words, while the disk is moving to the requested block, it opportunistically reads blocks that it passes over and provides them to the data mining application. If this application is operating directly at the disk drive in an Active Disk environment, then the block can be immediately processed, without ever having to be transferred to the host. As long as the data mining application - or any other background application - can issue a large number of requests at once and does not depend on the order of processing the requested background blocks, the background application will read a significant portion of its data without any cost to the OLTP workload. The disk will ensure that only blocks of a particular application-specific size (e.g. database pages) are provided, and that all the blocks requested are read exactly once, but the order of blocks will be determined by pattern of the OLTP requests.

Figure 5-13 shows the basic intuition of the proposed scheme. The drive maintains two request queues: 1) a queue of demand foreground requests that are satisfied as soon as possible; and 2) a list of the background blocks that are satisfied when convenient. Whenever the disk plans a seek to satisfy a request from the foreground queue, it checks if any of the blocks in the background queue are "in the path" from the current location of the disk head to the desired foreground request. This is accomplished by comparing the delay that will be incurred by a direct seek and rotational latency at the destination to the time required to seek to an alternate location, read some number of blocks and then perform a second seek to the desired cylinder. If this "detour" is shorter than the rotational delay, then some number of background blocks can be read without increasing the response time of the foreground request. If multiple blocks satisfy this criterion, the location that satisfies the largest number of background blocks is chosen. Note that in the simplest case, the drive will continue to read blocks at the current location, or seek to the destination and read some number of blocks before the desired block rotates under the head.

5.4.4 Experiments

All of the experiments in the following sections were conducted using a detailed disk simulator [Ganger98], synthetic traces based on simple workload characteristics, and traces taken from a server running a TPC-C transaction workload. The simulation models a closed system with a think time of 30 milliseconds which approximates that seen in our

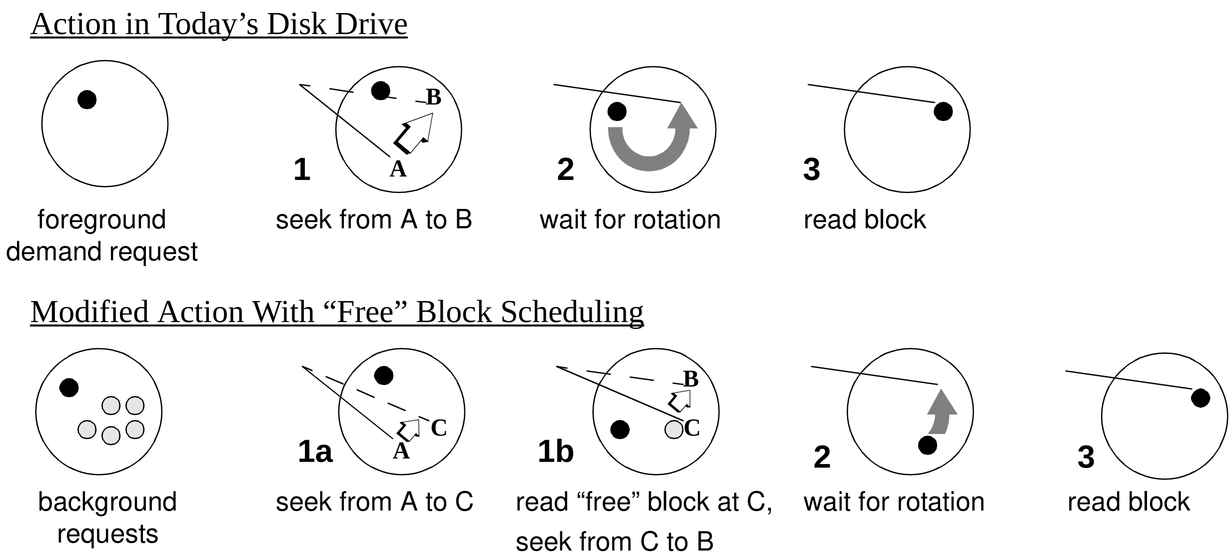


Figure 5-13 Illustration of 'free' block scheduling. In the original operation, a request to read or write a block causes the disk to seek from its current location (A) to the destination cylinder (B). It then waits for the requested block to rotate underneath the head. In the modified system, the disk has a set of potential blocks that it can read "at its convenience". When planning a seek from A to B, the disk will consider how long the rotational delay at the destination will be and, if there is sufficient time, will plan a shorter seek to C, read a block from the list of background requests, and then continue the seek to B. This additional read is completely 'free' because the time waiting for the rotation to complete at cylinder B is completely wasted in the original operation.

traces. The multiprogramming level of the OLTP workload is varied to illustrate increasing foreground load on the system. Multiprogramming level is specified in terms of disk requests, so a multiprogramming level of 10 means that there are ten disk requests active in the system at any given point (either queued at one of the disks or waiting in think time).

In the synthetic workloads, the OLTP requests are evenly spaced across the entire surface of the disk with a read to write ratio of 2:1 and a request size that is a multiple of 4 kilobytes chosen from an exponential distribution with a mean of 8 kilobytes. The background data mining (Mining) requests are large sequential reads with a minimum block size of 8 kilobytes. In the experiments, Mining is assumed to occur across the entire database, so the background workload reads the entire surface of the disk. Reading the entire disk is a pessimistic assumption and further optimizations are possible if only a portion of the disk contains data (see Section 5.4.4.5). All simulations run for one hour of simulated time and complete between 50,000 and 250,000 foreground disk requests and up to 900,000 background requests, depending on the load.

There are several different approaches for integrating a background sequential workload with the foreground OLTP requests. The simplest only performs background requests during disk idle times (i.e. when the queue of foreground requests is completely empty). The second uses the "free blocks" technique described above to read extra background blocks during the rotational delay of an OLTP request, but does nothing during disk idle times. Finally, a scheme that integrates both of these approaches allows the drive to service background requests whenever they do not interfere with the OLTP workload. This section presents results for each of these three approaches followed by results that show the effect is consistent as data is striped over larger numbers of disks. Finally, we present results for the traced workload that correspond well with those seen for the synthetic workload.

5.4.4.1 Background Blocks Only, Single Disk

Figure 5-14 shows the performance of the OLTP and Mining workloads running concurrently as the OLTP load increases. Mining requests are handled at low priority and are serviced only when the foreground queue is empty. The first chart shows that increasing the OLTP load increases throughput until the disk saturates and queues begin to build. This effect is also clear in the response time chart below, where times grow quickly at higher loads. The second chart shows the throughput of the Mining workload at about 2 MB/s for low load, but decreases rapidly as the OLTP load increases, forcing out the low priority background requests. The third chart shows the impact of Mining requests on OLTP response time. At low load, when requests are already fast, the OLTP response time increases by 25 to 30%. This increase occurs because new OLTP requests arrive while a Mining request is being serviced. As the load increases, OLTP request queueing grows,

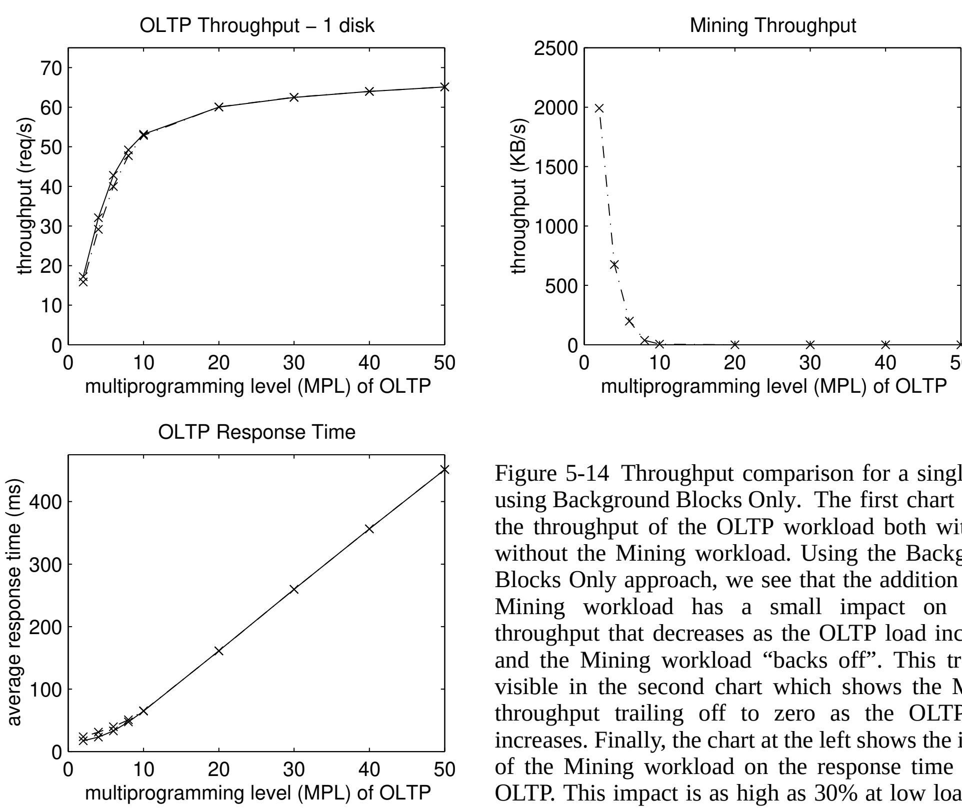
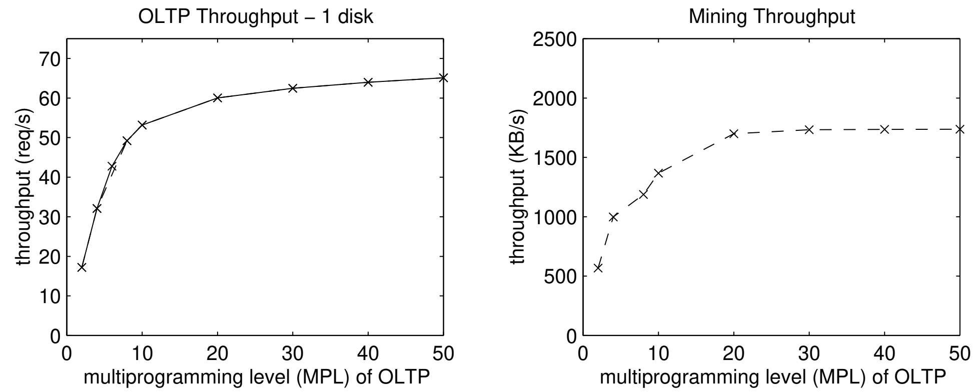


Figure 5-14 Throughput comparison for a single disk using Background Blocks Only. The first chart shows the throughput of the OLTP workload both with and without the Mining workload. Using the Background Blocks Only approach, we see that the addition of the Mining workload has a small impact on OLTP throughput that decreases as the OLTP load increases and the Mining workload “backs off”. This trend is visible in the second chart which shows the Mining throughput trailing off to zero as the OLTP load increases. Finally, the chart at the left shows the impact of the Mining workload on the response time of the OLTP. This impact is as high as 30% at low load, and decreases to zero as the load increases.



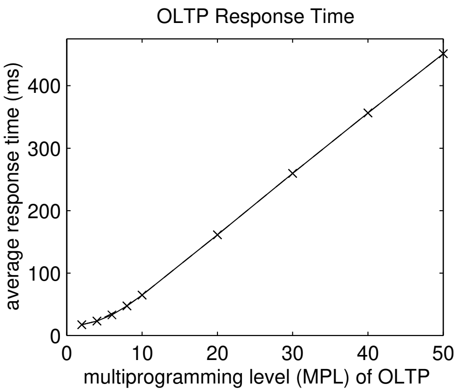


Figure 5-15 Performance of the Free Blocks Only approach. When reading exclusively 'free' blocks, the Mining throughput is limited by the rate of the OLTP workload. If there are no OLTP requests being serviced, there are also no 'free' blocks to pick up. One advantage of using only the 'free' blocks is that the OLTP response time is completely unaffected, even at low loads. The true benefit of the 'free' blocks comes as the OLTP load increases. Where the Background Blocks Only approach rapidly goes to zero at high loads, the Free Blocks Only approach reaches a steady 1.7 MB/s of throughput that is sustained even at very high OLTP loads.

reducing the chance that an OLTP request would wait behind a Mining request in service and eliminating the increase in OLTP response time as the Mining work is forced out.

5.4.4.2 'Free' Blocks Only, Single Disk

Figure 5-15 shows the effect of reading 'free' blocks while the drive performs seeks for OLTP requests. Low OLTP loads produce low Mining throughput because little opportunity exists to exploit 'free' block on OLTP requests. As the foreground load increases, the opportunity to read 'free' blocks improves, increasing Mining throughput to about 1.7 MB/s. This is a similar level of throughput seen in the Background Blocks Only approach, but occurs under high OLTP load where the first approach could sustain significant Mining throughput only under light load, rapidly dropping to zero for loads above 10. Since Mining does not make requests during completely idle time in the 'Free' Blocks Only approach, OLTP response time does not increase at all. The only shortcoming of the 'Free' Blocks Only approach is the low Mining throughput under light OLTP load.

5.4.4.3 Combination of Background and 'Free' Blocks, Single Disk

Figure 5-16 shows the effect of combining these two approaches. On each seek caused by an OLTP request, the disk reads a number of 'free' blocks as described in Figure 5-13 in the previous section. This models the behavior of a query that wishes to

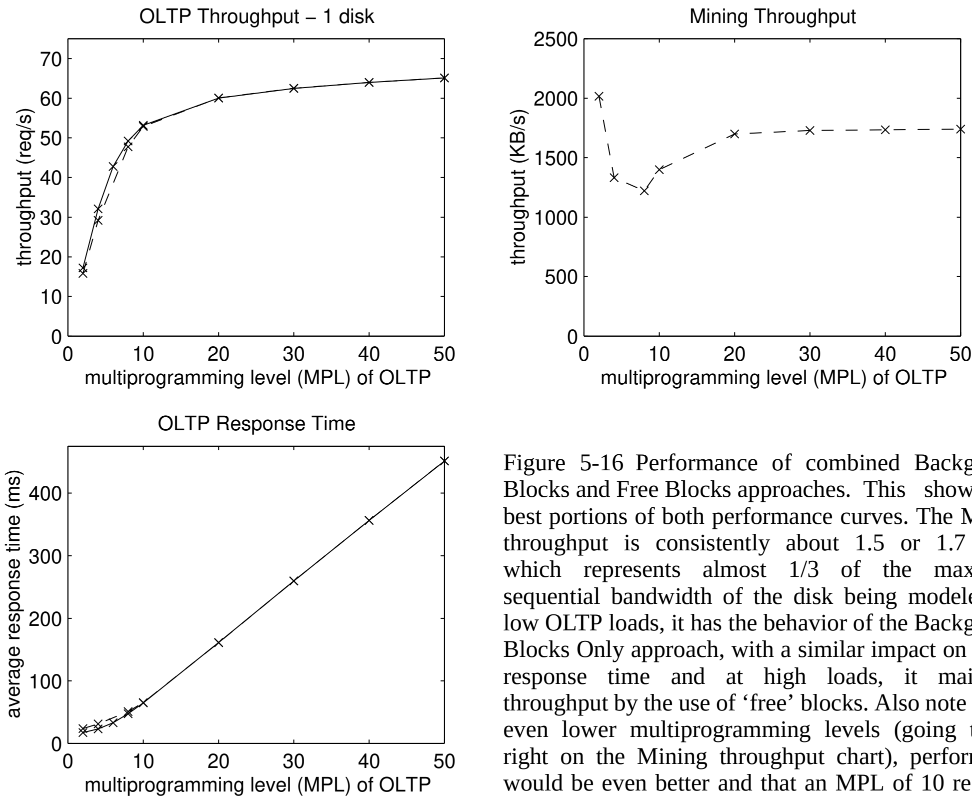


Figure 5-16 Performance of combined Background Blocks and Free Blocks approaches. This shows the best portions of both performance curves. The Mining throughput is consistently about 1.5 or 1.7 MB/s, which represents almost 1/3 of the maximum sequential bandwidth of the disk being modeled. At low OLTP loads, it has the behavior of the Background Blocks Only approach, with a similar impact on OLTP response time and at high loads, it maintains throughput by the use of 'free' blocks. Also note that at even lower multiprogramming levels (going to the right on the Mining throughput chart), performance would be even better and that an MPL of 10 requests outstanding at a single disk is already a relatively high absolute load.

scan a large portion of the disk, but does not care in which order the blocks are processed. Full table scans in the TPC-D queries, aggregations, or the association rule discovery application [Riedel98] could all make use of this functionality. Figure 5-16 shows that Mining throughput increases to between 1.4 and 2.0 MB/s at low load. At high loads, when the Background Blocks Only approach drops to zero, the combined system continues to provide a consist throughput at about 2.0 MB/s without any impact on OLTP throughput or response time. The full sequential bandwidth of the modeled disk (if there were no foreground requests) is only 5.3 MB/s to read the entire disk[1], so this represents more than 1/3 of the raw bandwidth of the drive completely "in the background" of the OLTP load.

5.4.4.4 Combination Background and 'Free' Blocks, Multiple Disks

Systems optimized for bandwidth rather than operations per second will usually have more disks than strictly required to store the database (as illustrated by the decision

1. As mentioned before, reading the entire disk is pessimistic since reading the inner tracks of modern disk drives is significantly slower than reading the outer tracks. If we only read the beginning of the disk (which is how "maximum bandwidth" numbers are determined in spec sheets), the bandwidth would be as high as 6.6 MB/s, but our scheme would also perform proportionally better.

support system of Table 5-13). This same design choice can be made in a combined OLTP/Mining system.

Figure 5-17 shows that Mining throughput using our scheme increases linearly as the workloads are striped across a multiple disks. Using two disks to store the same database (i.e. increasing the number of disks used to store the data in order to get higher Mining throughput, while maintaining the same OLTP load and total amount of "live" data) provides a Mining throughput above 50% of the maximum drive bandwidth across all load factors, and Mining throughput reaches more than 80% of maximum with three disks.

We can see that the performance of the multiple disk systems is a straightforward "shift" of the single disk results, where the Mining throughput with *n* disks at a particular MPL is simply *n* times the performance of a single disk at *1/n* that MPL. The two disk system at 20 MPL performs twice as fast as the single disk at 10 MPL, and similarly with 3 disks at 30 MPL. This predictable scaling in Mining throughput as disks are added bodes well for database administrators and capacity planners designing these hybrid systems. Additional experiments indicate that these benefits are also resilient in the face of load imbalances ("hot spots") in the foreground workload.

5.4.4.5 'Free' Blocks, Details

Figure 5-18 shows the performance of the 'free' block system at a single, medium foreground load (an MPL of 10 as shown in the previous charts). The rate of handling background requests drops steadily as the fraction of unread background blocks decreases and more and more of the unread blocks are at the "edges" of the disk (i.e. the areas not often accessed by the OLTP workload and the areas that are expensive to seek to). This means that if data can be kept near the "front" or "middle" of the disk, overall 'free' block performance would improve (staying to the right of the second chart in Figure 5-18). Extending our scheduling scheme to "realize" when only a small portion of the background work remains and issue some of these background requests at normal priority

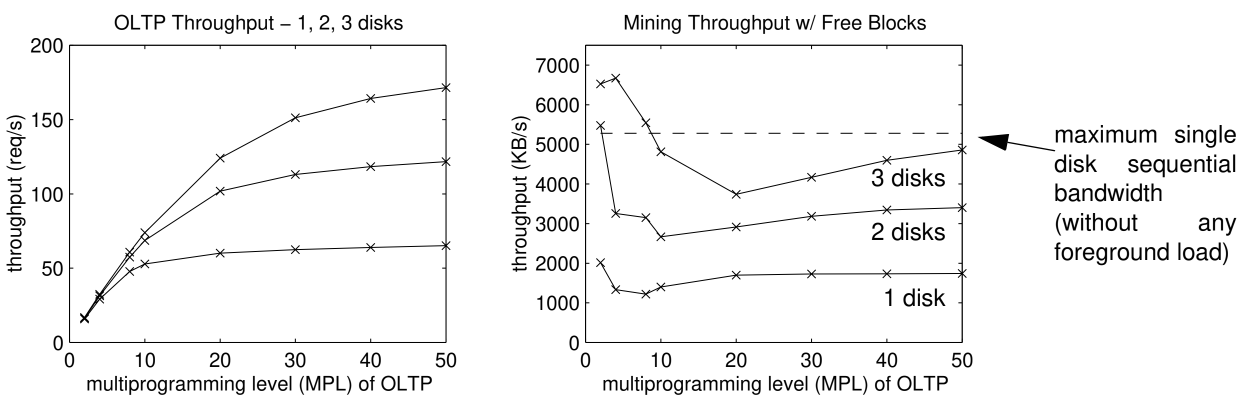


Figure 5-17 Throughput of 'free' blocks as additional disks are used. If we stripe the same amount of data over a larger number of disks while maintaining a constant OLTP load, we see that the total Mining throughput increases as expected.

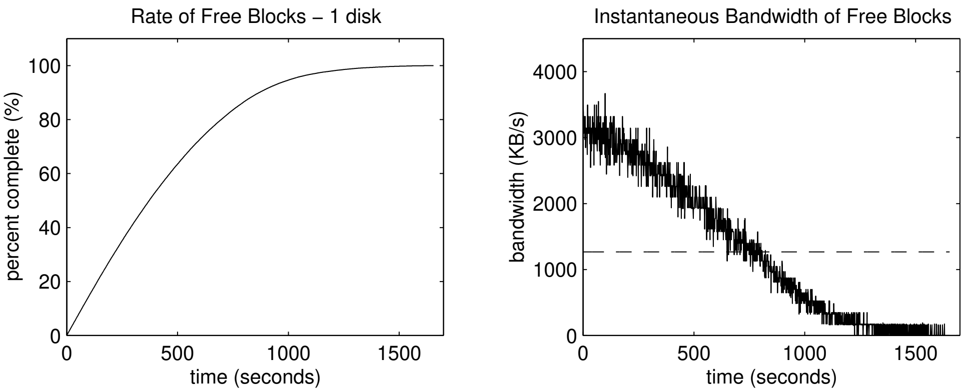


Figure 5-18 Details of ‘free’ block throughput with a particular foreground load. The first plot shows the amount of time needed to read the entire disk in the background at a multiprogramming level of 10. The second plot shows the instantaneous bandwidth of the background workload over time. We see that the bandwidth is significantly higher at the beginning, when there are more background blocks to choose from. As the number of blocks still needed falls, less of them are “within reach” of the ‘free’ algorithm and the throughput decreases. The dashed line shows the average bandwidth of the entire operation.

(with the corresponding impact on foreground response time) should also improve overall throughput. The challenge is to find an appropriate trade-off of impact on the foreground against improved background performance.

Finally, note that even with the basic scheme as described here, it is possible to read the entire 2 GB disk for ‘free’ in about 1700 seconds (under 28 minutes), allowing a disk to perform over 50 “scans per day” [Gray97] of its entire contents completely unnoticed.

5.4.4.6 Workload Validation

Figure 5-19 shows the results of a series of traces taken from a real system running TPC-C with varying loads. The traced system is a 300 MHz Pentium II with 128 MB of memory running Windows NT and Microsoft SQL Server on a one gigabyte TPC-C test

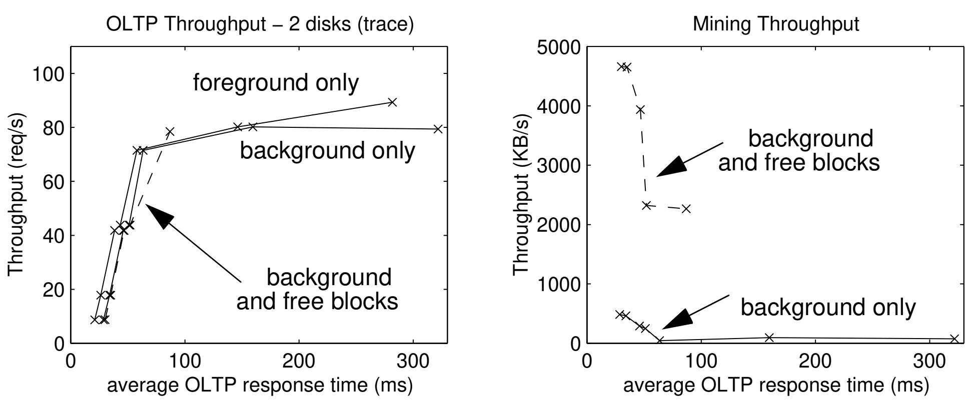


Figure 5-19 Performance for the traced OLTP workload in a two disk system. The numbers are more variable than the synthetic workload, but the basic benefit of the ‘free’ block approach is clear. We see that use of the ‘free’ block system provides a significant boost above use of the Background Blocks Only approach. Note that since we do not control the multiprogramming level of the traced workload, the x axes in these charts are the average OLTP response time, which combines the three charts given in the earlier figures into two and makes the MPL a hidden parameter.

database striped across two Viking disks. When we add a background sequential workload to this system, we see results similar to those of the synthetic workloads. At low loads, several MB/s of Mining throughput are possible, with a 25% impact on the OLTP response time. At higher OLTP loads, the Mining workload is forced out, and the impact on response time is reduced unless the 'free' block approach is used. The Mining throughput is a bit lower than the synthetic workload shown in Figure 5-19, but this is most likely because the OLTP workload is not evenly spread across the disk while the Mining workload still tries to read the entire disk.

The disk being simulated and the disk used in the traced system is a 2.2 GB Quantum Viking 7,200 RPM disk with a (rated) average seek time of 8 ms. We have validated the simulator against the drive itself and found that read requests come within 5% for most of the requests and that writes are consistently under-predicted by an average of 20%. Extraction of disk parameters is a notoriously complex job [Worthington95], so a 5% difference is a quite reasonable result. The under-prediction for writes could be the result of several factors and we are looking in more detail at the disk parameters to determine the cause of the mismatch. It is possible that this is due to a more aggressive write buffering scheme modeled in the simulator than actually exists at the drive. This discrepancy should have only a minor impact on the results presented here, since the focus is on seeks and reads, and an underprediction of service time would be pessimistic to our results. The demerit figure [Ruemmler94] for the simulation is 37% for all requests.

5.4.5 Summary

The previous chapters have shown that Active Disks can provide the compute power, memory, and reduction in interconnect bandwidth to make data mining queries efficient on a system designed for a less demanding workload. This section illustrates that there is also sufficient disk bandwidth in such a system to make a combined transaction processing and data mining workload possible. It shows that a significant amount of data mining work can be accomplished with only a small impact on the existing transaction performance. This means that if the "dumb" disks in a traditional system are replaced with Active Disks, there will be sufficient resources in compute power, memory, interconnect bandwidth, and disk bandwidth to support both workloads. It is no longer necessary to buy an expensive second system with which to perform decision support and basic data mining queries.

The results in Section 5.4.4.5 indicate that the current scheme is pessimistic because it requires the background workload to read every last block on the disk, even at much lower bandwidth. There are a number of optimization in data placement and the choice of which background blocks to "go after" to be explored, but the simple scheme described here shows that significant gains are possible.

Chapter 6: Software Structure

This chapter describes the structure and implementation of the Active Disk code for the applications described in the previous sections. It describes the basic structure of an application for Active Disks, as well as the details of the prototype implementation. In addition, it describes the separation of the database system into a host and Active Disk portion, and the basic set of primitive functions required at the Active Disks to support this structure. Finally, it quantifies a promising characteristics of code running on the Active Disks, the ability to specialize a piece of code to the exact execution environment in a particular drive architecture and for a particular application function.

6.1 Application Structure for Active Disks

This section provides an outline of the structure of applications that execute on Active Disks, including that design philosophy, the structure of the on-drive code and the types of changes required for the code that remains on the host.

6.1.1 Design Principles

The basic design principles of developing an application to run in an Active Disk setting are to 1) expose the maximum amount of parallelism in the data-intensive portion of the processing, 2) isolate the code to be executed at the disks as much as possible to form self-contained and manageable units, and 3) utilize adaptive primitives to take full advantage of variations in available resources during execution. These three goals allow the largest amount of performance and flexibility in the placement and execution of the application code.

6.1.2 Basic Structure

The basic structure of an Active Disk application is that a “core” piece of code will run at each of the drives, while any high-level synchronization, control, or merging code continues to run at the host. This creates a client-server parallel programming model as illustrated in Figure 6-1. Input parameters are initialized at the host and distributed to all of the disks. Each of the disks then computes on its own local data and produces its portion of the final result. The host collects the results from all the disks and merges them into the final answer. Since there is a portion of code that runs at the host and processes data from

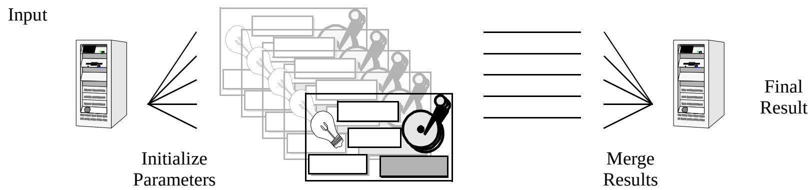


Figure 6-1 Basic structure of Active Disk computation. The host initializes the computation, each of the drives computes results for its local data, and these results are combined by the host.

the disks, it is always possible to "fix up" an incomplete computation at the drive, as long as the drive functions always act conservatively. If they improperly filter a record that should have been part of the result, for example, then the host code will never catch it.

The high-level structure of an Active Disk application is similar to the normal processing loop that simply uses that basic filesystem calls to do I/O. At an abstract level, any data processing application will have a structure similar to the following:

```
(1)  initialize
(2)  foreach block(B) in file(F)
(3)       read(B)
(4)       operate(B) -> B'
(5)       combine(B') to result(R)
(6)  cleanup
```

The challenge for Active Disks is to specify the code for steps (3) and (4) in such a way that this portion of the code can be executed directly at the disks, and in parallel. This means there must not be any dependence on global state, or requirements for ordering in the processing of the blocks, because the execution will occur in parallel across all the disks on which `F` is stored. In all of the applications discussed here, this extraction of the appropriate code was done manually within the source code, although it should be possible to do a significant portion of this extraction automatically, or at least provide tools to aid the programmer in identifying candidate sections of code and eliminating global dependencies.

As a specific example of this two-stage processing, operating in parallel on the separate blocks in a file and then combining all these partial results, consider the frequent sets application. It operates on blocks of transaction data individually and converts them into itemset counts. The itemsets counts for all the blocks are then combined at the host simply

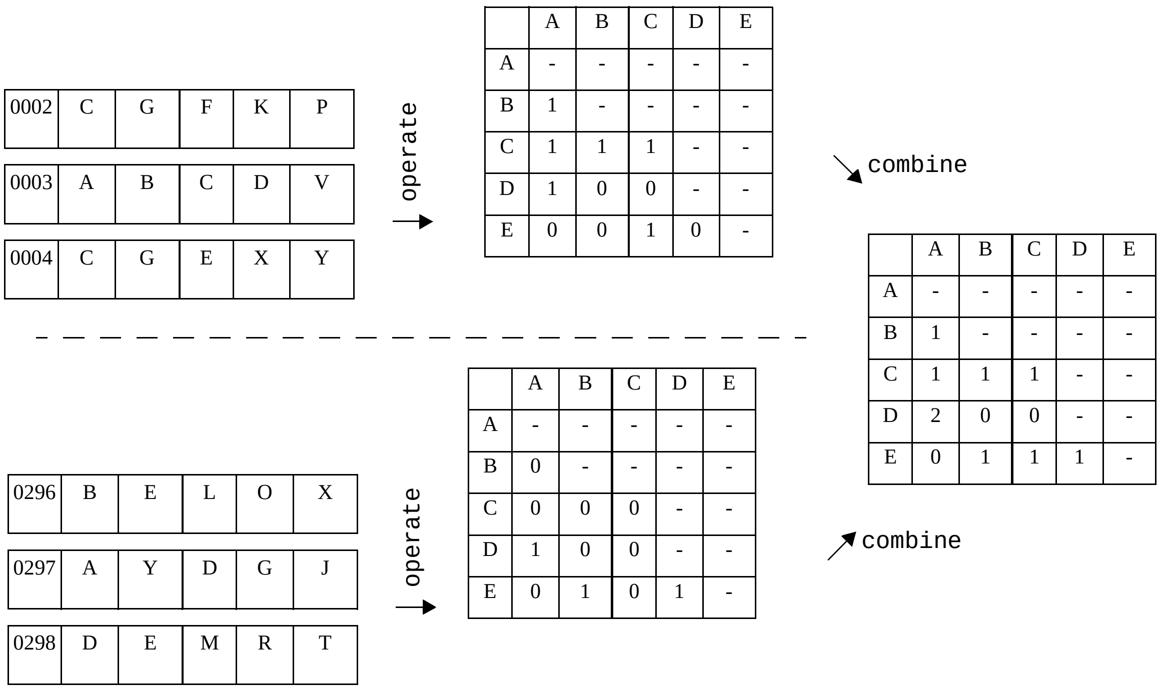


Figure 6-2 Basic structure of one phase of the frequent sets application. The blocks of transaction data are converted into itemset counts, which can then be combined into total counts.

by summing the individual counts, as shown in Figure 6-2. The `operate` step for frequent sets takes raw transaction data and converts it into an array of counts of candidate itemsets. The figure shows the 2-itemset phase, where the counts are stored as a two-dimensional array and a 1 is added whenever a pair of items appears together in a transaction (in a particular shopper's basket). This set of counts can be calculated independently at each of the disks, and the `combine` step merges the arrays from all the disks by simply summing the values into a final array. There are no global dependencies once the initial list of candidate itemsets is distributed, and each block of transactions can be processed in parallel and in any order.

The search application is partitioned in a similar manner, with the `operate` step choosing the *k* closest records on a particular disk, and the `combine` step choosing the *k* closest records across all these individual lists. As with the frequent sets application, the serial fraction of the search application is orders of magnitude less expensive than the parallel, counting fraction of the application, which leads to the linear speedups shown in Chapter 5.

The edge detection and image registration applications operate purely as filters, so they consist of completely parallel `operate` phases that convert an input image into a set of output parameters. These is no `combine` step, as the results are simply output for each image, without being combined further at the host. Since these two applications contain essentially no serial fraction (every application has a tiny serial fraction in overhead to start the computation across n disks, however, this cost is quickly amortized when used across reasonably-sized data sets), they also show linear scalability in the prototype results from Chapter 5.

6.2 Implementation Details

This section provides specific details about the prototype implementation evaluated in the previous chapter. The prototype makes specific assumptions about how the code is organized and what the execution environment at the drives looks like. These assumptions are not specifically required, but are discussed here to motivate one particular set of design choices.

6.2.1 Background - NASD Interface

The prototype uses the Network-Attached Secure Disk (NASD) system developed at Carnegie Mellon as a basis. This means that the drives are already network-attached devices that can communicate directly with clients, with high-level management functions provided by an external *file manager* that is not on the primary data path [Gibson97]. The interface to these drives is object-based, rather than block-based as SCSI is today. The drive provides a flat namespace of *objects* or extents, where space management and layout within an object are controlled by the drive itself. In the traditional SCSI interface, a drive provides a linear address space of fixed-size (and small, 512 bytes is the standard unit) blocks. Higher-level filesystems are then responsible for allocating and managing this space, without the drive having any knowledge of how these individual blocks are aggregated into the "objects" seen by users (whether files, database tables, etc.). The object interface of NASD allows the drive to manage sets of blocks as a whole, and communicate to clients at a higher level. The choice of how to map user-understood "objects" onto NASD objects is up to the filesystems that are implemented on top of NASD. The group at Carnegie Mellon has explored the mapping of the Network File System (NFS) and the Andrew File System (AFS) onto the object interface, where each file and directory in these filesystems is mapped onto a separate NASD object [Gibson97a]. One could assume more complex mappings where larger groups of files are mapped onto objects, or where a single file is broken into multiple objects, but this work only considers the case where files map onto single NASD objects. This allows the entire file to be treated as a single unit for optimization and management.

6.2.2 Goals

There were several goals in the implementation of the prototype that influenced the basic decisions made. It was desireable for the Active Disks functions to 1) operate outside the security system, 2) get resource management and thread of control from the host, and 3) minimize the amount of code that is executed at the drives.

By operating above the security system, the functions executing on the disks are required to obtain and present the same capabilities to the security system in order to gain access to data as functions executing on clients would. The functions executing at the disks would not have priveleges beyond those of external code, except for the fact that they could execute operations more quickly than a host, making, for example, a denial-of-service attack more dangerous. By operating outside the security system, the data on the disks is protected, just as it is in a conventional NASD system, without remote functions. If a client does not have a capability to access a particular object, then there is no way that a remote function operating on behalf of that client could gain the necessary capability.

By depending on the host for a thread of control, the drive does not have to manage the highest level execution of an application. Just as in a traditional disk, the drive obtains a request for service, expends some amount of resources servicing that request, and returns control to the client (whether or not the client is blocked waiting for the request to return, or is expecting to be "called back" when the request have been completed makes no difference to the underlying system, the programmer is logically giving a "thread of control" to the drive for the duration of the request, whether it is requiring it to operate synchronously or is prepared to handle asynchronous completion).

By minimizing the amount of code that is run at the drives, the Active Disk code is "built up" from simply reading the data to adding parts of the surrounding processing, rather than being "stripped down" for a much larger piece of code. This allows the programmer to be more careful about both the amount of parallelism available and the amount of synchronization required by the code.

6.2.3 Basic Structure

All of the data mining and multimedia applications described in Chapter 4 follow the same basic model. The on-disk functions operate by processing a block of data after it is read from the disk and before it is sent on the network, as follows:

```
(1)  request(R)
(2)  map(R) -> block_list(L)
(3)  foreach block(B) in block_list(L)
(4)       read(B) from disk
(5)       operate(B) -> B'
(6)       send(B') to host
(7)  complete(R)
```

where the `operate` operation reduces the amount of data to be sent on the network by the selectivity factor of the particular application. Note that this code structure allows the order in which blocks are read and processed to be determined by the on-disk scheduler, opening the way for optimizations such as the one discussed at the end of Chapter 5.

6.2.3.1 Initialization

Active Disk processing is initiated by specifying a particular function to execute as part of a modified SETATTR request. This causes the Active Disk runtime system to pass the filesystem_specific portion of the SETATTR to the installed code as a set of parameters of the form:

```
typedef struct remote_param_s {
      nasd_identifier_t object_id; /* object to operate on */
      nasd_len_t block_len; /* block size */
      nasd_identifier_t code_obj; /* code to execute */
      nasd_identifier_t param_obj; /* init parameters */
}
```

This causes a new Active Disk computation state to be initialized and bound to the particular object_id. The initialization returns a tag which refers to this particular instance of the execution for this object.

In the prototype, all the functions are directly linked with the code of the drive, so the code_obj identifiers are simply static "code slots" at the drive that point to one of the pieces of on-disk code. In a general-purpose system, this pointer would specify a second on-disk object that contains the code to be executed. This code object would then contain the text of the function to be executed using some type of portable, mobile code system (such as Java, for example, but many other systems are possible).

Note that this arrangement allows multiple code segments (via separate installs) to be attached to the same object_id, allowing for example a "counting" and a "filtering" request stream all to be active against the same object at the same time. It does *not* provide any way for multiple computations with the same code but different "states" to be active at the same time (except by installing the same code twice).

6.2.3.2 Operation

Function execution is accomplished by specifying the tag returned from the Initialization as a special parameter to a normal READ request that includes an object identifier, offset and length specification, and an appropriate capability. The requested data is read and the function executed before data is sent on the network. Each request is accepted individually because this allows the host to control the overall pace of the execution. A request from the host provides both a capability (for the security system) to access a particular region of the object and manages the high-level resource management. Each request "enables" a particular amount of execution on the drive processor.

Another design option would be to allow the code to be initiated at the drive and then told to "START", with the drive controlling the pace of the execution. In choosing the request-at-a-time option, the idea is to let the host continue to provide high-level flow control and manage the execution, rather than letting the thread of control transfer explicitly to the drive. This level of control must be balanced against the desire to give the drive sufficiently large request to optimize over. The greater range of the object that the request covers, the more aggressive the drive scheduler can be. Within the region of the object specified by the READ, the drive is welcome to reorder requests as it sees fit[1].

6.2.3.3 Memory Allocation

The goal of the memory allocation system is to prevent unnecessary copying of data into and out of the Active Disk applications and to equitably share memory segments between the remote programs and the disk cache. Instead of copying buffers between the disk subsystem ("kernel") and remote code ("user"), block descriptors are passed between modules. The Active Disk function simply acquires pages from the on-disk cache that are not backed by physical blocks on the disk. These pages can then be used by the Active Disk function at will.

In the protoype implementation, an allocation of blocks to an Active Disk function is permanent until it is explicitly released by the function. In a general-purpose system, memory pages for Active Disk functions would be integrated into the same page-management algorithm used for allocating pages for caching to individual objects and request streams. This would provide a mechanism whereby Active Disk functions would be asked to give up pages, or have them forcibly reclaimed.

6.2.3.4 Interface

The following list the functions are provided to support the four basic data mining and multimedia applications:

```
typedef struct filter_itemset_param_s {
      unsigned int min_support;
} filter_itemset_param_t;

void filter_setup_itemsets(filter_itemset_param_t params)
void filter_itemsets(char* buffer, unsigned int len)
void filter_complete_itemsets(char* output, unsigned int *out_len,
      unsigned int max_len)
```

1. this allows optimizations such as the reordering discussed at the end of Chapter 5, although this system is not implemented in the current prototype and blocks are simply processed sequentially.

```
typedef struct filter_search_param_s {
       int salary; int commission; int loan; int age; int zip;
       int car; int house; int education; int years; int group;
       unsigned int num_matches;
} filter_search_param_t;

void filter_setup_search(filter_search_param_t params)
void filter_search(char* buffer, unsigned int len)
void filter_complete_itemsets(char* output, unsigned int *out_len,
       unsigned int max_len)
```

```
typedef struct filter_edges_param_s {
       unsigned int which_algorithm;
       unsigned int brightness;
} filter_edges_param_t;

void filter_setup_edges(filter_edges_param_t params)
void filter_edges(char* buffer, unsigned int len,
              char* output, unsigned int *out_len, unsigned int max_len)
void filter_complete_edges(void)
```

```
typedef struct filter_register_param_s {
       int max_iterations;
} filter_register_param_t;

int filter_setup_register(filter_register_param_t params)
void filter_register(char* buffer, unsigned int len,
       char* output,unsigned int *out_len, unsigned int max_len)
void filter_complete_register(void)
```

Note the similarity between the the frequent sets and the search, both of which simply consume their inputs, with results provided by the final complete call, while both the edge detection and image registration functions produce output results as they go, returning a reduced amount of data (potentially none) in the output buffer for each block processed. The same basic structure is seen for all four functions, an initialization routine that takes a particular set of parameters, a processing routine that is executed once for each block of data, and a completion function to extract any collected result.

6.3 Database System

This section outlines the changes necessary to adapt a relational database system for use with Active Disks. Figure 6-3 gives an overview of the PostgreSQL structure for pro-

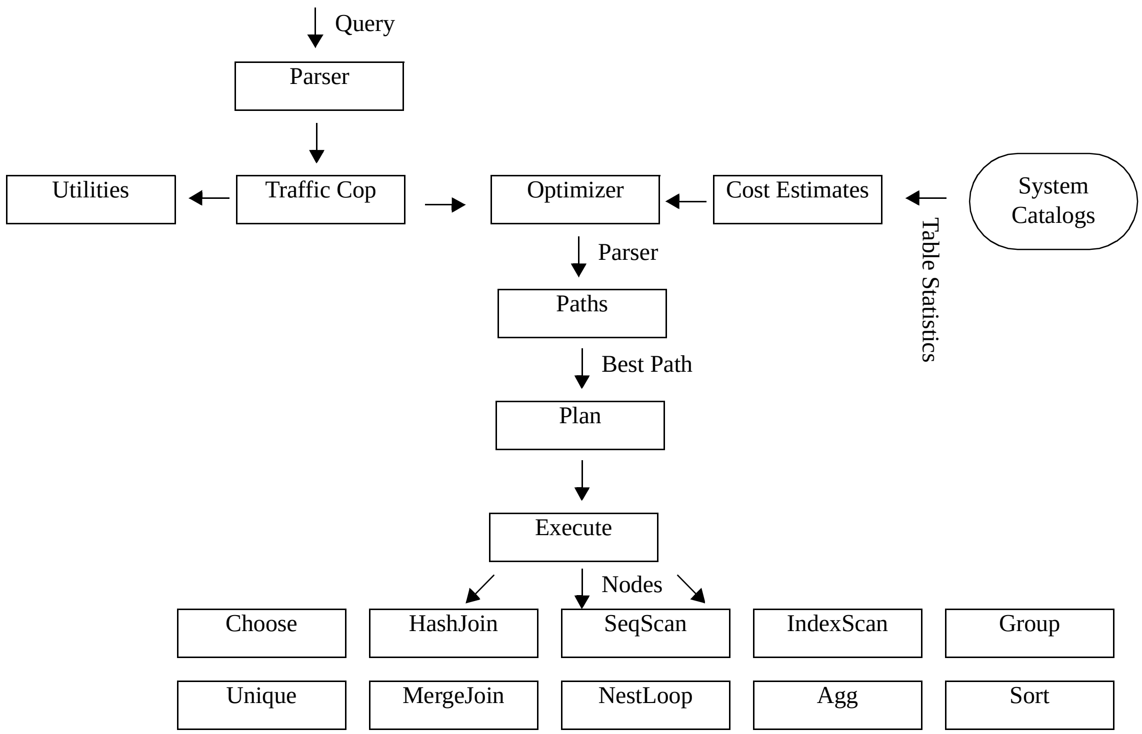


Figure 6-3 Overview of PostgreSQL query structure. The PostgreSQL engines accepts a query, determines a set of possible paths for executing the query, uses cost estimates to choose an optimal plan, and then processes the query as a series of execute nodes.

cessing a query. A query is parsed from the SQL input string. The traffic cop determines whether the request is a SQL statement to be further optimized (select, insert, delete, etc.) or a utility function (create table, destory table, etc.) and passes the request to the proper place. If the query is to be optimized, the optimizer generates a description of all the possible execution paths as a series of nodes. It then uses cost estimates that take into account the cost of the various nodes and statistics from the tables to be processed to determine the optimal path. This path is then converted into a final plan and initialized for execution. The execution engine takes the set of nodes that make up the query and executes them in turn, from bottom up, allowing pipeline parallelism wherever possible.

Figure 6-4 expands the detail of one of the Execute nodes to incorporate the access to storage and the required data type functions. The diagram illustrates a sequential scan node that traverses a table from beginning to end, matching tuples against a qualification condition and returning only matching tuples. A database page is read from the disk, processed by the File subsystem, which provides the interaction with the underlying operating system, passed through a particular access method (a Heap in this case), and then combined with the table schema that identifies the layout of tuples on the page. Tuples are then

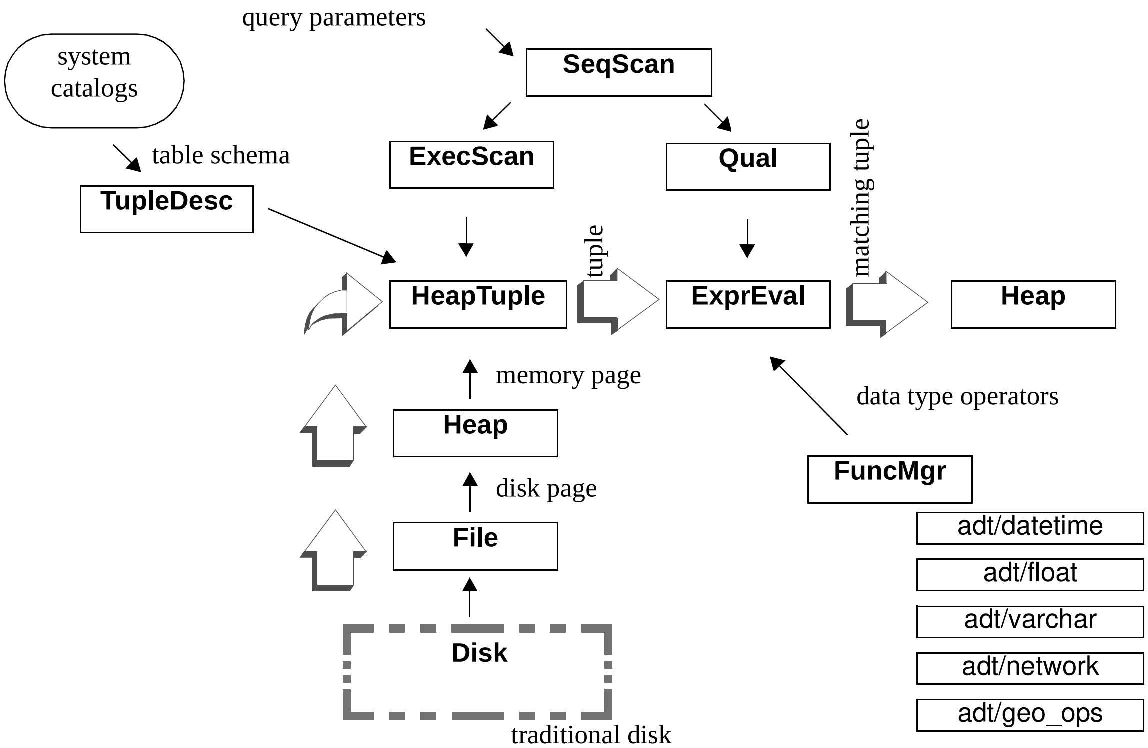


Figure 6-4 Details of a PostgreSQL Execute node. The diagram shows the execution of a sequential scan node, including the access method for getting tuples from disk and tuple descriptions and data type-specific functions through the Function Manager.

processed one by one and the qualification condition is evaluated. The evaluation of the qualification requires both the condition as described in the query (e.g. the field offset and constant to match against for `dept_id = 5`) and the data type-specific functions to perform the comparison (e.g. the code for =, equality of integers). If the tuple matches, it is passed along to the next node in the chain. Other nodes, such as MergeJoin, combine tuples from multiple relations, so these will have two “streams” of incoming tuples. A node such as Group that groups a set of tuples matching a particular condition together may have several logical “streams” of output tuples.

Figure 6-5 shows the changes necessary in a system with Active Disks to execute the same sequential scan node. The code is logically split into a drive and host portion. We see that the disk now processes much more of the code than before. The tuple descriptions from the table schema, the expressions and constants from the qualification, and the operators for evaluating the condition are all statically bound into the Active Disk function and shipped to the drives for execution. Just as the original File module was modified to process pages in parallel from multiple drives, the execution nodes are modified to accept tupes in parallel from multiple drives executing concurrently.

The modified diagram in Figure 6-6 shows the high-level overview of PostgreSQL as modified for Active Disks. The query optimizer is extended to take into account system

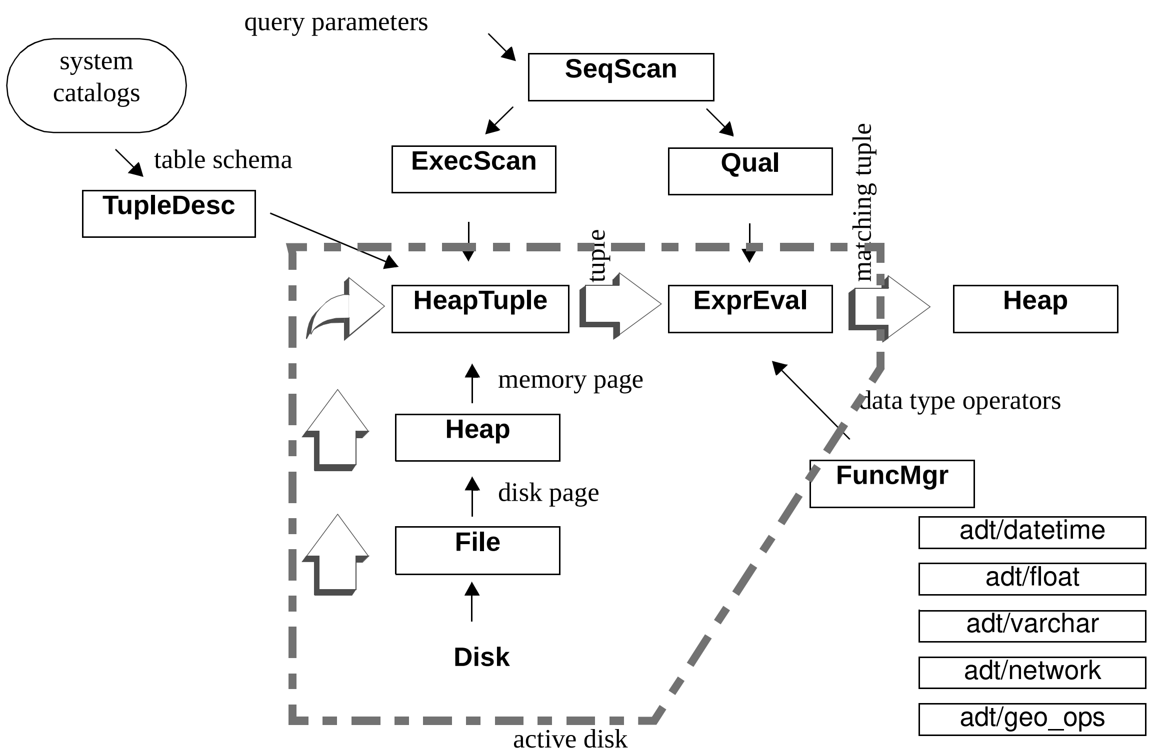


Figure 6-5 PostgreSQL Execute node for Active Disks. The diagram shows the execution of a sequential scan node, including the access method for getting tuples from disk and tuple descriptions and data type-specific functions through the Function Manager.

configuration parameters such as the number and types of Active Disks and the network connecting them. In addition, several of the execute nodes are combined to produce logical nodes that can be executed more efficiently in an Active Disk system. Essentially, this combines the Group and Aggregation node types with the Sort node, which would have otherwise been executed one after the other, but can be done much more efficiently fused together and executed as a whole. When combined into a single execution node, tuples can be aggregated as they are sorted. The basic algorithm used is replacement selection, where a tuple that matches the key of an existing tuple in the tree is merged into a single "aggregated" tuple, rather than inserting the new tuples into the tree. This means that the total amount of memory required for an aggregation is the size of the aggregated result (i.e. determined by the number of unique values of the `group by` keys), rather than the size of the input. If the nodes are not combined, then this computation is performed by PostgreSQL in two stages, where the data is first sorted, and then adjacent tuples are aggregated. This requires a much greater amount of memory and is not necessary since the aggregation only needs to be able to combine adjacent tuples, and never requires all the tuples to be in fully sorted order. Of course, this type of optimization is not limited to the Active Disk case, it simply makes the execution primitive on the Active Disks more efficient as well as more adaptive.

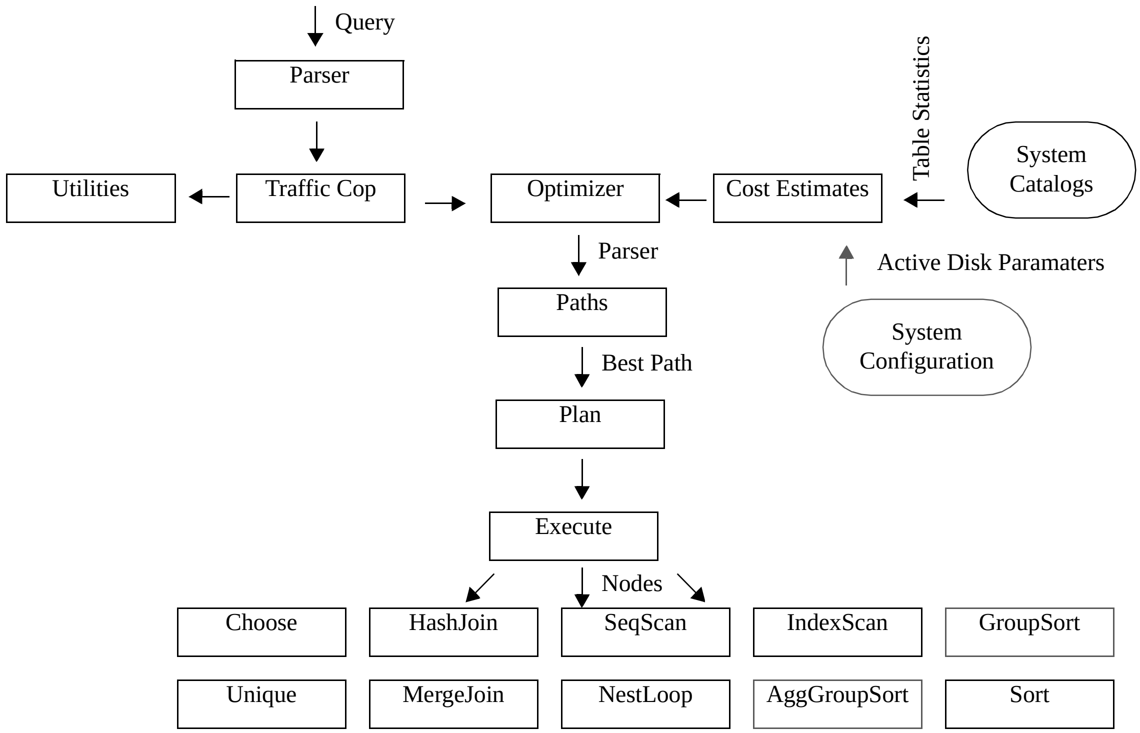


Figure 6-6 Overview of PostgreSQL query structure with Active Disks. The cost estimates are modified to also take into account the system parameters, including the number and capabilities of Active Disks in the system. For efficient execution, the Active Disk system also combines several node types so that Group and Sort and Aggregation, Group, and Sort are done as entire nodes that can be executed together, rather than in series, one after the other.

6.3.1 Query Optimization

The PostgreSQL query optimizer must be modified to take into account the presence of a particular number of Active Disks, as well as their relative memory sizes and processing rates. The optimizer must also have some idea of the processing power and memory of the host and the basic performance characteristics of the network connecting the Active Disks to the host. The optimizer can then combine this system information with characteristics of the data in the tables being operated on and the structure and parameters of the query to estimate selectivities and determine the appropriate placement of functions.

Figure 6-7 shows the query plan generated by the PostgreSQL optimizer for Query 5 from the TPC-D decision support benchmark. This query performs a total of five joins among six relations, followed by a final sort and aggregation step.

6.3.1.1 Cost Functions

The optimizer contains cost estimates for each of the basic operations parameterized by the sizes of the tables, the selectivities of the qualification and join conditions being evaluated, and the basic system parameters. Table 6-1 shows the cost equations for each

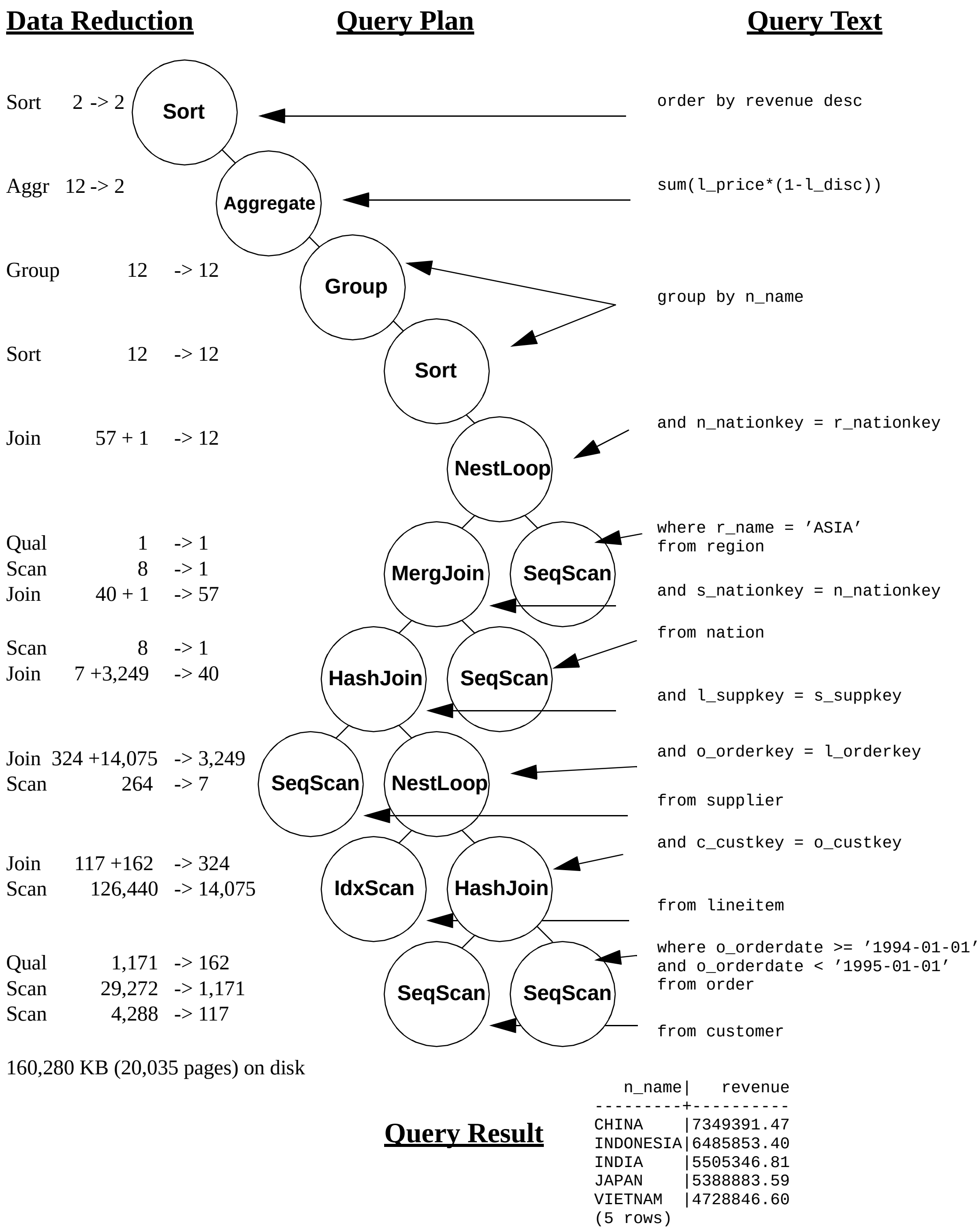


Figure 6-7 Text, execution plan, and result for Query 5. The right column shows the text of the query, the center diagram shows the final plan chosen by the optimizer for this execution, and the left column shows the amount of data reduction at each node in the plan (according to the optimizer estimates).

Execute node as implemented in the default PostgreSQL optimizer and the modifications required to take into account the availability of Active Disk processing.

Node	Basic Equation	Equation with Active Disks
SeqScan	$\frac{npages}{r_d} + \frac{ntuples \cdot nclauses \cdot w}{s_{\mathrm{cpu}}}$	$\frac{npages}{r_d'} + \frac{ntuples \cdot nclauses \cdot w}{d \cdot s_{\mathrm{cpu}}'}$
IndexScan	$\frac{nindexpages + nexpectedpages}{r_d} + \frac{ntuples \cdot selectivity \cdot w}{s_{\mathrm{cpu}}} + \frac{nindextuples \cdot selectivity \cdot w_{index}}{s_{\mathrm{cpu}}}$	$\frac{nindexpages + nexpectedpages}{r_d'} + \frac{nindexpages \cdot d}{r_n'} + \frac{ntuples \cdot selectivity \cdot w}{d \cdot s_{\mathrm{cpu}}'} + \frac{nindextuples \cdot selectivity \cdot w_{index}}{d \cdot s_{\mathrm{cpu}}'}$
HashJoin	$Hash(ninnertuples) + \frac{ninnerpages \cdot (nrun - 1)}{r_d} + \frac{noutertuples \cdot w}{s_{\mathrm{cpu}}}$	$Hash(ninnertuples) + \frac{ninnerpages \cdot (nrun - 1)}{r_d} + \frac{ninnerpages \cdot d}{r_n'} + \frac{noutertuples \cdot w}{d \cdot s_{\mathrm{cpu}}'}$
Hash	$\frac{ntuples \cdot selectivity \cdot w}{s_{\mathrm{cpu}}}$	-

Table 6-1 Cost equations used within the PostgreSQL optimizer. The table shows the cost functions used in the default PostgreSQL optimizer, as well as the modifications necessary to support estimates of Active Disk processing.

6.3.1.2 System Parameters

In order to compare the query performance on the Active Disks against execution exclusively at the host, some additional parameters about the underlying components must be available. In the default optimizer, there is only one choice for placement of the function, so the relative performance of the on-disk processing is not a consideration. It is, however, necessary to consider the size of the host memory even in normal optimization because the algorithms chosen, particularly for joins and sorting, depend on how much of a relation can fit into memory at one time. The parameters tracked are the same System Parameters used in the performance model of Chapter 3. The relative cpu rates of the disks and host, the relative network rates, and the rate of the underlying raw disks.

Note that the PostgreSQL system only handles a single query at a time, so there is no inter-query optimization. It is assumed that all system resources are available to execute the query being considered. This is obviously limiting in the general case, and additional work would be necessary to extend the optimizations to take into account current system load and the availability of resources over time for a long-running query.

```
starelid|staattnum|staop|stalokey    |stahikey
--------+---------+-----+------------+----------------------
   18663|        1|   66|           1|600000
   18663|        2|   66|           1|20000
   18663|        3|   66|           1|1000
   18663|        4|   66|           1|7
   18663|        5|  295|           1|50
   18663|        6|  295|         901|95949.5
   18663|        7|  295|           0|0.1
   18663|        8|  295|           0|0.08
   18663|        9| 1049|           A|R
   18663|       10| 1049|           F|O
   18663|       11| 1087|  01-02-1992|12-01-1998
   18663|       12| 1087|  01-31-1992|10-31-1998
   18663|       13| 1087|  01-08-1992|12-30-1998
   18663|       14| 1049| COLLECT COD|TAKE BACK RETURN
   18663|       15| 1049|         AIR|TRUCK
   18663|       16| 1049| 0B6wmAww2Pg|zzzyRPS40ABMRSzmPyCNzA6
[...more...]
(61 rows)
```

Figure 6-8 Statistics tables maintained by PostgreSQL for the lineitem table. Depending on the type and range of a particular column, the optimizer has sufficient information to accurately estimate the sizes of intermediate query results.

```
attrelid|attname        |atttypid|attdisbursion|attlen|attnum
--------+---------------+--------+-------------+------+------
   18663|l_orderkey     |      23|  2.33122e-06|     4|     1
   18663|l_partkey      |      23|  1.06588e-05|     4|     2
   18663|l_suppkey      |      23|  0.000213367|     4|     3
   18663|l_linenumber   |      23|    0.0998572|     4|     4
   18663|l_quantity     |     701|   0.00434997|     8|     5
   18663|l_extendedprice|     701|  2.66427e-06|     8|     6
   18663|l_discount     |     701|    0.0247805|     8|     7
   18663|l_tax          |     701|    0.0321099|     8|     8
   18663|l_returnflag   |    1042|     0.307469|    -1|     9
   18663|l_linestatus   |    1042|     0.300911|    -1|    10
   18663|l_shipdate     |    1082|  8.94076e-05|     4|    11
   18663|l_commitdate   |    1082|  8.33926e-05|     4|    12
   18663|l_receiptdate  |    1082|  8.90733e-05|     4|    13
   18663|l_shipinstruct |    1042|     0.100238|    -1|    14
   18663|l_shipmode     |    1042|    0.0451101|    -1|    15
   18663|l_comment      |    1042|            0|    -1|    16
[...more...]
(572 rows)
```

6.3.1.3 Database Statistics

Figure 6-8 shows the statistics maintained by PostgreSQL for the `lineitem` table from TPC-D. The table has a total of 16 attributes and the first statistics table gives the minimum and maximum values for each column. The second table gives the types and sizes of each attribute, as well as an estimate of the disbursion of the values currently in the database for that attribute. In PostgreSQL, this disbursion is calculated by counting the item that occurs most frequently in the particular column plus 20% of the number of unique values. A more sophisticated database system would keep a small number of histograms for each column to provide better estimates at the cost of additional space for storing the statistics. These statistics are maintained in a lazy fashion, usually through an explicit `analyze` function that is run against the database at regular intervals or after any bulk loading operation. These statistics are only used to estimate the cost for query planning, so 100% accuracy is not required.

From the second table, we see that an attribute such as the orderkey have a very low disbursion value, meaning that there are a large number of values, while attributes such as the returnflag and linestatus have a high disbursion, meaning a much smaller number of unique values. The first table then provides additional details by showing the minimum and maximum values. For example, looking at `l_returnflag` and `l_linestatus`, which are the `group by` keys for Query 1 from TPC-D, the data type is known to be `varchar`, with a size of 1 (from the schema for the `lineitem` relation), which means that the optimizer can guarantee that no `group by` result will exceed 65536 (256 x 256) unique values, no matter how many records are processed. By looking at the low and high values in the statistics, this can be further reduced to 153 (from 'A' to 'R' in the `l_returnflag`, and from 'F' to 'O' in the `l_linestatus`). Finally, considering the disbursion values for these two attributes gives an estimate of about 10 (1/0.3 x 1/0.3). It turns out that there are only four unique combinations of these two attributes in the final result, but the statistics have helped narrow the size estimate from the 600,000 records in the original `lineitem` table, to an estimate of 65536, 153, or 10 output records, depending on which statistics are used.

6.3.1.4 Estimates

Table 6-2 shows the date sizes at the intermediate points of several TPC-D queries.

Query	Input Data (KB)	Scan Result (KB)	Optimizer Estimate (KB)	Qual Result (KB)	Optimizer Estimate (KB)	Aggr Result (bytes)	Optimizer Estimate (bytes)
Q1	126,440	35,189	35,189	34,687	33,935	240	9,180
Q4	29,272	2,343	2,343	86	141	80	64
Q6	126,440	9,383	9,383	177	43	8	8

Table 6-2 Data sizes and optimizer estimates for stages several TPC-D queries. Plans are as chosen by the PostgreSQL optimizer. We see that the optimizer may both under- and over-estimate the expected size.

The table shows the estimates made by the PostgreSQL optimizer at each stage as well as the actual size during execution of the query. Notice that the optimizer may overestimate or underestimate the size and selectivities of the various stages of processing, but usually comes very close to the actual size. The Scan column gives the estimate for the data reduction by removing unneeded columns from the relation - for example, not returning the address or comment fields when these are not part of the query result. Since this is a static reduction in the amount of data based on the database schema, and this version of PostgreSQL always stores fixed-size fields, the estimate for this step will always be correct. The Qual column estimates the qualification condition in the `where` clause of the query - `l_shipdate <= '1998-09-02'` in the case of Query 1. Finally, the Aggr column estimates the size of the aggregation result, using 153 records as the estimate for Query 1, based on the statistics discussed at in the previous section.

6.3.2 Storage Layer

The lowest-level storage layer in the database must be replaced with a system that understands how to communicate with network-attached Active Disks. The basic PostgreSQL system maps each relation in the database into a separate UNIX file. This makes mapping these files into a NASD object system straightforward - each file (and therefore each relation) becomes a single object in NASD. This means that operations such as sequential scans that operate on the relation as a whole have the advantage of adressing a single object within NASD. When data is striped over multiple disks, each disk contains an object with its portion of the relation. This means that in a system with 10 disks, the relation will consist of ten NASD objects, one on each disk. Data is written to the disks in round-robin fashion, using a block size of 256 KB. Since the database pages are 8 KB in size, no page is ever split between multiple disks, eliminating the need for special detection of such edge cases, as would be required for an application with arbitrary record sizes.

When a request is made for a particular page within a relation, the storage layer maps this request into a block request from the appropriate disk. The storage layer also performs prefetching when it detects that a file is being scanned sequentially and has sufficient buffer space available.

6.3.3 Host Modifications

This section describes the modifications required to the PostgreSQL code to support the prototype results presented here. This gives an idea of the extent of the modification necessary to have a database system support Active Disk functions.

6.3.3.1 Parameter Passing

The host must provide a “bypass” mechanism whereby information for scans, aggregates, and joins can be sent to the drives as parameter values to the core operations. This includes scan conditions, relation schemas, and the bloom filter bit vectors for semi-join.

For scans and semi-joins, the work at the drives is considered “pre-work” and the host-side code does not have to know what work has been done. For the aggregation, this is more difficult, because certain aggregation functions cannot be handled without knowing that pre-computation has already happened at the drives (e.g. `average`, which must track the total number of records counted, as well as the running total of the values). In this case, the format of the table seen at the host is implictly modified to be a representation of the table with the “pre-aggregates” included as additional columns. The on-disk code is responsible for converting tuples in the actual format of the table to tuples containing the pre-aggregates necessary for this particular query.

6.3.4 Active Disk Code

The code for PostgreSQL processing on the Active Disks can be divided into four categories. The disk must include the code for basic page layout, for the layout of tuples or records within a page, support for operations on the data types required by a particular

Module	Original		Modified Host (New & Changed)		Active Disk	
	Files	Code	Files	Code	Files	Code
access	72	26,385	-	-	1	838
bootstrap	2	1,259	-	-	-	-
catalog	43	13,584	-	-	-	-
commands	34	11,635	-	-	-	-
executor	49	17,401	9	938	4	3,574
parser	31	9,477	-	-	-	-
lib	35	7,794	-	-	-	-
nodes	24	13,092	-	-	6	4,130
optimizer	72	19,187	2	620	-	-
port	5	514	-	-	-	-
regex	12	4,665	-	-	-	-
rewrite	13	5,462	-	-	-	-
storage	50	17,088	1	273	-	-
tcop	11	4,054	-	-	-	-
utils/adt	40	31,526	-	-	2	315
utils/fmgr	4	2,417	-	-	1	281
utils	81	19,908	-	-	1	47
Total	578	205,448	12	1,831	15	9,185
					New	1,257

Table 6-3 Code sizes for the Active Disk portions of PostgreSQL. Size of the various portions of the Active Disk code for the database system.

query (operators such as less-than, greater-than, equality, plus, or minus), and support for the core database functions. This means that all the infrastructure for query parsing, query optimization, and recovery does not have to be duplicated at the drive. Table 6-3 gives the sizes and breakdown of the major portions of code required at the drives in the PostgreSQL prototype.

6.3.4.1 Page Layout

The code required at the drive must understand the layout of a PostgreSQL page. Since database pages come in fixed-length chunks (8 KB, in the case of PostgreSQL), this is relatively easy to manage at the Active Disks. There are no concerns about alignment or pages that span multiple disks, as there is in the case of arbitrary filesystem files. However, this code will still need to be specific each database system being used, Oracle and Informix, for example, will have their own page layout formats.

6.3.4.2 Tuple Layout

The code for dealing with database schemas and tuple layouts, as well as NULL-handling, must also be specific for each database system. There must be a way to describe this structure between the host-based code and the drive code. In the execution of PostgreSQL with Active Disks, the tuple format is one of the parameters passed to the Active Disk code when it is initialized.

6.3.4.3 Data Type Operators

Operators for dealing with basic calculations on database fields must be present at the drive for all the data types that the drive supports. This includes comparison operators for scanning and sorting and the arithmetic operators for aggregation.

In PostgreSQL, the type system is extensible through user-provided functions, and these functions are written in C and linked with the core database code. For the prototype Active Disk system, only the operators necessary for completing the TPC-D queries were ported to the drive, and the total code size of these pieces is shown in Table 6-3.

6.3.4.4 Core Operations

The basic database operations on Active Disks - scan, semijoin, sort, and aggregation can be used in common among multiple database systems. These operations require user-provided routines for comparing and merging tuples, but the code for the basic scan, sort, and replacement selection with aggregation can be common among multiple database platforms. The basic operators are `scan`, `semijoin`, and `aggregate`.

6.3.4.5 Scan

The `scan` primitive at the drive supports all simple scans, using a static condition that is evaluated for every tuple in the relation, such as the `where` clause from Query 1 of TPC-D that specifies `l_shipdate <= '1998-09-02'`. The `tuple_desc` describes the layout of tuples within the relation and `qual_expression` gives the condition to be evaluated, which is can be any SQL condition, including constants, expressions, and references to individual fields in the relation being processed.

```
typedef struct database_scan_param_s {
       char* tuple_desc; /* format of the tuples on disk */
       char* qual_expression;
} database_scan_param_t;

int database_setup_scan(database_scan_param_t params)
void database_scan(char* buffer, unsigned int len,
       char* output,unsigned int *out_len, unsigned int max_len)
void database_complete_scan(void)
```

When used in a database system that allows user-defined-functions and the addition of user-defined abstract data types, this scan primitive with the appropriate data types could be used to implement all of the data mining and multimedia applications described in Chapter 4. This would require providing the user-defined functions and data type operators that operate as disk functions, just as the basic data type operators must be made available at the Active Disks.

6.3.4.6 Semijoin

The `semijoin` primitive at the drive supports the semijoin portion of a full join operation. The `tuple_desc` describes the layout of tuples within the relation and `join_key` gives the field that is being joined. The `join_filter` is the bloom filter representing a list of join key values, such as those for `l_partkey` from Query 9 or `l_suppkey` from Query 5.

```
typedef struct database_semijoin_param_s {
      char* tuple_desc; /* format of the tuples on disk */
      char* join_keys;
      char* join_filter;
} database_semijoin_param_t;

int database_setup_semijoin(database_scan_param_t params)
void database_semijoin(char* buffer, unsigned int len,
      char* output,unsigned int *out_len, unsigned int max_len)
void database_complete_semijoin(void)
```

When used in a multiple-pass algorithm such as the hybrid hash join [DeWitt90], the bloom filter is simply set to select only tuples from the the partition that is currently being processed. Depending on the number of partitions, simply re-scanning the entire relation and returning the matching tuples will be more efficient than writing the partitioned records during the initial scan phase.

6.3.4.7 Aggregate/Sort

The `aggregate` primitive at the drive supports both aggregation and generic sorting. The basic algorithm used is replacement selection in both cases. Aggregates are simply a special case where records with matching keys are merged using the aggregation function, rather than being inserted into adjacent slots in the sorted heap. In the case of sorting, the `aggr_expr` is null and no merge operation is provided, all records are simply output in sorted order. If the amount of memory at the Active Disks exceeds the size of the sorted output, then separate sorted runs are output and must then be merged at the host. This allows a single, flexible primitive to be used for both operations.

```
typedef struct database_aggr_param_s {
      char* tuple_desc;/* format of the tuples on disk */
      char* sort_keys;
      char* aggr_expr;
} database_aggr_param_t;

int database_setup_aggr(database_aggr_param_t params)
void database_aggr(char* buffer, unsigned int len,
      char* output,unsigned int *out_len, unsigned int max_len)
void database_complete_aggr(void)
```

The use of replacement selection as the basic algorithm provides the benefit of longer average run length, adaptivity in the face of changing memory conditions (the ability to give up memory pages as the operation progresses, and make use of additional memory page that become available [Pang93a]), and support for the merging operation necessary to perform aggregation while sorting.

6.4 Code Specialization

Since users have to rewrite their code to take advantage of Active Disks, this is an excellent opportunity to get them to write the code "the right way" and reduce some of the aliasing and code analysis problems that traditional software now suffers. The work of the COMPOSE group at IRISA and the Synthetix work at OGI have shown that code specialization through partial evaluation can be a powerful tool [Massalin89, Pu95, Volanschi96, Consel98, Marlet99]. This type of specialization is particularly benefitial for operating system code and for small code "kernels" such as the functions to be executed on Active Disks. The work of Consel et al. makes the observation that 25% of operating system code is spent verifying arguments and parsing (or "interpreting") system data structures. This work is often redundant across calls - traversing the same pointer chain in a ready queue, for example - and can easily be specificialized [Consel98]. Furthermore, this type of specialization can also allow a particular piece of code to be optimized for the environment in which it runs - taking knowledge of the particular machine architecture available on the drive (cache size, processor type) or the number of memory buffers available into account.

We know that once a particular piece of code is sent to the drive it will be executed many times, amortizing the specialization cost. This leverages the information the programmer provides when creating an Active Disk function ("this is important, this is the core part of my application") where a general-purpose system (running on a host for example) would have to first "discover" which particular pieces of code to specialize.

Such selective specialization should also be particularly successful for the core database functions that often traverse the same basic expression tree during the execution of a particular query. Database engines are basically “interpreters” from a query language, SQL, to the functions implementing tuple layout, memory management, and the core database functions.

The next two sections examine the possibilities for code specialization in the context of the PostgreSQL system and the TPC-D benchmark running on Active Disks.

6.4.1 Case Study - Database Aggregation

Table 6-4 shows the cost of executing Query 1 from the TPC-D benchmark using C code written specifically to handle that single query for the particular table schema using a file of test data from the benchmark. Data is read from a binary file of records, and the

query	type	computation (instr/byte)	throughput (MB/s)	memory (KB)	selectivity (factor)	instructions (KB)
Q1	aggregation	1.82	73.1	488	816	9.1 (4.7)
Q13	hash-join	0.15	886.7	576	967,000	14.3 (10.5)

Table 6-4 Cost of Query 1 and Query 13 in direct C code implementation. The computation cost and memory requirements of a basic aggregation and hash-join implemented in C code specifically written to access the TPC-D tables from raw disk files. The last column also gives the total size of the code executed at the drives (and the total size of the code that is executed more than once).

entire processing of Query 1 is hand-coded into the program. No processing of schema descriptions or of the query text is done at runtime. We see that the instruction per byte cost is very low, giving a very high theoretical throughput on the prototype Active Disk. This should represent close to the fastest possible execution of this particular query, with the exact schema of the records on disk, the datatypes, as well as the query text, known at compile time.

Table 6-5 shows the same aggregation query as executed by the full PostgreSQL code. We see that the cost is much higher when the fully general code that handles arbitrary datatypes and query texts, is used. This code is able to handle tables and records of an arbitrary schema, rather than being specific to one particular record format. This code

operation	computation (instr/byte)	throughput (MB/s)	selectivity (factor)
scan	28	17.8	4.00
qualification	29	17.2	1.05
sort/group	71	7.0	1.00
sort/aggregate	196	2.5	3,770

Table 6-5 Cost of Query 1 using the full PostgreSQL code. The computation cost of the phases of computation to handle the query in a general database system.

also deals with an arbitrary expression for the Qualification, the Sort, and the Aggregate steps, where the C code is hard-coded for the particular qualification constants, sort keys, and aggregation expressions used in Query 1 of TPC-D.

These two sets of code represent the extremes of the spectrum, the fully general code from PostgreSQL that can handle any SQL query to the direct C code implementation that can only perform a single hard-coded query. The insight of a code specialization and optimization system is to bridge the two order of magnitude performance gap between the two. Since the PostgreSQL code is repeatedly processing the same format tuples and evaluating the same conditions, it should be possible to generate more efficient, specialized code for the execution of any given query.

Figure 6-9 shows the structure of the PostgreSQL code that is executed at the Active Disk. We see that a number of parameters and operators can be statically bound into the code at the time it is prepared for Active Disk execution. This means that the code on the Active Disks can take advantage of the knowledge of the tuple layouts, condition parameters, and operators to specialize for this particular query. By statically binding the tuple descriptions, the qualification expression and constants, and the subset of data type opera-

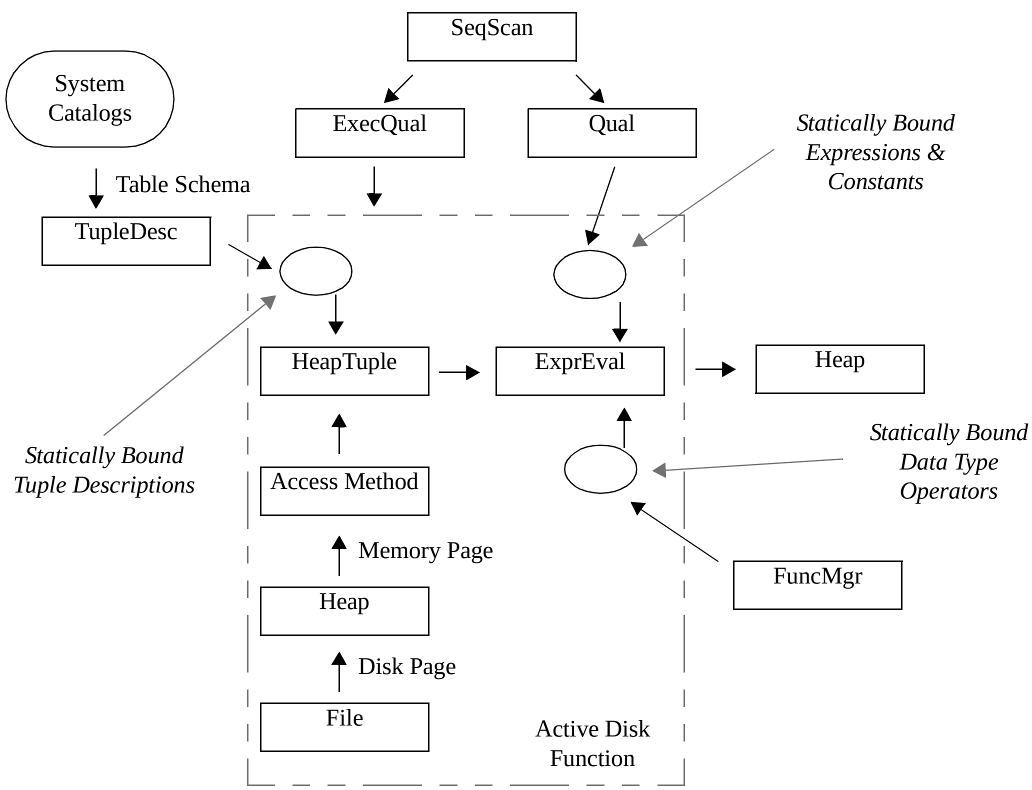


Figure 6-9 Active Disk processing of PostgreSQL Execute node. This diagram shows the change to the previous diagram necessary to support Active Disk processing. Note that the table schemas, expressions, and data type operators are statically bound in as part of the Active Disk function, this enables further optimizations to specialize the code that executes at the drives.

Routine	File	Instructions	Percent	Description
memcpy	libc	6,385,334	20.81	copy buffer
ExecEvalVar	executor/execQual.c	4,598,000	14.98	evaluate column
ExecMakeFunctionResult	executor/execQual.c	3,572,000	11.64	result
ExecEvalExpr	executor/execQual.c	3,534,000	11.52	expression
ExecEvalFuncArgs	executor/execQual.c	2,774,000	9.04	arguments
fmgr_c	utils/fmgr/fmgr.c	1,748,000	5.70	function dispatch
bzero	libc	1,421,000	4.63	clear buffer
process_rawtuple	executor/rawUtil.c	1,406,563	4.58	
ExecQual	executor/execQual.c	1,027,126	3.35	qualification
ExecEvalOper	executor/execQual.c	912,005	2.97	operator
ExecQualClause	executor/execQual.c	684,000	2.23	
ExecStoreTuple	executor/rawUtil.c	418,000	1.36	
date_gt	utils/adt/rawdatetime.c	228,000	0.74	data comparison

Table 6-6 Most frequently executed routines in PostgreSQL select. These thirteen routines account for close to 95% of the execution time at the drive.

tors needed for a particular schema and query text, the runtime code specialization system can create code that is much closer to the hand-coded version in performance.

6.4.2 Case Study - Database Select

This section explores the types of specialization possible in the code for select in PostgreSQL as described in Chapter 5. Table 6-6 shows the amount of code executed for the thirteen most popular routines in the database `select` operation on a column of type `date`. We see that the largest single fraction of the time is spent copying data, but that the tuple processing and interpretation take over 50% of the code in just six routines. Much of the processing in these routines is repetitive and can be specialized away when the code is optimized for a particular query. By collapsing the general-purpose data type and expression-parsing routines that operate on any data types into a single routine that only knows how to compare a single date column with a constant date value, the total number of instructions necessary would be greatly reduced.

Further study is needed to determine how much of these indivdual routines can be specialized, and what trade-offs the runtime system must make between the potential savings and the overhead of performing the specialization, but the data presented in the last two sections shows that this is a promising direction for an Active Disk runtime system to take advantage of.

Chapter 7: Design Issues

There are many additional design considerations for Active Disk systems. This chapter addresses a number of these issues and provides an overview of the questions that will need to be answered by anyone designing an environment for Active Disks.

7.1 Partitioning of Code for Active Disks

The basic contention of the previous chapter on Software Structure was that a single version of the application code can be written that can then be executed at the drive or on the host as necessary. Such code would be written in a mobile code system and would expose the maximum amount of parallelism possible in the core processing. This would then allow the code to be distributed across any number of drives (without concern for how many drives were being used), or to be run at the host, perhaps as multiple threads in an SMP system. The user is required to re-write their application once, using separate, mobile modules inside the I/O-intensive loops. This code can then be distributed among hosts and drives either statically placed by the programmer, or automatically placed by a runtime system. In the case of the database system, this placement can take advantage of the existence of the query optimizer to assist in this placement. This allows the database runtime system to combine statistics about the underlying data sets being processed and the properties of the code being executed to choose an optimal placement (and to choose when Active Disk execution will improve performance). These placement decisions are more difficult in general-purpose code, that does not have as much detailed information as the query optimizer.

A system with a single set of mobile code modules at its core also provides a natural way to utilize systems with legacy disks that do not support Active Disk processing. If the drive does not have Active Disk capabilities, or if it is does not have the resources available to perform a particular requested function, then the disk can act as a normal drive and return only the requested data as stored, with processing occurring exclusively at the host.

7.2 Identifying I/O Intensive Cores

Identifying the characteristics of appropriate code for Active Disks can be tricky. Table 7-1 shows the breakdown of application characteristics across several phases of the

Phase	Computation (inst/byte)	Memory (KB)	Program (KB)	Selectivity (factor)
1-itemsets	8.31	32	1.3	-
2-itemsets	22.22	582	1.3	-
3-itemsets	90.59	426	2.9	-
4-itemsets	29.85	426	2.9	-
5-itemsets	29.77	49	2.8	-
6-itemsets	12.82	49	2.8	-
overall	32.37	992	23.1	1891

Table 7-1 Computation, memory, and code sizes of the frequent sets application. The highlighted numbers in each column show the maximum value for that parameter across all the phases. We see that the maximum differ considerably from the averages at the bottom of the columns. If the application were measured as a whole, the parameters would be those across the bottom, whereas the execution environment would see the much higher values at runtime.

frequent sets application. We see that if we look at the overall average behavior, we get a significantly different picture than if we look at the individual phases one at a time.

This data is obtained by instrumenting a single execution of the code and counting the total number of instructions executed, the number of unique memory pages accessed, and the total size of the code executed. Simply looking at a execution profile does not reveal this multiple-phase structure. The tool used to obtain these measurements captures data between successive calls to `read()`. An additional difficulty in identifying the proper "core" code in this particular application comes from the fact that the program actually has three call sites that initiate a read from the filesystem, but the tool only identifies one use of the `read()` system call since all three application-level calls use the `fread()` routine from the `stdio` library instead of calling `read()` directly. This will be a challenge for any automated tool trying to identifying the various phases of computation, since such a tool would have to unravel all these layers of abstraction.

7.3 Why Dynamic Code

One suggestion for Active Disk processing is to have only a fixed set of functions, for example sort, join, aggregate, and filter that allow settings from the programmer to handle field lengths, comparison functions (for sorts), and simple finite state machines for filters. The contention is that such a very basic set of primitives could capture 80 percent of the cases where Active Disk execution is desired. The benefit of this approach is that the on-drive code can be developed separately from the rest of the application code. The basic primitives essentially become part of the firmware of the drive, and are developed and tested by the drive manufacturers. The disadvantage of this approach is that it limits the flexibility of the on-drive functions to the set of primitives included with the drive. This eliminates one of the basic benefits of Active Disks, to be able to execute arbitrary application code that allows any application to take advantage of the processing capability. In addition, this scheme does not have a direct benefit for safety, since user-provided routines such as the comparison functions for sorting, might still cause unpredictable behavior.

7.4 On-Drive Processing

One of the obvious questions about the viability of the Active Disk approach is the availability of “excess” cycles on the disks. The disk controller is already performing a number of complex functions, and it is natural to question how much processing power is available for “new” functions, particularly expensive user-provided code. It is certainly true that today’s disk drives are not designed with any “excess” processing capacity, due to the cost constraints of the drive market. However, there are several trends that suggest that a significant amount of processing cycles will be available in the future. First, the current drive controller must be powerful enough to meet the peek demand of the drive, meaning that it’s average utilization should be much lower. Second, the electronics in modern drives are designed to offload as much of the “common case” functions as possible from the control processor. This is the goal of the specialized drive ASIC discussed in Chapter 2, which handles servo processing (the most compute-intensive of the basic drive control tasks), data transfer (via specialized DMA engines), and any specialized functions such as drive-support XOR [Seagate98a]. This leaves the control processor only responsible for functions such as request scheduling and buffer management.

The most processor-intensive function performed by the drive is the processing of a `WRITE` request from a host. On a Fibre Channel drive, the disk must receive burst data from the host at (theoretically) up to 100 MB/s. The disk must accept this data at the full speed of the host, store it in its memory buffers, and then write it (much more slowly) to the media. The buffer allocation necessary for a large `WRITE` will be the most pressure on the control processor. During `READ` requests, data is streamed directly from the media to the network by the DMA engines, leaving the processor largely idle. During a seek, the servo engine is responsible for tracking the location of the head, again leaving the processor largely idle.

Finally, the most significant motivation for drive manufacturers to provide this type of execution capability is an attempt to differentiate their products by adding value-added functionality for which they can charge higher margins. This is one way to escape the low-margin commodity nature of the current drive business.

7.5 Locking

One of the basic questions when running a single application across multiple nodes is the question of concurrency control. This will be one of the key issues in the design of an Active Disk runtime system, and there are a number of promising existing solutions that would apply to an Active Disk setting.

7.5.1 Cursor Stability

For many queries, such as the decision support workloads of the TPC-D benchmark, it may be sufficient to provide Level 2 stability or *cursor stability* [Gray92]. In the context of a database system, this guarantees that a particular page does not change during execution of a particular query on that page. This ensures that any page layout or tuple layout information on the page is in a consistent state when it is processed. This is relatively straightforward for a database system, because page sizes in databases are fixed-size (and small) blocks of data. This does mean that an Active Disk system on the drives must provide a way to guarantee atomic across application-defined units, such as 4 KB or 8 KB database pages, not just on disk-level blocks, such as the 512 byte sectors which form the basic unit of atomicity in today's drives.

7.5.2 Transient Versioning

An alternative technique, since disk capacity is increased faster than disks can be filled (or accessed, recall the discussion in Chapter 2 on the disparity between the growth in capacity and the increase in transfer rates), a tradeoff can be made to use additional space in order to efficiently support more sophisticated concurrency control schemes. If space is cheap, an approach such as *transient versioning* can trade off increased disk usage against performance [Mohan92, Merchant92]. Such a scheme works by using version vectors at the disks to maintain multiple versions of the same page. The primary problem with this approach is the extra disk storage required for the copies of recently modified pages. Given today's drive capacities, this should be a feasible option. The second performance objection to this approach is the amount of metadata that must be maintained to ensure that the right block is picked by a particular query. Since drives using the NASD interface are already doing their own space management, it should be possible to add support for version vectors without too much additional effort. The specific system reported in [Mohan92] can be designed to guarantee that there never need to be more than three versions of the same block, and puts bounds on the costs of *cleaning*, or garbage collecting, versions that are no longer needed.

7.5.3 Optimistic Protocols

In the same vein as transient versioning, there are a number of schemes in *optimistic concurrency* [Amiri99, Adya99, Kung81] that would apply to an Active Disk setting. Such protocols maintain a list of blocks that a particular transaction depends on, and a list of blocks to be modified. The system then uses write-ahead logging to ensure that all drive operations are recoverable in the case that they are aborted at some later time, when a conflicting update detected. Such schemes are *optimistic* in that they assume that the normal case (the common case) is that there is no contention for a particular block and that it is easier to fix things up in the rare case when contention is detected, rather than pessimistically locking large numbers of blocks across large numbers of devices in order to synchronize a commit ordering. Using the optimistic method greatly increases the scope for concurrency and parallelism, thereby improving overall system throughput. Rather than having to wait for locks, operations can proceed on the assumption that there will not be any conflicts, and resolve any conflicts that do occur at later point.

Recent work in this area has shown that the types of functionality required within the drive runtime system to support this is minimal, and that the memory requirements can be kept small. Particularly if the amount of excess disk space used is allowed to grow reasonably large. The design of adaptive systems for concurrency control also allow a switch between optimistic and pessimistic locking when a large amount of contention and large numbers of aborts are detected [Amiri99].

7.5.4 Drive Requirements

At the most basic level, all of these methods require that the drives support reads in e.g. 8 KB pages so that applications don't see inconsistent pages. This size can be negotiated higher under control of the drive, but applications must be able to specifiy a "least acceptable unit" they can tolerate. This does not mean requests must be exclusively in this unit. With proper buffering, something more complex could be done. This allows applications such as the image processing that require entire 256 KB images at a time to do their own buffering. An open question is which level of the API will provide this support. Drive internals must provide a mechanism to guarantee update control on a page basis, otherwise an application could write inconsistent data. But this can still be as simple as knowing what the minimum consistency unit is at format time. Drive interals must be able to handle more than sector size in order to protect applications from each other. But there is no need to support arbitrary sizes.

7.6 Why This Isn't Parallel Programming

One often-heard objection to Active Disks is that this is simply parallel programming, and could just as easily be done with a massively parallel processor, or a cluster of PC nodes. The primary difference to Active Disks is the additional power for the additional cost. In parallel computing, system designers have to justify the price of the additional nodes in their speedups numbers. In practice, these speedups are usually in the range of 5x on 8 nodes, or 10x on 16 nodes and that is considered "good". With Active Disk processing, the additional processing capability comes essentially for free. If this processing power is available on every disk drive, then it would be largely unused without some manner of Active Disk capability. This means that *any* use of the processing capability on Active Disks will reduce the load on the host and improve overall performance.

In addition, as discussed above, taking advantage of Active Disks does not require re-writing an entire application to take advantage of the parallelism available. As proposed here, an Active Disk system still has a powerful front-end host processor with a large memory that is effective for many tasks that cannot be parallelized across the nodes. Effective use of Active Disks simply requires re-working of the core data-processing portions of the code to take advantage of an additional hardware feature.

The offloading of the CPU for peripheral-specific tasks is well-established. In the I/O realm, this is exemplified by the difference between SCSI and IDE drives. The use of DMA in SCSI offloads the simple portion of the work (the data transfer from disk buffers to application buffers) to the peripheral component, and reduces direct host intervention and management.

A second example is Postscript printers. For higher-level functionality, the use of Postscript as a page description language allows rendering of a page to be performed by a specialized component (which today is also simply a RISC processor programmed in C, just as this dissertation proposes for Active Disks and disk firmware). This system provides support for remote functions (Postscript code) to execute on behalf of the host. The original motivation for Postscript was to offload the work a CPU had to perform to render a page directly by providing a language to describe page layout more compactly and flexibly than simple providing a bitmap [Perry88]. Documents can now be shipped to the remote device (often "network-attached" in today's environments[1]) and offload hosts.

Note that there is a second, even more important reason why Postscript was a success. It created a level of abstraction between the description of a page and the direct workings of a particular print engine. This allowed applications to create "device independent" documents, that would provide predictable output across a range of print engines and printers. This platform independence is also important for Active Disks, even if it is not the key factor that it was for Postscript. It is useful to consider the Active Disk code as being "independent" of the number of storage devices being used. This allows the code to execute regardless of whether it is a "dumb" disks or Active Disks are being used, whether the data is striped across 4 disks, across 104 disks, in RAID1, in RAID5, and so on. This is analogous to what storage interfaces already do today, where RAID controllers and striping software all provide a "logical" interface that looks the same as the single-device interface - the recursive SCSI block interface. This is what the object interface in NASD is intended to address, and the Active Disk interface is a logical step beyond that.

It should be noted that the logical step between the NASD object interface and Active Disks is a large one because a set of additional "control structures" are being brought across the interface. An Active Disk must know that "this code is executed in a loop iterating over this entire data file" in order to be most efficient in scheduling and performing the processing. A much higher level of parallelism and a larger amount of work must be exposed in order to take full advantage of the execution capability.

1. but not secure, note the difference to NASD, where data must be protected from unauthorized discovery or alteration. This is not necessary on a printer, since the only thing that can go wrong is an "unauthorized" document is printed. Information cannot (except a few edge cases) be leaked or destroyed by an errant Postscript function. In this sense the difference is that between an "input" device such as a disk drive vs. a printer which is simply an "output" device.

7.7 Additional On-Drive Optimizations

There are a number of areas where close integration between the on-disk system and application functions benefits performance of the system. One example of this is the use of background work as described at the end of the previous chapter. We saw that it was possible to effectively take advantage of resources that would otherwise be "wasted" by having more detailed knowledge of the workload. The type of optimization described for identifying "free" blocks would only be possible if performed directly at the disk scheduler, which is the only place with the appropriate level of knowledge of particular drive characteristics (exact seek times and head settle times, for example) and the exact logical-to-physical mapping of on-drive objects. The ability of the drive scheduler to "callback" into application code at a time that is convenient from its point of view makes possible a powerful new set of abstractions.

Another area for optimization is batch scheduling of requests for the same regions of data. If there are two (or more) scans going on of the same object, these scans can be "combined" at the drive and can be satisfied in (as good as) half the time. The system can "fast forward" one scan to start at the active position of the current one and then complete the prefix when the entire object has been scanned. This type of sharing of operators is discussed in [Mehta93] in the context of a database system. They consider a batch system where there are tens or hundreds of queries to be executed at once, and relative scheduling among this entire set of queries is possible.

This type of integration in scheduling is also important in the context of storage devices that are shared among a number of hosts. Traditionally, individual disk drives have been controlled by a single host that directed all aspects of its function. The use of dual-ported drives to provide fault-tolerance has loosened this somewhat, but still depends on a particular host being the "primary" and the other the (idle until needed) "hot spare". This means that the drive still has to contend with only one "master" at a time. In the context of Storage Area Networks (SANs), this distinction no longer holds. In a Fibre Channel fabric, it is possible for a large number of hosts to make requests of the same device at the same time. The design of the devices is aimed at keeping this number reasonably small [Anderson95], but as soon as there is more than one host issuing requests, the drive must perform properly scheduling among all the hosts. In this new architecture, the individual devices are the only place where coordinated scheduling of request streams can take place. It is not practical to require that all hosts that wish access to a particular device tightly coordinate their requests. This would require a great deal of messaging among the hosts and would rapidly overcome the benefits of having network-attached storage. This is particularly true if the hosts sharing a particular device are heterogeneous - it is possible that the shared storage device(s) may be the *only* point of sharing among such hosts.

Chapter 8: The Rebirth of Database Machines

In a March 1979 special issue of *IEEE Computer,* David Hsiao of Ohio State University titled his editor's introduction "Data Base Machines Are Coming, Data Base Machines are Coming!" after a popular movie[1] of the time [Hsiao79]. In that issue, a number of articles talked about the advances in database and hardware technology that made possible the development of special-purpose hardware to support basic database management functions. The main benefit was to move processing close to the data and offload the general-purpose processors that were inefficient for data-intensive processing. The research at the time included a range of machines with varying degrees of functionality, including CASSM (content addressable segment sequential memory), RAP (relational associative processor), and RARES (rotating associative relational store) [Su79, Ozkarahan75]. A mere four years later, Haran Boral and David DeWitt of the University of Wisconsin published a paper entitled "Database Machines: An Idea Whose Time Has Passed?" [Boral83]. They examined the work before and since the *Computer* articles and concluded that the time for database machines had passed because 1) a single general-purpose host processor was sufficient to execute a scan at the full data rate of a single disk, 2) special-purpose hardware increased the design time and cost of a machine, 3) for a significant fraction of database operations, such as sorts and joins, simple hardware did not provide significant benefits.

Fortunately, the technology trends in the years since 1983 have affected all of these arguments, as discussed in the previous chapters. Aggregate storage bandwidth has dramatically improved due to the widespread use of disk parallelism in arrays with large numbers of disks. The increasing transistor count in inexpensive CMOS microchips is driving the use of microprocessors in increasingly simple and inexpensive devices. Network interfaces, peripheral adapters, digital cameras, graphics adapters, and disk drives all have microprocessors whose power exceeds the host processors of 15 years ago. The previous chapters have argued that next-generation disk drives will have this processing power in "excess" and that it can be put to good use for a range of data-intensive applications. This chapter will explore the research on database machines at that time and since. It

1. the movie was a Cold War film about Russians, not databases

will give a basic overview of a range of machine architectures, and draw parallels between the functionality and performance of these machines and the functionality proposed for Active Disk systems.

8.1 Early Machines

The first database machines all included specially-designed circuitry that carried out database primitives in conjunction with some type of rotating storage elements. Several of the machines were predicated on the imminent development of new storage technologies such as bubble memories, which did not develop as anticipated.

8.1.1 Database Machines are Coming!

The introduction of Hsiao to a special issue of *IEEE Computer* devoted to database machines is entitled "Database Machines Are Coming, Database Machines are Coming!" and lists several of the factors that made these architectures look attractive at the time [Hsiao79]. He points to benefits in reliability due to codifying particular functions in hardware and verifying those more closely, rather than depending on a monolithic mass of software. However, the main benefit is in performance, where he suggests that conventional machines are optimized for numerical computations and simple data processing, but not for concurrent access to data or for the amount of data movement required in processing large databases. He contends that these processors spend most of their time traversing layers of software rather than processing data.

He also identifies the roadblocks to such machines before that time. In particular, the immaturity of database research, where the relational model had gained respectability only recently, with Codd's 1970 paper [Codd70] on the relational model and the immaturity of the relevant hardware technologies (large associative memories, etc.).

These arguments find parallels in the Active Disk work presented in the previous chapters. The codifying of particular functions to execute on the disks also requires separating a "core" portion of the code of the database system, and gets the verification as well as the optimization benefits (see Section 6.4) of having isolated smaller portions of code. This focus on a small set of data-intensive code that can be parallelized and optimized frees up the host processor to focus on the more complex portions of the processing, such as interaction with the user, query optimization, and recovery mechanisms that are not performance-critical. Perhaps this will not directly make the entire system more reliable, as Hsiao suggests, but it can aid the developer of the database system in separating the performance-critical portions from the high-functionality pieces.

8.1.2 Content-Addressable Segment Sequential Memory (CASSM)

The CASSM (Content Addressable Segment Sequential Memory) takes the form of a fixed-head floppy disk drive with logic associated with each head [Su79]. The machine was designed to be an entire database engine, complete with its own query language, designed as a processor-per-track (PPT) machine as discussed in Section 2.1.1. The core logic associated with each track performs basic arithmetic, logical operations, and aggregation functions including *sum*, *count*, *min*, and *max*. It also contains mark bits on each track to maintain state across several phases of a search operation. One revolution is required to evaluate each condition in a search, including on pass to identify the relation that is targeted for a select. Rows are selected by setting mark bits as the conditions are checked and finally outputting the rows that pass all the tests. These mark bits are stored in a small RAM associated with the device, so they need not be written to the storage medium on each revolution [Smith79].

The basic hardware architecture of CASSM uses a number of cells, each attached to a rotating memory element. The cells are then connected by a tree of interconnects that also contains the logic gates to perform the various aggregation functions. Each cell operates as a pipeline that decodes the instruction being executed (instructions are also stored in the device), reads the data from the rotating memory, performs the requested instruction, and writes the data back to the memory. The use of a pipelined system means that each instruction takes essentially one cycle, although the overall latency is several cycles.

Data items can be marked for deletion and are then garbage collected and moved toward the "bottom" of the memory elements by a specialized algorithm to free up space. Since the data item is written on every iteration, it is possible to introduce a "stall" and leave a blank space in the memory that can be filled in with a record to be inserted at a particular location, instead of simply appending the record into the free space at the "bottom" of the memory. A single cell prototype of the hardware was demonstrated in 1976, but the multi-cell version was never built due to the complexity of the hardware, so the only evaluation was in detailed simulation [Su79].

8.1.3 Relational Associative Processor (RAP)

The RAP operates similarly, again using processor-per-track computation elements, but contains enough logic to perform *k* comparisons on each revolution [Ozkarahan75]. It also uses mark bits to maintain state across a more complex search, but these marks as stored on the media along with the data for a track, not in a separate RAM as in CASSM.

The RAP processes a specialized RAP microcode that implements a set of basic access functions that control a set of *k* comparators in each logic element. Operations include *select* to mark tuples that match a particular condition; *cross-select* that serves as the basis for joins; *read* and *save* for returning tuples to or writing them from the front-end working store; *sum*, *count*, *min*, and *max* functions for aggregation; *update* and *deletion*. The language also provides for conditional branches.

The design of the original machine allowed communication among all the individual cells, but the second generation RAP prototype explicitly eliminated support for direct cell-to-cell communication because of 1) the high cost of providing physical connections between all the cells, 2) the space requirement of providing transmission lines of sufficient length, 3) the negative impact on asynchroneity and reliability when cells can communicate with each other at will, and 4) limited need for such messaging [Schuster79].

The prototype RAP.2 system developed in 197x contains $k = 3$ comparators per logic unit. The RAP.2 hardware was completed to a prototype state including a front-end system that translated queries into RAP microcode instructions. An illustration of the select operation in this machine was provided at the beginning of Chapter 2. The basic limitation of the RAP was the small amount of comparator logic available. If more than k conditions are required for a particular query, multiple passes across the data must be made. This is particularly complex in the case of joins, where the first k keys of the inner relation are loaded and searched, then replaced with the next k, and so on. This means it will require n/k revolutions to search for all n keys. Active Disks avoid this problem by providing general-purpose programmed logic, rather than fixed comparison functions in hardware. They also provide a large associated random-access memory that can store intermediate results and tables of keys, allowing a better tradeoff of computation and number of rotations of the disk.

8.1.4 Rotating Associative Relational Store (RARES)

The RARES (Rotating Associative Relational Store) is a back-end processor and depends on a query optimizing front-end that determines the appropriate distribution of function between the front-end machine and the specialized back-end logic, which provided a selection and sort machine [Lin76, Smith79].

Due to its orthogonal data layout - rows are stored across rather than along tracks - RARES is particularly well-suited to maintaining the existing sort order of a relation. At any given point in time, all the pieces of logic will be operating on the same record, limiting the contention for the output channel when a record matches. In the other machines, several records are being compared at once, meaning that when there is contention for the output channel, they might easily be output in an order other than the one in which they were stored, thereby ruining any existing sort order among the records.

A sort in the RARES machine is performed in two phases. In the first phase, a histogram is built based on the distribution of the sort keys. On the second pass, rows are output in chunks based on this histogram. The size of individual chunks is chosen to match the amount of memory available at the front-end for performing a single-pass, in-core sort. This means that a sort operation will require $n/m+1$ revolutions where n is the number of records and m is the size of the front-end memory.

8.1.5 DIRECT

The DIRECT machine uses a set of processing elements that are not directly associated with the heads or tracks on the disk [DeWitt79]. Instead, it depends on a full crossbar switch which allows any of the processing elements to process data from any of the storage units. Storage consists of both CCD associative memory and mass storage (disk) elements. The processing elements operate in a MIMD (multiple instruction, multiple data), rather than in the SIMD (single instruction, multiple data) fashion used by the processor-per-track machines. This means that multiple queries can be active in the machine at the same time. The DIRECT machine does not use mark bits as in the previous architectures, but operates instead via temporary relations stored in the CCD memory elements. Instructions for the processing elements are compiled at the front-end as “query packets” and the architecture contains support for queuing these packets to the back-end processors. This queue can be managed in a variety of ways to maximize utilization of the back-end processors while still ensuring reasonable query response times.

8.1.6 Other Systems

The LEECH system includes support for joins that makes use of filter bits similar to those described in the Bloom Joins of the previous chapter [Smith79]. The inner relation is scanned, matching rows are output and a vector of bits is set for each matching row. This vector is then used on a second pass to scan the outer relation, again outputting matching rows, which can then be combined by the front-end for a full join result. The Data Base Computer (DBC) also did database functions in hardware. The Britton-Lee Machine was one of the more commercially popular of the database machines.

8.1.7 Survey and Performance Evaluation [DeWitt81]

A survey paper by David DeWitt and Paula Hawthorn uses analytic models of several database machine architectures to predict and compare their performance on several workloads DeWitt81]. They compare a conventional system (CS), to a processor-per-track system (PPT) with logic for every track on the disk, a processor-per-head system (PPH) with logic for every head, a processor-per-disk system (PPD) with a single processor for the entire disk, and a multi-processor cache system (MPC) where disk controllers and processing units are connected by a crossbar network and work is shared among the nodes.

Their basic conclusion is that the PPT and PPH systems perform well for selection operations in the absence of suitable indices, but that the performance of these architectures rapidly degrades with increasing output contention. The performance gap from PPT and PPH to the CS, PPD, and MPC methods is narrowed considerably by the presence of indices, as shown in Table 8-1, which shows the performance as predicted by the formulas

50,000 tuples, no index

Selectivity	CS	PPT	PPH	PPD	MPC
0.0001	11.40	0.179	0.732	6.80	6.83
0.001	11.40	0.179	0.732	6.80	6.83
0.01	11.40	0.186	0.732	6.80	6.83
0.1	11.40	0.473	0.732	6.80	6.83

50,000 tuples with index

Selectivity	CS	PPT	PPH	PPD	MPC
0.0001	0.278	0.179	0.288	0.288	0.315
0.001	0.288	0.179	0.288	0.294	0.320
0.01	0.387	0.186	0.311	0.351	0.378
0.1	1.389	0.473	0.597	0.938	0.965

Table 8-1 Predicted performance of database machines from [DeWitt81]. The tables reproduce the values in Table 4.1 and 4.2 of the 1981 DeWitt and Hawthorn paper on database machine performance. The results are the performance (in seconds) predicted for each of the five database machine architectures to perform a select with the selectivity given. Note the large performance improvement by using the processor-per-track and processor-per-disk architectures in the no index case. In the index case, the performance benefit is less dramatic, but still shows an improvement of 40% for PPT over CS and PPH.

in [DeWitt81]. We see the huge advantage of PPT and PPH for selection without an index in the first table. The second table shows that the benefit is considerably less when an index is available. The performance of PPH is now comparable with CS, although PPT is still 40 or 50% faster. At this point, with less than a factor of two advantage in performance, the authors dismiss the PPT architecture as too expensive and claim the PPH equivalent to the conventional system.

This conclusion is predicated on two factors that the original authors did not consider - the cost of random access versus sequential access to the disk, and the increasing size of databases. The values in Table 8-2 show the same comparison with these two factors taken into account.

with index and realistic disk access

Selectivity	CS	PPT	PPH	PPD	MPC
0.0001	0.317	0.179	0.326	0.326	0.353
0.001	0.327	0.179	0.326	0.332	0.359
0.01	0.426	0.186	0.329	0.390	0.417
0.1	1.418	0.473	0.597	0.967	0.994

1,000,000 tuples with index and dac

Selectivity	CS	PPT	PPH	PPD	MPC
0.0001	0.338	0.179	0.326	0.339	0.365
0.001	0.536	0.218	0.343	0.454	0.481
0.01	2.597	0.791	0.916	1.686	1.713
0.1	23.824	6.524	6.649	14.624	14.651

Table 8-2 Predicted performance of database machines with realistic disk times. The first table shows the numbers from the index case in the previous table modified to take into account a more realistic value for accessing index pages on the disk. This increases the time in both the CS and PPH systems, and makes the advantage of PPT close to a factor of two. The second table further extends these results to a more realistically sized table with one million, instead of only 50 thousand, tuples. The times of both the PPT and PPH are much less affected by the enlargement of the table than the CS, PPD, and MPC systems.

The first table recalculates the values from the previous table, which is Table 4.2 in the [DeWitt81] paper to take into account the full cost of random disk accesses when reading pages from a table through an index. The formulas used in [DeWitt81] assume that the set of pages returned from an index lookup can be read with the same efficiency as a full scan of the table - they assume simply the cost of a track switch, rather than a full seek between page accesses. The first table in Table 8-2 recalculates the values using an equation that models an average seek time for each page accessed through the index. This increases the time taken by the CS, without any effect on the PPT and relatively minor impact on the PPH performance. It does, however, increase the gap between the CS and PPT to more than a factor of two, depending on the selectivity.

If we now take into account the growth of database sizes, where a table with 50,000 tuples of 100 bytes each, as assumed by the authors in 1981, is very small, we see that the improvement from PPH and PPT is much more significant. The amount of seeking required by the CS greatly handicaps this system at high selectivity and large database size. The 50,000 tuples used in the paper represent only 5% of the capacity of the disk being modeled, where 1,000,000 is close to 100% of the capacity, leading to much longer average seeks. Using the larger database size, the second table in Table 8-2 shows that the advantage of PPT and PPH is more than a factor of four over the conventional system. Note also that this comparison does not take into account any possible caching or batching of page requests, which could reduce the number of seeks required and improve the performance somewhat in the CS and PPH cases.

The paper goes on to examine joins, and concludes that the PPT and PPH architectures will perform much more poorly than the conventional system - by a factor of three for PPT and more than a factor of ten for PPH. The basic assumption that causes this huge performance gap is the number of comparisons that can be performed by the per-track and per-head logic on a single pass. The basic comparison rate of the PPT and PPH systems is limited by the number of comparisons (*k*) that can be performed on a single revolution of the disk. When more than k comparisons must be performance to satisfy a query - for example, in a join, where the number of comparisons is proportional to the number of tuples in the inner relation - then the PPT and PPH architectures incur multiple revolutions in order to identify all the matching records. The first table in Table 8-3 shows the performance of a join using a relation with 10,000 tuples and an inner relation of 3,000 tuples. This duplicates the results presented in Table 4.3 of [DeWitt81], using the underlying equations as published in the paper[1]. The second table of Table 8-3 shows the modified numbers by varying the number of comparisons that can be performed at once, from the *k = 1* of the original paper, up to *k = 100*. We see that the PPT rapidly outperforms the CS, and that the PPH catches up once several dozen comparisons can be done at once, performing several times faster with 100 comparisons per revolution. Finally, Table 8-4 quantifies the performance differences with increasing database size. Using *k = 25* as a

10,000 tuples R, 3,000 tuples S

Selectivity	CS	PPT	PPH	PPD
0.0001	34.1	83.1	443.9	4086.5
0.001	34.1	83.1	443.9	4086.5
0.01	34.1	83.1	443.9	4086.5
0.05	34.1	103.7	443.9	4086.5
0.1	34.1	199.2	443.9	4086.5

10,000 tuples R, selectivity 0.0001

k	CS	PPT	PPH	PPD
1	34.1	83.1	443.9	4086.5
3	34.1	29.1	149.8	1364.0
10	34.1	10.2	46.9	411.2
100	34.1	2.9	7.2	43.6

Table 8-3 Predicted performance of join from [DeWitt81]. The first table reproduce the values in Table 4.3 of the 1981 DeWitt and Hawthorn paper on database machine performance. The results are the performance (in seconds) predicted for each of the five database machine architectures to perform a join of two relations R and S with the sizes and selectivity given. Both the processor-per-track and processor-per-disk architectures suffer due to the limited number (the paper assume k = 1) of parallel comparators available in the per-track or per-head logic. If this restriction is relaxed to allow additional per-track or per-head logic, then the numbers change to those in the second table, where PPT and PPH perform significantly better.

selectivity 0.0001, k = 25

Size of R	CS	PPT	PPH	PPD
10,000	34.1	5.4	20.4	166.1
50,000	187.2	5.4	72.4	800.9
100,000	402.0	5.4	137.3	1,594.4
1,000,000	5,067.1	5.4	1,306.5	15,876.9

Table 8-4 Predicted performance of joins varying by relation size. The table models the performance of a machine with *k = 25* comparators, a selectivity of 0.0001 and an increasing size of the base table. In this case, the performance advantage of PPT is clear across all the values, and the PPH is more than a factor of two faster for most table sizes.

1. the numbers do not match precisely, because the exact details of the equations used are not given and I have reconstructed the equations as close as possible from the descriptive text in the paper.

reasonable estimate, this table shows the performance of joining a table from 10,000 to one million tuples in size with the same 3,000 tuple inner relation (a variation in the relative size of these two relations would change the performance, as analyzed in the discussion of joins in Chapter 4, but only the simplest case is shown here). We see that the performance of the conventional system rapidly degrades as the size of the database increases, with the PPT performance staying constant, and the PPH always at least a factor of two better than the CS. This shows that the conclusion drawn in 1981 due to a dirth in the amount of logic available in the per-track and per-head devices, would be very different with the much greater amounts of logic that would be possible today.

8.2 Later Machines

The second generation of machines changed from purpose-built processing elements to using general-purpose processors as the core building blocks. This eliminated some of the complexities of actually designing and building hardware, and allowed more rapid turn-around of ideas for higher-level software structures.

8.2.1 An Idea Who's Time Has Passed? [Boral83]

The basic contention of Boral and DeWitt in this survey article is that 1) only 2 or 3 processors (VAX 11/780s) are required to handle the bandwidth of a single disk (Fujitsu Eagle or IBM 3380), even if they are doing projects and joins [Boral83]. Only a single processor is required for selects, and 2 or 3 for joins. This determination is based on a measured 2.5 ms for select and 13 - 22 ms for joins, with 10 ms for a sequential disk read and 30 ms for a random disk read. This is significantly better than the thousands of processors needed for a PPT system. The second contention of the article is that 2) one cannot build large databases under a PPT system, there is simply too little data per track.

Today there is 1) much more data per track (but everything is head-based, not track-based, no more fixed track disks) and 2) some increase in readout bandwidth (although not nearly as much as in capacity per track). Instead, people have followed the trends and now have 1/10 or 1/100 of one processor per disk (e.g. Digital AlphaServer 8400 with 12 processors and 520 disks). Active Disks suggest that this go back to one processor per disk. This overcomes the objection of Boral and DeWitt, because they objected mainly to multiple processors per disk (as in the PPT machines, with processors proportional to the number of tracks on the disk). In addition, the single processor proposed for Active Disks today is also much more powerful than two or three VAX 11/780s, even relative to today's higher readout bandwidths, meaning that Active Disks can easily perform the sort and join operations that were beyond the PPT machines.

The article contains additional objections in terms of the complexity of microcode for specialized database machines. This is overcome in Active Disk systems by using general-purpose processors (i.e. RISC cores) and general-purpose languages (e.g. Java). The challenge is to have people write code in a very general-purpose fashion that can then be used in an Active Disk setting just as easily as a traditional single processor or in an SMP architecture. The basic contention is that this is not that difficult - people *want* parallelism, people *can* think in terms of “small”, “mobile” functions.

8.2.2 GAMMA

The initial GAMMA prototype developed at the University of Wisconsin used seventeen VAX 11/750s acting in concert behind a single front-end processor. The second-generation prototype upgraded this hardware to an Intel iPSC/2 32-node hypercube system using Intel 386 processors. The designers identify network bottlenecks as a primary concern in the system [DeWitt90].

This system used a shared-nothing architecture and depended heavily on hash-based algorithms for easily distributing the load among the disk and processing nodes. Much of the work in GAMMA focussed on the algorithms necessary to efficiently take advantage of this type of system, including hybrid hash-joins and memory-adaptive algorithms for sort and join. Many of these algorithms are relevant to an Active Disk system and have already been mentioned in previous chapters.

The Bubba prototype was also designed as a shared-nothing system and supported a number of different architectural features with unique partitioning of function between the database and storage systems [DeWitt92].

The work of Hagmann and Ferrari summarizes a number of different architectures for partitioning function between front-end and back-end nodes. They present six different possible splits in software and provide benchmark results [Hagmann86].

8.3 Exotic Architectures

There are a number of “exotic” technologies that were proposed as the basis for database machines, none of which succeeded in displacing rotating magnetic storage as a cost-effective permanent medium. At the time, CCD storage, bubble memory, and associative memories, were considered as the basis of fast, random access devices [Hsiao79]. More recently, both RAM (as discussed in Section 2.2.2) and Flash memory have been repeatedly proposed as imminent replacements for magnetic disks, but disk technology has so far managed to stay ahead of the cost/performance curves of every contender.

Ongoing advances in the technology for MEMS-based storage offer the most compelling opportunity for a major change in storage technology [Carley99]. This technology uses micro-mechanical systems built in a silicon process to marry permanent data storage, through an underlying layer of magnetic material, with the density and manufacturing benefits of integrated circuits. A single chip is etched with thousands of tiny mechanical *tips* that can each move across a small portion of the magnetic surface and act as read/write heads. A large number of individual tips can read or write at the same time, leading to data rates much higher than today’s single-head disk drives. The use of a silicon process means that the areas surrounding the mechanical tips can be shaped into processing circuitry. This allows a single chip to contain both processing elements, memory, and permanent storage. This would be the ultimate “computer-on-a-chip” device, with all processing done close to the data.

8.4 Commercial Systems

There were a number of database machines that were marketed as commercial products, the most successful and longest-lived of which is the Content-Addressable File Store (CAFS) from ICL that was sold primarily to mainframe customers as a database “accelerator”. In addition to this, systems from Tandem and Teradata picked up many of the parallel database ideas [DeWitt92] and modern versions of these systems - based on commodity component nodes, without specialized hardware - are still being sold quite successfully today.

8.4.1 Content-Addressable File Store (CAFS and SCAFS)

The original Content-Addressable File Store (CAFS) included, among other things, support for a special piece of hardware called the file-correlation unit [Babb85] that used bit maps to assist in processing relational joins and projections, very similar to the filters used for the semi-joins in the Active Disk implementation.

The program to develop SCAFS (Son of CAFS) focussed on two primary lessons from the designers’ experience with CAFS, that 1) it should be invisible at the application level (i.e. hidden below SQL) and 2) it should be established as an industry standard solution designed for the low-cost, high-volume market.

To achieve these goals, the technology was introduced as a platform-independent library interface, called the “smart disk” interface by the database vendors, which provided access to the search functions of the SCAFS Accelerator hardware. This system used the existing query optimizer to include knowledge of the “smart disk” interface which could be chosen to execute single table select operations that would then execute within the SCAFS system. The hardware of SCAFS made use of a device with a 3.5” form factor that fit into the same SCSI enclosure as the database disks [Martin94].

Results with SCAFS show an improvement from 40% to 3x in a mixed workload (production transaction processing and decision support) and response time improvements of from 20% to 20x. In a pure decision support workload, throughput was 2 to 20 times better, and individual transaction response time improved up to 100 times. The use of SCAFS also helped systems that ordinarily benefit from heavy indexing by allowing administrators to rebalance the trade-off between the number of secondary indices and the amount of full scanning performed. This allows the production database to save the cost of maintaining multiple indices, and then depend on the Accelerator for decision support queries. The performance boost of the Accelerator reduces the need for secondary indices to achieve the same level of performance. This allows better performance on inserts while scans remain fast. The use of the Accelerator for evaluating predicates means that processor-intensive searches that require string-matching can be handled as efficiently those using numeric values.

A performance study of SCAFS in the context of a U.K. government customer [Anand95] shows that the improvement possible with the Accelerator depends heavily on the “hit” rate of the query, i.e. the *selectivity* as the term has been used in previous sections. The more data that is returned from the disk sub-system, the lower the benefit of using SCAFS - although the result is never less than a 2x improvement. They also found that the INGRES optimizer used in the study was not choosing to make use of the Accelerator as often as it could. The conclusion of the authors was that INGRES chose to use the Accelerator only if it had already decided to do a full scan of the table. It did not take the presence of the Accelerator into account when estimating the cost of other possible plans. This means that plans that used the Accelerator executed much faster than the optimizer estimated, but that other queries where use of the Accelerator would have been less globally expensive were not even considered. This means that the feedback system in the optimizer, as discussed in Section 6.3.1 is an important component of a database system optimized for Active Disks.

Chapter 9: Related Work

The basic idea of executing functions in processing elements directly attached to individual disks was explored extensively in the context of database machines, as discussed in the previous chapter. These machines fell out of favor due to the limited performance of disks at the time and the complexity of building and programming special-purpose hardware that could only handle limited functions. Instead, database research has developed large-scale, shared-nothing database servers with commodity processing elements [DeWitt92]. It has recently been suggested that the logical extension is to perform *all* application processing inside programmable system peripherals [Gray97].

9.1 Network-Attached Storage

This work on Active Disks follows from prior work at Carnegie Mellon on Network-Attached Secure Disks (NASD), which exploit the computational power at storage devices to perform parallel and network file system functions, as well as more traditional storage optimizations [Gibson97, Gibson98]. Our initial work in the area of Active Disks discusses several classes of applications that can benefit from Active Disks - including filters, multimedia, batching, and storage management - and enumerates the challenges to providing an execution environment on commodity disk drives [Riedel97].

9.1.1 Network-Attached Secure Disks

The basic goal of the NASD project is to eliminate the server bottleneck from the storage hierarchy, and make disks directly accessible to clients [Gibson97]. This eliminates the need to move all data from the disks, over a storage "network", through the memory system of a server machine, over a client network, and to the clients [Gibson98]. Management is performed asynchronously, and is only invoked for metadata management and to distribute access capabilities as required by the security system at each of the disks [Gobioff97]. The basic NASD system describes an interface for network-attachment, and for an object interface to replace the block interface of SCSI. Details of the object interface and the security system as they relate to Active Disks have already been discussed in previous chapters. Communication is via remote procedure call, and builds upon general-purpose networking protocols to allow connectivity across standard local area networks. Recent work on optimized protocols [vonEiken92, Wilkes92, vonEiken95, Benner96, Intel97] has eliminated many of the software overheads associated with general-purpose networking, and allowed systems that use more reliable underlying physical transports to reduce their processing requirements by building on the reliability of the underlying fabrics [Boden95, Horst95]. The popularity of Fibre Channel [Benner96] and Gigabit Ethernet [3Com99], which share a common physical layer protocol, may soon lead to a merger of these protocols into a single system appropriate for both storage and general-purpose network traffic. This would allow all clients to take advantage of direct communication with the storage devices.

9.1.2 Storage Area Networks (SANs)

Almost every major vendor in the storage industry has announced products or plans in the area of Storage Area Networks (SANs) [Clariion99, Seagate98, IBM99, HP98a, StorageTek99, Veritas99]. The core concept is to use a fully-connected network infrastructure, rather than direct-attached SCSI devices, to manage a collection of storage devices. There is a single network infrastructure that connects storage devices and hosts, but is still separate from the general-purpose network that connects clients and hosts. This allows multiple hosts to share the same storage, but still requires clients to access data through intermediate hosts, rather than directly. The initial products in this realm are based on Fibre Channel arbitrated loops, which are very much like SCSI except that they allow multiple initiator hosts to connect to the same device. Switched Fibre Channel is starting to become available [Brocade99] and promises support for much larger fabrics and greater scalability. This type of shared storage infrastructure provides more efficient access from multiple hosts to the same storage and makes possible device-to-device transfers without requiring all the data to traverse a host. This enables, for example, direct drive to tape backup operations [Clariion99, Legato98] or even direct drive to drive transfers.

9.1.3 Network-Attached Storage (NAS)

Products in the area of Network-Attached Storage (NAS) address the second portion of the NASD work, the question of higher-level interfaces to storage. These devices provide storage service at the distributed filesystem level through standardized protocols including NFS, HTTP, and CIFS. A number of vendors provide devices that have been specialized to perform only this function [Hitz94] and connect directly to a local area network, where they act as file servers would in a traditional setting. These are usually large, expensive devices, but smaller devices are becoming available based on commodity components and operating systems [Cobalt99]. Such devices all contain general-purpose processors that run the internal filesystem code. This makes them excellent candidates for Active Disk processing. They simply require the addition of an appropriate execution environment and a coherent programming model. In addition, such devices could make use of Active Disks internally (replacing the SCSI disks they currently use) to offload portions of their own file system processing or take advantage of additional scheduling knowledge at the disks.

9.2 Disk Drive Details

There is not much published material on the internal functioning of disk drives, as much of these details are protected by the patents or trade secrets of the various drive manufacturers. However, a number of academic studies have attempted to model the performance of disk drives and storage systems simply be observing their external behavior.

9.2.1 Local Disk Controller

An early article by Houtekamer explored the use of a local disk controller (LDC) in an IBM System/370 to replace the existing I/O subsystem [Houtekamer85]. He analyzed the performance benefits of placing a controller adjacent to each disk drive, rather than having a single controller as in the existing 370 architecture. The primary benefit was the offloading of the shared interconnect, and the additional asynchoneity and parallelism introduced when each of the disks could operate independently without holding the shared bus for the duration of a request.

These controllers used *channel programs* that are much simpler than the general-purpose programming environment proposed for Active Disks, but still allow a level of programmability at the individual devices and allow the drives to operate in parallel, only taking the shared bus when they have data to transmit. In a network-attached architecture with switched fabrics, there is no longer a shared bus, but, as previous sections have argued, the interconnect bandwidth is still a significant limitation.

9.2.2 Drive Modeling

The work of Shriver at New York University develops a very detailed model for how a disk drive performs under a particular workload [Shriver97, Shriver98]. The analytic models are driven by application traces and are highly accurate, giving agreement within 17% of the performance of real disks by adding details of prefetching and scheduling that had not previously been considered. These models have been used to develop higher-level storage management systems that predict system load and rebalance work appropriately [Borowsky96, Borowsky97, Borowsky98].

9.2.3 Drive Scheduling

The work of Worthington and Ganger at Michigan studied the benefits of using various levels of complexity in the scheduling of disk requests. They found that modeling prefetching and caching helps dramatically, but that detailed geometry information provides only marginal benefits [Worthington94]. The relation of this work to scheduling in Active Disks is discussed in more detail below.

9.3 Storage Architectures with Smarts

A number of other groups have examined issues similar to those discussed for Active Disks and proposed different types of hardware architectures and partitionings of applications.

9.3.1 Active Disks

Work at Santa Barbara and Maryland has applied Active Disk ideas to a set of applications similar to those discussed in the previous chapters, including database select, external sort, datacubes, and image processing. Their work proposes an extended-firmware model based on the block-level access of today's disks [Acharya98]. Much of this work has focussed on describing a programming model for Active Disks based on streams. Applications are designed as a number of streams that are then mapped to parallel processing on the disk nodes. The mapping of metadata is done at the host, along with all control of the overall application processing. The studies are based on extensive simulations that compare Active Disks to architectures using clusters of commodity PCs and to large SMP systems.

9.3.2 Intelligent Disks

A group at Berkeley has estimated the benefit of Active (Intelligent in their terminology) Disks for improving the performance of large SMP systems running sort, scan, and hash-join operations in a database context [Keeton98]. They estimate that decision support systems account for 35% of database server sales, and that the size of individual systems is growing by over 100% per year. Intelligent Disks are seen as a logical successor to networks or workstations, with lower cost from tighter integration and higher performance from closer coupling of storage, processing, and switched-based serial interconnect fabrics. This work uses an analytic model to estimate the benefits of Intelligent Disks in a decision support environment where all computation is performed by intelligent storage elements.

Previous work by members of this group has studied the performance of relational database code on modern multiprocessors in the context of a transaction processing workload [Keeton98a]. A group at Rice and Compaq has provided a similar analysis for both decision support and transaction processing workloads [Ranganathan98]. Both of these studies focus strictly on the detailed processor performance, seeing the behavior of the input/output system as secondary.

9.3.3 SmartSTOR

Work at IBM Almaden and Berkeley analyzed the performance of the TPC-D decision support queries and found that single table acceleration was insufficient, especially in the context of database systems that make heavy use of pre-aggregation and summary tables [Hsu99]. This work proposes that using processing elements per-disk will not be cost-effective, and suggests the use of front-end units with more powerful processing and memory resources to manage a number of underlying, “dumb” disk drives.

Recent changes to the TPC-D benchmark definition have led to a split of this benchmark into two separate benchmarks: TPC-R, which allows the use of summary tables and serves as a benchmark for a “reporting” workload, and TPC-H which contains more stringent requirements against pre-aggregation and re-establishes the benchmark's original focus on ad-hoc queries [TPC99f, TPC99g]. The use of ad-hoc queries, where query patterns are generally not known beforehand, more closely mirrors the way large systems are used in practice and derives much less benefit from pre-aggregates than the well-structured reporting workloads of TPC-R. Ad-hoc queries will benefit much more from the aggregate power of Active Disks, even with relatively low processing power on the individual disks, as described in detail in Chapter 5.

9.4 Parallel Programming

The parallel programming community has long sought a basic set of primitives that all programmers could use, or an automatic method for parallelizing code without direct help from the user. These efforts have met with only mixed success as it can be difficult to identify (much less eliminate) serial dependencies among code that was not written with parallelism in mind.

9.4.1 Scan Primitives

The work of Blelloch at Carnegie Mellon explored a number of primitives that were implemented in the NESL parallel programming language and allow the programmer to express explicit parallelism in their computation. These primitives were specified in the context of a functional language, thereby simplifying a number of the problems using legacy code written in procedural languages such as C [Blelloch89].

9.4.2 Data and Task Parallelism

A considerable body of work has explored the parallelization of applications across the nodes of both massively-parallel machines [JaJa92] and networks of workstations [Subhlok93]. This has been done both by parallelizing compilers [HPF93] and by application-specific libraries optimized for parallel execution [Dongarra79, Blackford97]. The partitioning of applications across hosts and Active Disks is similar to the parallelization of applications in general, although the nature of I/O make it both easier and more difficult. I/O operations are usually much more coarse-grained than array accesses that parallel compilers must distribute. Programmers are already familiar with the idea of I/O being done in largish blocks and via a relatively small set of interfaces (read, write, open, etc.) which should aid isolation of I/O units. On the other hand, most applications are coded to deal with a "sequential" model of file access, rather than operating in parallel, so partitioning into concurrently executing portions may be more complex than what is currently done for parallel array accesses. A body of work on parallel I/O has also explored the distribution of function across compute nodes and I/O nodes in massively parallel machines [Corbett93, LoVerso93] and particular parallel I/O interfaces [Nieuwejaar95].

The identification of parallelism across different tasks in a computation (task parallelism), rather than simply across all the elements in a distributed array (data parallelism), introduces some additional complexity. Such systems allow different code elements to run at different times within the same architecture, and take advantage of the structure of the computation to re-map to different portions of the underlying machine as appropriate [Gross94, Stricker95]. The effectiveness of this approach was demonstrated in a number of specific problem domains, including vision [Webb93], earthquake modeling [O'Hallaron98], and air pollution modeling [Segall95]. This work in programming models and algorithms for automatic placement of function [Yang93]can be used by Active Disks to properly partition applications and expose the available parallelism.

9.5 Parallel I/O

A number of projects have addressed the optimization of I/O in a system with many parallel nodes.

9.5.1 Disk-Directed I/O

The disk-directed I/O work of Kotz shows that providing an aggregate description of a large amount of work to an I/O node (or perhaps across a number of nodes) allows it (them) to schedule the work of the entire request and exhibits significant performance gain over executing the work as a series of simple requests [Kotz94]. This includes functions such as scatter/gather operations that distribute data across a large number of clients (collective I/O) or a drive-to-drive copy controlled at the drives, rather than through the client.

9.5.2 Automatic Patterns

Work by Madhyastha at the University of Illinois has focussed on the identification of patterns of I/O access in a running system. When a particular, known pattern is detected, the system adjusts its behavior to better match this pattern (e.g. increasing or decreasing the amount of cache memory used, or changing how aggressively prefetching is done). The use of a neural network allows the system to "learn" additional patterns and strategies for optimization. This is done through a modification of the runtime system which "observes" the behavior of running application. It does not require any changes to the applications [Madhyastha96, Madhyastha97].

Complementary work at Maryland on optimizing I/O in a variety of parallel applications has found large benefits with relatively small changes to the original code [Acharya96]. These two approaches can be seen as both competitive and complementary. The classification work assumes that applications are not modified, and allows the system to adapt to the request streams generated by the applications. The optimization work assumes that the applications are somehow "broken" in their use of parallel I/O and believe that the applications should be modified to match the characteristics of the underlying system. This has the advantage that the parallelism can now be made explicit, giving the runtime system exact knowledge of what is going on, rather than the "guesses" of the automatic system. On the other hand, this type of optimization tends to specialize the code for a particular system architecture. If this code is then moved to a significantly different architecture with different performance characteristics, it will need to be modified again.

The ideal system would allow the parallelism to be made explicit, without specializing to a particular machine.

9.6 Data Mining and OLTP

One of the performance advantages of Active Disks discussed in Chapter 5 was the use of integrated scheduling at the individual disk drives to combine a "background" workload that can take advantage of the characteristics of a particular "foreground" workload to share resources more efficiently. The most obvious example of this is the combination of a decision support workload and a transaction processing workload. This allows decision makers to identify and evaluate patterns in the database while the system continues to process new transactions. The closer this connection is, the more up-to-date and relevant decisions can be. Chapter 5 proposed a system where these decision support queries can be performed against the "live" production system. This extends previous work in mixed database workloads, and in disk scheduling.

9.6.1 OLTP and DSS

Previous studies of combined OLTP and decision support workloads on the same system indicate that the disk is the critical resource [Paulin97]. Paulin observes that both CPU and memory utilization is much higher for the Data Mining workload than the OLTP, which is also clear from the design of the decision support system shown in Table 5-13 in Section 5.4.2 of Chapter 5. In his experiments, all system resources are shared among the OLTP and decision support workloads with an impact of 36%, 70%, and 118% on OLTP response time when running decision support queries against a heavy, medium, and light transaction workload, respectively. The author concludes that the primary performance issue in a mixed workload is the handling of I/O demands on the data disks, and suggests that a priority scheme is required in the database system as a whole to balance the two types of workloads.

9.6.2 Memory Allocation

Brown, Carey and DeWitt [Brown92, Brown93] discuss the allocation of memory as the critical resource in a mixed workload environment. They introduce a system with multiple workload classes, each with varying response time goals that are specified to the memory allocator. They show that a modified memory manager is able to successfully meet these goals in the steady state using 'hints' in a modified LRU scheme. The modified allocator works by monitoring the response time of each class and adjusting the relative amount of memory allocated to a class that is operating below or above its goals. The scheduling scheme we propose here for disk resources also takes advantage of multiple workload classes with different structures and performance goals. In order to properly support a mixed workload, a database system must manage all system resources and coordinate performance among them.

9.6.3 Disk Scheduling

Existing work on disk scheduling algorithms [Denning67,..., Worthington94] shows that dramatic performance gains are possible by dynamically reordering requests in a disk queue. One of the results in this work indicates that many scheduling algorithms can be performed equally well at the host [Worthington94]. The scheme that we propose here takes advantage of additional flexibility in the workload (the fact that requests for the background workload can be handled at low priority and out of order) to expand the scope of reordering possible in the disk queue. Our scheme also requires detailed knowledge of the performance characteristics of the disk (including exact seek times and overhead costs such as settle time) as well as detailed logical-to-physical mapping information to determine which blocks can be picked up for free. This means that this scheme would be difficult, if not impossible, to implement at the host without close feedback on the current state of the disk mechanism. This makes it a compelling use of additional "smarts" directly at the disk.

With the advent of Storage Area Networks (SANs), storage devices are being shared among multiple hosts performing different workloads [HP98a, IBM99, Seagate98, Veritas99]. As the amount and variety of sharing increases, the only central location to optimize scheduling across multiple workloads will be directly on the devices themselves.

9.7 Miscellaneous

There are several areas of research that have explored "activeness" in other contexts, placing general-purpose computation outside the domain of traditional microprocessors. There have also been significant advances in the commercial deployment of small-footprint execution environments that can be used in very resource-constrained environments.

9.7.1 Active Pages

The Active Pages work at the University of California at Davis proposes computation directly in memory elements, moving parallel computation to the data [Oskin98]. Their architecture is based on a memory system where RAM is integrated with some amount of reconfigurable logic. Results from a simulator promise performance up to 1000 times that of conventional systems, which often cannot keep their processors fed with data due to limitations in bandwidth and parallelism. This work takes advantage of the same silicon technology trends as Active Disks, but must operate at a much lower granularity than the parallelism of Active Disk operations.

The authors suggest that the partitioning between the computation performed in the processor and in the Active Pages can be done by a compiler that takes into account bandwidth, synchronization, and parallelism to determine the optimal location for any piece of code. For Active Pages, this scheduling would have to be done at the instruction or basic block level due to the tight coupling between the processor and the Active Pages. For Active Disks, this scheduling would be done at the module or component level, as discussed in the previous sections, since the coupling is much lower and the “distance” between Active Disks and the host is much larger.

9.7.2 Active Networks

The Active Networks project provides the inspiration for the name Active Disks[1] and proposes a mechanism for running application code at network routers and switches to accelerate innovation and enable novel applications in the movement of data and network management [Tennenhouse96]. This work suggests two possible approaches for managing network programs - a *discrete* approach that allows programs to be explicitly loaded into the network and affect the processing of future packets and an *integrated* approach in which each packet consists of a program instead of simply “dumb” data. The tradeoff between the two is the amount of state that devices can be expected to maintain between requests and how many requests can be active at any given time. The implementation of the Active IP option [Wetherall96] describes a prototype language system and an API to access router state and affect processing. It does not address the resource management issues inherent in allowing these more complex programs.

These types of functions are much more sensitive to execution time than Active Disk functions. Network packets within IP switches are processed at rates of gigabits per second, while Active Disks have the “advantage” of being limited on one side by the (low) performance of the mechanical portions of the disks. This also means that the resource management system for Active Disks must only take into account a small number of concurrently running functions at the disks, while Active Network switches might easily have thousands of concurrent processing streams.

9.7.3 Small Java

There has been considerable work on optimizing safe languages such as Java through the use of just-in-time compilation [Gosling96, Grafl96] or translation [Proebsting97]. Small-footprint Java implementations are becoming available for embedded devices due to the popularity of the language and the promise of portability among hardware platforms. Recent product announcements promise a Java virtual machine in 256K of ROM [HP98] or as tiny as a smart card that provides a Java virtual machine in 4K of ROM and can run bytecode programs up to 8K in size for a significant subset of the language [Schlumberger97]. This demonstrates that it is possible to implement a workable subset of the Java virtual machine in a very limited resource environment. Other systems such as Inferno [Inferno97] are specifically targeted for embedded, low-resource environments and might also be appropriate choices for Active Disk execution.

---

1. The name was originally suggested by Jay Lepreau from the University of Utah in October 1996 during a question at the OSDI work in progress session where the original work on Network-Attached Secure Disks, later published as [Gibson97] was being presented.

Chapter 10: Conclusions and Future Work

The continued increase in performance and decrease in cost of processors and memory are causing system "intelligence" to move from CPUs to specialized system peripherals. In the context of storage systems, designers have been using this trend to perform more complex optimizations inside individual devices. To date, these optimizations have been limited by the relatively low-level nature of today's storage protocols. At the same time, trends in storage density, mechanics, and electronics are eliminating the bottlenecks to moving data off the storage media and putting pressure on interconnects and hosts to move data more efficiently as it is processed further "upstream". The ability to execute application code directly at storage devices allows processing to be performed close to the data; enables application-aware scheduling; and makes possible more complex and specialized operations than a general-purpose storage interface would normally support.

This dissertation has demonstrated an important class of applications that will see significant gains - in many cases linear scaling in the number of devices added to the system - from the use of Active Disks. These applications take advantage of the parallelism in large storage systems to greatly increase the total computational power available to them, and circumvent the limited interconnect bandwidth in these systems, greatly increasing the apparent data rate from storage. An analytic model for estimating traditional server and Active Disk performance predicts the speedups possible given a simple set of application characteristics. A prototype Active Disk system with up to 10 disks realizes speedups of more than a factor of two over a comparable traditional server. This system should easily scale to speedups of more than 10x in reasonably-sized systems similar to those already in use for large databases today.

Emerging applications such as data mining, multimedia feature extraction, and approximate searching involve ever-larger data sets, on the order of 100s of GB or TB, and justify large numbers of Active Disks. Many of these applications have the characteristics that make them attractive for execution across Active Disks. This dissertation has described a set of compelling example applications from these domains and measured their performance in the prototype system. In addition, the preceding chapters have shown that all of the core functions of a relational database system can be implemented effectively in the context of Active Disks, with dramatic performance improvements on a benchmark decision support workload.

10.1 Contributions

This work makes several contributions to the understanding and analysis of storage and database systems:

- the basic concept of *Active Disks*, proposing the use of excess processing power on commodity storage devices to execute application-level code, rather than simply optimizing within a single, strictly-defined storage interface
- a *validated performance model* that predicts the performance of an application in an Active Disk system given a few basic characteristics of the application and the underlying hardware
- a *description of how to adapt* data-intensive applications from database, data mining, and multimedia to Active Disks
- the *choice of appropriate algorithms* for performing all the core functions of a relational database system and the development of a small set of on-disk primitives
- evaluation of a *prototype system* to demonstrate the benefits promised by the performance model and show that the code changes required to take advantage of Active Disks are feasible and straightforward
- the *modification of a relational database system* to use Active Disks and show dramatic improvements on portions of an industry-standard benchmarks for such systems
- the demonstration of a *novel approach to disk scheduling* that allows the combination of application-level knowledge and drive-specific information that can only be performed directly at the drives and promises significant performance improvements on mixed database workloads
- the identification of an additional advantage in *code specialization* made possible by extracting a “core” portion of an application’s processing and mapping it to a particular, well-known architecture for optimization
- the *discussion* of previous research on database machines and its relevance to the design of Active Disk systems today

all of which support the claims made in the thesis of this work that was presented in the introduction: that data-intensive applications can take advantage of computational power available directly at storage devices to improve their overall performance, more effectively balance their consumption of system-wide resources, and provide functionality that would not otherwise be available.

These points also serve to answer the major objections to Active Disks and illustrate that this is an architecture with novel cost/performance tradeoffs, that significant performance benefits are possible, and that these benefits are attainable with straightforward modifications to existing applications. In addition, the dissertation has introduced two areas of further optimization, in disk scheduling and code specialization that are made possible by partitioning applications in this way.

10.2 Future Work

There are a number of areas to be explored before the benefits presented here can be fully put into practice. Providing a safe environment for application code inside the drive in order to both protect the integrity of data on the drive and ensure proper function in the presence of misbehaved application code is critical. One of the key benefits of the SCSI interface to today's disks is that it is easily understood and easily evaluated or "certified". By introducing greater variety in the functions that can be executed by the storage devices, this simplicity of analysis will suffer. The specific limits that will be required to ensure continued "reliable" operation of these systems are still unclear. The issue of resource management becomes considerably more complex as computations becomes more distributed. Active Disks will need to make more complex scheduling decisions than disk drives do today, but they also open many new areas for optimization by exploiting the much richer interfaces they make possible.

By demonstrating the need for and benefits of a programmable interface at these devices, this work opens the way for applications and uses far beyond what has been discussed here. By providing what is simply a *capability* that others can build on, Active Disks open up a range of new possibilities and research areas.

10.2.1 Extension of Database Implementation

There are several additional areas of specialization within the context of existing database systems. The prototype does not attempt to optimize index-based scans, for example, although benefits in scheduling and re-ordering of disk requests are certainly possible. The system described here also does not attempt any optimization when writing data. There are numerous possibilities for optimizing the layout of data as it is written, or for re-organizing data as information about access patterns and usage becomes available. Combining such knowledge with integrated scheduling directly at the disks, should open up a number of optimizations of the 10% and 25% variety that are not the orders of magnitude promised by parts of this work, but are very acceptable in much-studied, and commercially important, transaction processing systems.

10.2.2 Extension to File Systems

If the benefits of the database structure and presence of the query optimizer could be extended to a more general filesystem interface, then more applications could take advantage of this type of system. The basic tradeoff is a more structured way of managing data (fixed-size pages, explicit schemas) and limited types of operations (a set of basic operations, operators, and data types) than simply treating filesystem objects as “bags of bits”. The more information on explicit typing and primitive “operators” that is available, the better a runtime system will be able to optimize a particular function or application, as is done with queries in the relational database system, which is built on a well-defined core model. This type of information on “structure” is the key to being able to automatically partition, distribute, and parallelize processing. How to “discover” it where it exists automatically, or impose it where it does not, is an open question.

10.2.3 Pervasive Device Intelligence

Advances in technology such as MEMS-based storage open up a range of new options for processing coupled directly with data. The use of micro-mechanical systems promises the density and capacity of magnetic storage, along with the form factor and manufacturing process advantages of silicon. This makes possible a single chip that contains both magnetic media for permanent storage, computation elements, and memory. The issues in how to program a massive collection of such components, that can be embedded in a huge range of individual devices, or aggregated into very high-density “blocks” of computation and storage are only beginning to be understood. Such devices will break the normal paradigms for developing applications, and will require a much more data-centric model of computation than is commonly used today. The partitioning of applications for Active Disks as discussed here is a first step in that direction, but the possibilities of such pervasive devices are much larger.

10.2.4 The Data is the Computer

The processing of large volumes of data will continue to become more important as more and more of the world’s data is digitized and stored. The number of daily transactions and events that will soon be tracked (and later analyzed in the search for patterns) is enormous. Companies and individuals are just beginning to realize the possibilities opened up when massive amounts of data, of huge variety, can be easily searched and analyzed. These data sets will not be the structured types that succumb easily to indexing and pre-aggregation, but will demand high throughput and flexibility in storage systems and access methods. The parallelism and flexibility required will again change the nature of application development, and will need to build on a new set of storage and processing primitives that can be combined in highly parallel and distributed ways.

References

[3Com99] 3Com Corporation “Gigabit Ethernet Comes of Age” *Technology White Paper*, June 1999.

[Abbott93] Abbott, M.B. and Peterson, L.L. “Increasing Network Throughput by Integrating Protocol Layers” *IEEE Transactions On Networking* 1 (5), October 1993.

[Acharya96] Acharya, A., Uysal, M., Bennett, R., Mendelson, A., Beynon, M., Hollingsworth, J., Saltz, J. and Sussman, A. “Tuning the Performance of I/O-Intensive Parallel Applications” *IOPADS*, May 1996.

[Acharya98] Acharya, A., Uysal, M. and Saltz, J. “Active Disks” *ASPLOS*, October 1998.

[Adl-Tabatabai96] Adl-Tabatabai, A., Langdale, G., Lucco, S. and Wahbe, R. “Efficient and Language-Independent Mobile Programs” *PLDI*, May 1996.

[Adya97] Adya, A. and Liskov, B. “Lazy Consistency Using Loosely Synchronized Clocks” *ACM PODC*, August 1997.

[Adya99] Adya, A. “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions” *PhD Thesis*, MIT Laboratory for Computer Science, March 1999.

[Agrawal95] Agrawal, R. and Srikant, R. “Fast Algorithms for Mining Association Rules” *VLDB*, September 1994.

[Agrawal96] Agrawal, R. and Schafer, J. “Parallel Mining of Association Rules” *IEEE Transactions on Knowledge and Data Engineering* 8 (6), December 1996.

[Almaden97] Almaden CattleCam, IBM Almaden Research Center *www.almaden.ibm.com/almaden/cattle/home_cow.htm*, January 1998.

[Amiri99] Amiri, K., Gibson, G. and Golding, R. “Scalable Concurrency Control and Recovery for Shared Storage Arrays” *Technical Report CMU-CS-99-111*, February 1999.

[Anand95] Anand, S.S., Bell, D.A. and Hughes, J.G. “Experiences using the Ingres Search Accelerator for a Large Property Management Database System” *ICL Systems Journal* 10 (1), May 1995.

[Anderson91] Anderson, T.E., Levy, H.M., Bershad, B.N. and Lazowska, E.D. “The Interaction of Architecture and Operating System Design,” *ASPLOS*, September 1991.

[Anderson95] Anderson, D., Seagate Technology, Personal Communication, 1995.

[Anderson98] Anderson, D. “Seagate’s Ideas of What Active Disks Might Be Like“ *Oakland NASD Workshop: What is to be done with lots of computation in storage?*, May 1998.

[Anderson99] Anderson, D., Seagate Technology, Personal Communication, April 1999.

[ANSI86] ANSI, “Small Computer System Interface (SCSI) Specification”, ANSI X3.131, 1986.

[ANSI93] ANSI, “Information technology - Small Computer System Interface - 2” ANSI X3T9.2/375D Working Group, *www.t10.org/drafts*, September 1993.

[ARM98] Advanced RISC Machines Ltd., “ARM Processors and Peripherals” *www.arm.com/Pro+Peripherals*, November 1998.

[ARM99] Advanced RISC Machines Ltd., “ARM Powered Products” *www.arm.com/Markets/ARMapps*, September 1999.

[Arpaci-Dusseau97] Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Culler, D.E., Hellerstein, J.M. and Patterson, D.A. “High-Performance Sorting on Networks of Workstations” *SIGMOD*, June 1997.

[Arya94] Arya, M., Cody, W., Faloutsos, C., Richardson, J. and Toga, A. "QBISM: Extending a DBMS to Support 3d Medical Images" *International Conference on Data Engineering*, February 1994.

[Babb85] Babb, E. “CAFS File-Correlation Unit” *ICL Technical Journal* 4 (4), November 1985.

[Baker91] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W. and Ousterhout, J.K. “Measurements of a Distributed File System” *SOSP*, October 1991.

[Baker92] Baker, M., Asami, S., Deprit, E., Ousterhout, J.K. and Seltzer, M.I. “Non-Volatile Memory for Fast, Reliable File Systems” *ASPLOS*, 1992.

[Barclay97] Barclay, T. “The TerraServer Spatial Database” *www.research.microsoft.com/terraserver*, November 1997.

[Benner96] Benner, A.F. *Fibre Channel: Gigabit Communications and I/O for Computer Networks*, McGraw Hill, 1996.

[Bennett91] Bennett, J.M., Bauer, M.A. and Kinchlea, D. “Characteristics of Files in NFS Environments” *ACM Symposium on Small Systems*, 1991.

[Berchtold96] Berchtold, S., Keim, D.A. and Kriegel, H. “The X-tree: An Index Structure for High-Dimensional Data” *VLDB*, 1996.

[Berchtold97] Berchtold, S., Boehm, C., Keim, D.A. and Kriegel, H. “A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space” *ACM PODS*, May 1997.

[Bernstein81] Bernstein, P.A. and Goodman, N. “Power of Natural Semijoins” *SIAM Journal on Computing* 10 (4), 1981.

[Bershad95] Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M.E., Becker, D., Chambers, C. and Eggers, S. “Extensibility, Safety, and Performance in the SPIN Operating System” *SOSP*, December 1995.

[Bhatti95] Bhatti, N. and Schlichting, R.D. “A System for Constructing Configurable High-Level Protocols” *SIGCOMM*, August 1995.

[Birrell80] Birell, A.D. and Needham, R.M. “A Universal File Server” *IEEE Transactions on Software Engineering* 6 (5), September 1980.

[Birrell93] Birrell, A.D., Hisgen, A., Jerian, C., Mann, T. and Swart, G. “The Echo Distributed File System” *Research Report 111*, DEC Systems Research Center, September 1993.

[Bitton88] Bitton, D. and Gray, J. “Disk Shadowing” *VLDB*, 1988.

[Blackford97] Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R.C. *ScaLAPACK User’s Guide*, SIAM, May 1997.

[Blelloch89] Blelloch, G.E. “Scans as Primitive Parallel Operations” *IEEE Transactions on Computers* 38 (11), November 1989.

[Blelloch97] Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J. and Zagha, M. “An Experimental Analysis of Parallel Sorting Algorithms” *Communications of the ACM*, To Appear.

[Boral83] Boral, H. and DeWitt, D.J. “Database Machines: An Idea Whose Time Has Passed?” *International Workshop on Database Machines*, September 1983.

[Borowsky96] Borowsky, E., Golding, R., Merchant, A., Shriver, E., Spasojevic, M. and Wilkes, J. “Eliminating Storage Headaches Through Self-Management” *OSDI*, October 1996.

[Borowsky97] Borowsky, E., Golding, R., Merchant, A., Schreier, L., Shriver, E., Spasojevic, M. and Wilkes, J. “Using Attribute-Managed Storage to Achieve QoS” *5th International Workshop on Quality of Service*, June 1997.

[Borowsky98] Borowsky, E., Golding, R., Jacobson, P., Merchant, A., Schreier, L., Spasojevic, M. and Wilkes, J. “Capacity Planning With Phased Workloads” *WOSP*, October 1998.

[Brocade99] Brocade Communications Systems “Brocade Announces New Line of Entry-Level Fibre Channel Switches” *News Release*, November 1999.

[Brown92] Brown, K., Carey, M., DeWitt, D., Mehta, M. and Naughton, J. “Resource Allocation and Scheduling for Mixed Database Workloads” *Technical Report*, University of Wisconsin, 1992.

[Brown93] Brown, K., Carey, M. and Livny, M. “Managing Memory to Meet Multiclass Workload Response Time Goals” *VLDB*, August 1993.

[Cabrera91] Cabrera, L. and Long, D., “Swift: Using Distributed Disk Striping to Provide High I/O Data Rates” *Computing Systems* 4 (4), Fall 1991.

[Cao94] Cao, P., Lim, S.B., Venkataraman, S. and Wilkes, J. “The TickerTAIP Parallel RAID Architecture” *ACM Transactions on Computer Systems* 12 (3), August 1994.

[Carey94] Carey, M.J. DeWitt, D.J., Franklin, M.J., Hall, N.E., McAuliffe, M.L., Naughton, J.F., Schuh, D.T., Solomon, M.H., Tan, C.K., Tsatalos, O.G., White, S.J. and Zwilling, M.J. “Shoring Up Persistent Applications” *SIGMOD*, 1994.

[Carley99] Carley, L.R., Bain, J.A., Fedder, G.K., Greve, D.W., Guillou, D.F., Lu, M.S.C., Mukherjee, T., Santhanam, S., Abelmann, L., and Min, S. “Single Chip Computers with MEMS-based Magnetic Memory” *44th Annual Conference on Magnetism and Magnetic Materials*, November 1999.

[Chankhunthod96] Chankhunthod, A., Danzig, P.B., Neerdaels, C., Schwartz, M.F. and Worrell, K.J. “A Hierarchical Internet Object Cache” *USENIX Technical Conference*, January 1996.

[Chaudhuri96] Chaudhuri, S. and Shim, K. “Optimization of Queries with User-defined Predicates” *VLDB*, 1996.

[Chaudhuri97] Chaudhuri, S. and Dayal, U. “An Overview of Data Warehousing and OLAP Technology” *SIGMOD Record* 26 (1), March 1997.

[Chen93] Chen, J.B. and Bershad, B. “The Impact of Operating System Structure on Memory System Performance” *SOSP*, December 1993.

[Cirrus98] Cirrus Logic, Inc. “New Open-Processor Platform Enables Cost-Effective, System-on-a-chip Solutions for Hard Disk Drives” *www.cirrus.com/3ci*, June 1998.

[Clariion99] Clariion Storage Division, Data General Corporation “Data General’s Clariion Storage Division Unveils Industry’s First Full Fibre Channel Storage Area Network Solution” *News Release*, May 1999.

[Clark90] Clark, D.D. and Tennenhouse, D.L. “Architectural Considerations for a New Generation of Protocols” *SIGCOMM*, September 1990.

[Cobalt99] Cobalt Networks “Cobalt Networks Delivers Network Attached Storage Solution with the New NASRaQ” *News Release*, March 1999.

[Codd70] Codd, E.F. “A Relational Model for Data for Large Shared Data Banks” *Communications of the ACM* 13 (6), 1970.

[Consel98] Consel, C., Hornof, L., Lawall, J., Marlet, R., Muller, G., Noyé, J., Thibault, S., Volanschi, E.-N. “Tempo: Specializing Systems Applications and Beyond” *ACM Computing Surveys - Symposium on Partial Evaluation* 30 (3), September 1998.

[Corbett93] Corbett, P.F., Baylor, S.J. and Feitelson, D.G. “Overview of the Vesta Parallel File System” *IPPS Workshop for I/O in Parallel and Distributed Systems*, April 1995.

[Corbis99] Corbis Images “Corbis Images Launches Digital Fine Art Gallery in Response to Professional Demand” *News Release*, August 1999.

[Cuppu99] Cuppu, V., Jacob, B., Davis, B. and Mudge, T. “A Performance Comparison of Contemporary DRAM Architectures” *ISCA*, May 1999.

[Dahlin95] Dahlin, M.D. et al. “A Quantitative Analysis of Cache Policies for Scalable Network File Systems” *SOSP*, December 1995.

[Dahlin95a] Dahlin, M. “Serverless Network File Systems” *PhD Thesis*, University of California - Berkeley, December 1995.

[Dar96] Dar, S., Franklin, M.J., Jonsson, B.P., Srivastava, D. and Tan, M. “Semantic Data Caching and Replacement” *VLDB*, September 1996.

[deJonge93] deJonge, W., Kaashoek, M.F. and Hsieh, W.C. “The Logical Disk: A New Approach to Improving File Systems” *SOSP*, December 1993.

[Denning67] Denning, P.J. “Effects of Scheduling on File Memory Operations” *AFIPS Spring Joint Computer Conference*, April 1967.

[Dennis66] Dennis, J.B. and Van Horn, E.C. “Programming Semantics for Multiprogrammed Computations” *Communications of the ACM* 9 (3), 1966.

[DeWitt79] DeWitt, D.J. “DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems” *IEEE Transactions on Computers* 28 (6), June 1979.

[DeWitt81] DeWitt, D.J. and Hawthorn, P.B. “A Performance Evaluation of Database Machine Architectures” *VLDB*, September 1981.

[DeWitt84] DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M. and Wood, D. “Implementation Techniques for Main Memory Database Systems” *SIGMOD*, June 1984.

[DeWitt85] DeWitt, D.J. and Gerber, R. “Multiprocessor Hash-Based Join Algorithms” *VLDB*, 1985.

[DeWitt90] DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H. and Rasmussen, R. “The Gamma Database Machine Project” *TKDE* 2 (1), 1990.

[DeWitt91] DeWitt, D.J., Naughton, J.F. and Schneider, D.A. “Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting” *PDIS*, 1991.

[DeWitt92] DeWitt, D.J. and Gray, J. “Parallel Database Systems: The Future of High Performance Database Processing” *Communications of the ACM* 36 (6), June 1992.

[DeWitt93] DeWitt, D.J., Naughton, J.F., and Burger, J. “Nested Loops Revisited” *PDIS*, 1993.

[DiskTrend99] Disk/Trend News “1999 Rigid Disk Drive Report” *News Release*, May 1999.

[Dongarra79] Dongarra, J.J., Bunch, J.R., Moler, C.B. and Stewart, G.W. *LINPACK Users’ Guide*, SIAM, 1979.

[Douceur99] Douceur, J.R. and Bolosky, W.J. “A Large-Scale Study of File-System Contents” *SIGMETRICS*, May 1999.

[Drapeau94] Drapeau, A.L., Shirriff, K.W., Hartman, J.H., Miller, E.L., Seahan, S., Katz, R.H., Lutz, K., Patterson, D.A., Lee, E.K. and Gibson, G.A. “RAID-II: A High-Bandwidth Network File Server” *ISCA*, 1994.

[Druschel93] Druschel, P. and Peterson, L.L. “Fbufs: A High-Bandwidth Cross-Domain Transfer Facility” *SOSP*, December 1993.

[Elphick96] Elphick, M. “Trimming disk-drive chip count” *Computer Design*, December 1996.

[Engler95] Engler, D.R., Kaashoek, M.F., O'Toole, J. “Exokernel: An Operating System Architecture for Application-Level Resource Management” *SOSP*, December 1995.

[English92] English, R.M. and Stepanov, A.A. “Loge: a Self-Organizing Disk Controller” *Winter USENIX*, January 1992.

[Faloutsos94] Faloutsos, C., Barber, R., Flickner, M., Hafner, J., Niblack, W., Petkovic, D. and Equitz, W. “Efficient and Effective Querying by Image Content” *Journal of Intelligent Information Systems* 3 (4), July 1994.

[Faloutsos96] Faloutsos, C. *Searching Multimedia Databases by Content*, Kluwer Academic Inc., 1996.

[Fayyad98] Fayyad, U. “Taming the Giants and the Monsters: Mining Large Databases for Nuggets of Knowledge” *Database Programming and Design*, March 1998.

[Fayyad99] Fayyad, U., Microsoft Research, Personal Communication, May 1999.

[Flickner95] Flickner, M., Sawhney, H., Niblack, W., Ashley, J., Huang, Q., Dom, B., Gorkani, M., Hafner, J., Lee, D., Petkovic, D., Steele, D. and Yanker, P. “Query by Image and Video Content: the QBIC System” *IEEE Computer*, September 1995.

[Ford96] Ford, B. and Susarla, S. “CPU Inheritance Scheduling” *OSDI*, October 1996.

[Franklin96] Franklin, M.J., Jonsson, B.P. and Kossmann, D. “Performance Tradeoffs for Client-Server Query Processing” *SIGMOD*, June 1996.

[Ganger98] Ganger, G.R., Worthington, B.L. and Patt, Y.N. “The DiskSim Simulation Environment Version 1.0 Reference Manual” *Technical Report*, University of Michigan, February 1998.

[Garcia-Molina92] Garcia-Molina, H. and Salem, K. “Main Memory Database Systems: An Overview” *TKDE* 4 (6), December 1992.

[Gibson92] Gibson, G. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, MIT Press, 1992.

[Gibson97] Gibson, G., Nagle, D., Amiri, K., Chang, F.W., Feinberg, E., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D. and Zelenka, J. “File Server Scaling with Network-Attached Secure Disks” *SIGMETRICS*, June 1997.

[Gibson97a] Gibson, G., Nagle, D., Amiri, K., Chang, F.W., Gobioff, H., Riedel, E., Rochberg, D., and Zelenka, J. “Filesystems for Network-Attached Secure Disks” *Technical Report CMU-CS-97-112*, Carnegie Mellon University, March 1997.

[Gibson98] Gibson, G., Nagle, D., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. “A Cost-Effective, High-Bandwidth Storage Architecture” *ASPLOS*, October 1998.

[Gobioff97] Gobioff, H., Gibson, G. and Tygar, D. “Security for Network Attached Storage Devices” *Technical Report CMU-CS-97-185*, Carnegie Mellon University, October 1997.

[Golding95] Golding, R., Shriver, E., Sullivan, T. and Wilkes, J. “Attribute-Managed Storage” *Workshop on Modeling and Specification of I/O*, October 1995.

[Golding95a] Golding, R., Bosch, P., Staelin, C., Sullivan, T. and Wilkes, J. “Idleness is not sloth” *USENIX Technical Conference*, 1995.

[Gosling96] Gosling, J., Joy, B. and Steele, G. *The Java Language Specification*, Addison-Wesley, 1996.

[Goyal96] Goyal, P., Guo, X. and Vin, H.M. “A Hierarchical CPU Scheduler for Multimedia Operating Systems” *OSDI*, October 1996.

[Graefe95] Graefe, G. and Cole, R.L. “Fast Algorithms for Universal Quantification in Large Databases” *ACM Transactions on Database Systems* 20 (2), June 1995.

[Grafl96] Grafl, R. “Cacao: Ein 64bit JavaVM Just-In-Time Compiler” *Master’s Thesis*, University of Vienna, 1996.

[Gray92] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, September 1992.

[Gray95] Gray, J., Bosworth, A., Layman, A. and Pirahesh, H. “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals” *Technical Report MSR-TR-95-22*, Microsoft Research, November 1995.

[Gray97] Gray, J. “What Happens When Processing, Storage, and Bandwidth are Free and Infinite?” *IOPADS Keynote*, November 1997.

[Gray97a] Gray, J. “Sort Benchmark Home Page” *www.research.microsoft.com/barc/SortBenchmark*, 1997.

[Grochowski96] Grochowski, E.G. and Hoyt, R.F. “Future Trends in Hard Disk Drives” *IEEE Transactions on Magnetics* 32 (3), May 1996.

[Gross94] Gross, T., O’Hallaron, D. and Subhlok, J. “Task Parallelism in a High Performance Fortran Framework” *IEEE Parallel & Distributed Technology* 3, 1994.

[Guha98] Guha, S., Rastogi, R. and Shim, K. “CURE: An Efficient Clustering Algorithm for Large Databases” *SIGMOD*, 1998.

[Hagmann86] Hagmann, R.B. and Ferrari, D. “Performance Analaysis of Several Back-End Database Architectures” *ACM Transactions on Database Systems* 11 (1), March 1986.

[Harinarayan96] Harinarayan, V., Rajaraman, A. and Ullman, J.D. “Implementing Data Cubes Efficiently” *SIGMOD*, June 1996.

[Hartman93] Hartman, J.H. and Ousterhout, J.K. “The Zebra Striped Network File System” *SOSP*, December 1993.

[Hartman96] Hartman, J., Manber, U., Peterson, L. and Proebsting, T. “Liquid Software: A New Paradigm for Networked Systems” *Technical Report 96-11*, University of Arizona, 1996.

[Hawthorn81] Hawthorn, P.B. and DeWitt, D.J. “Performance Analysis of Alternative Database Machine Architectures” *IEEE Transactions on Software Engineering* SE-8 (2), January 1982.

[Hitz94] Hitz, D., Lau, J. and Malcolm, M. “File Systems Design for an NFS File Server Appliance”, *Winter USENIX*, January 1994.

[Horst95] Horst, R.W. “TNet: A Reliable System Area Network” *IEEE Micro*, February 1995.

[Houtekamer85] Houtekamer, G.E. “The Local Disk Controller” *SIGMETRICS*, 1985.

[Howard88] Howard, J.H. et al. “Scale and Performance in a Distributed File System” *ACM TOCS* 6 (1), February 1988.

[HP98] Hewlett-Packard Company “HP Offers Virtual-Machine Technology to Embedded-Device Market” *News Release*, March 1998.

[HP98a] Hewlett-Packard Company “HP to Deliver Enterprise-Class Storage Area Network Management Solution” *News Release*, October 1998.

[HPF93] High Performance Fortran Forum *High Performance Fortran Language Specification, Version 1.0,* May 1993.

[Hsiao79] Hsiao, D.K. “DataBase Machines Are Coming, DataBase Machines Are Coming!” *IEEE Computer*, March 1979.

[Hsu99] Hsu, W.W., Smith, A.J. and Young, H.C. “Projecting the Performance of Decision Support Workloads on Systems with Smart Storage (SmartSTOR)” *Technical Report CSD-99-1057*, University of California - Berkeley, August 1999.

[IBM99] IBM Corporation and International Data Group “Survey says Storage Area Networks may unclog future roadblocks to e-Business” *News Release*, December 1999.

[Illman96] Illman, R. “Re-engineering the Hardware of CAFS” *ICL Systems Journal* 11 (1), May 1996.

[Inferno97] "Inferno: Tomorrow's Full Service OS Today" *News Release*, November 1997.

[Intel97] Intel Corporation “Virtual Interface (VI) Architecture” *www.viarch.org*, December 1997.

[Jagadish94] Jagadish, H.V., Lieuwen, D.F., Rastogi, R., Silberschatz, A. and Sudarshan, S. “Dali: A High Performace Main Memory Storage Manager” *VLDB*, 1994.

[JaJa92] JaJa, J. *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[Katz92] Katz, R.H. “High Performance Network- and Channel-Attached Storage” *Proceedings of the IEEE* 80 (8), August 1992.

[Keeton98] Keeton, K., Patterson, D.A. and Hellerstein, J.M. “A Case for Intelligent Disks (IDISKs)” *SIGMOD Record* 27 (3), August 1998.

[Keeton98a] Keeton, K., Patterson, D.A., He, Y.Q., Raphael, R.C. and Baker, W.E. “Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads” *ISCA*, June 1998.

[Kim86] Kim, M.Y. “Synchronized disk interleaving” *IEEE Transactions on Computers* C-35 (11), November 1986.

[Kitsuregawa83] Kitsuregawa, M., Tanaka, H. and Moto-Oka, T. “Application of Hash To Data Base Machine and Its Architecure” *New Generation Computing* 1, 1983.

[Knuth79] Knuth, D.E. *The Art of Computer Programming - Volume 3*, Addison-Wesley, 1979.

[Korn98] Korn, F., Labrinidis, A., Kotidis, Y. and Faloutsos, C. “Ratio Rules: A New Paradigm for Fast, Quantifiable Data Mining” *VLDB*, August 1998.

[Kotz94] Kotz, D. “Disk-directed I/O for MIMD Multiprocessors” *OSDI*, November 1994.

[Kung81] Kung, H.T. and Robinson, J.T. “On Optimistic Methods for Concurrency Control” *ACM Transactions on Database Systems* 6 (2), June 1981.

[Lammers99] Lammers, D. “Cost Crunch Creates Push for Single-chip Drive” *EETimes Online*, May 1999.

[Lee96] Lee, E.K. and Thekkath, C.A. “Petal: Distributed Virtual Disks” *ASPLOS*, October 1996.

[Legato98] Legate Systems “Legato Systems Announces Immediate Support for Storage Networks” *News Release*, August 1998.

[Levin99] Levin, R. “Java Technology Comes of Age” *News Feature*, May 1999.

[Li95] Li, Z. and Ross, K.A. “PERF Join: An Alternative to Two-way Semijoin and Bloomjoin” *CIKM*, 1995.

[Lin76] Lin, C.S., Smith, D.C.P. and Smith, J.M. “The Design of a Rotating Associative Memory for Relational Database Applications” *ACM Transactions on Database Systems* 1 (1), March 1976.

[Livny87] Livny, M. “Multi-disk management algorithms” *ACM SIGMETRICS*, May 1987.

[Locke98] Locke, K. “Storage Squeeze” *Software Magazine*, January 1998.

[Long94] Long, D.D.E., Montague, B.R., and Cabrera, L., “Swift/RAID: A Distributed RAID System,” *Computing Systems* 7 (3), Summer 1994.

[Mackert86] Mackert, L.F. and Lohman, G.M. “R* Optimizer Validation and Performance Evaluation for Distributed Queries” *VLDB*, 1986.

[Madhyastha96] Madhyastha, T.M. and Reed, D.A.”Intelligent, Adaptive File System Policy Selection” *Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[Madhyastha97] Madhyastha, T.M. and Reed, D.A. “Exploiting Global Access Pattern Classification” *SC’97*, November 1997.

[Mangione98] Mangione, C. “Performance Tests Show Java as Fast as C++” *JavaWorld*, February 1998.

[Marlet99] Marlet, R., Thibault, S. and Consel, C. “Efficient Implementations of Software Architectures via Partial Evaluation” *Journal of Automated Software Engineering* 6 (4), October 1999.

[Martin94] Martin, M.W. “The ICL Search Accelerator, SCAFS: Functionality and Benefits” *ICL Systems Journal* 9 (2), November 1994.

[Martin99] Martin, R.P. and Culler, D.E. “NFS Sensitivity to High Performance Networks” *SIGMETRICS*, May 1999.

[Massalin89] Massalin, H, and Pu, C. “Threads and Input/Output in the Synthesis Kernel” *SOSP*, December 1989.

[McGraw97] McGraw, G. and Felten, E.W. *Java Security: Hostile Applets, Holes, and Antidotes*, John Wiley & Sons, 1997.

[Mehta93] Mehta, M., Soloviev, V. and DeWitt, D.J. “Batch Scheduling in Parallel Database Systems” *Data Engineering*, 1993.

[Mehta93a] Mehta, M. and DeWitt, D.J. “Dynamic Memory Allocation for Multiple-Query Workloads” *VLDB*, 1993.

[Merchant92] Merchant, A., Wu, K., Yu, P.S. and Chen, M. “Performance Analysis of Dynamic Finite Versioning for Concurrency Transaction and Query Processing” *SIGMETRICS*, June 1992.

[Mohan92] Mohan, C., Pirahesh, H. and Lorie, R. “Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions” *SIGMOD*, June 1992.

[Mosberger96] Mosberger, D. and Peterson, L.L. “Making Paths Explicit in the Scout Operating System” *OSDI*, October 1996.

[Mowry96] Mowry, T.C., Demke, A.K. and Krieger, O. “Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications” *OSDI*, October 1996.

[Necula96] Necula, G.C. and Lee, P. “Safe Kernel Extensions Without Run-Time Checking” *OSDI*, October 1996.

[Nelson88] Nelson, M.N., Welch, B.B. and Ousterhout, J.K. “Caching in the Sprite Network File System”, *ACM TOCS* 6 (1), February 1988.

[Nieuwejaar95] Nieuwejaar, N. and Kotz, D. “Low-level Interfaces for High-level Parallel I/O” *IPPS ‘95 Workshop for I/O in Parallel and Distributed Systems*, April 1995.

[Nyberg94] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J. and Lomet, D. “AlphaSort: A RISC Machine Sort” *SIGMOD*, May 1994.

[O’Hallaron98] O’Hallaron, D.R., Shewchuk, J.R. and Gross, T. “Architectural Implications of a Family of Irregular Applications” *HPCA*, February 1998.

[Oskin98] Oskin, M., Chong, F.T. and Sherwood, T. “Active Pages: A Computation Model for Intelligent Memory” *ISCA*, 1998.

[Ousterhout85] Ousterhout, J.K., DaCosta, H., Harrison, D., Kunze, J.A., Kupfer, M. and Thompson, J.G. “A Trace Drive Analysis of the UNIX 4.2 BSD FIle System” *SOSP*, December 1985.

[Ousterhout91] Ousterhout, J.K., “Why Aren’t Operating Systems Getting Faster As Fast As Hardware?” *Summer USENIX*, June 1991.

[Ozkarahan75] Ozkarahan, E.A., Schuster, S.A. and Smith, K.C. “RAP - An Associative Processor for Data Base Management” *Proceedings of AFIPS NCC* 44, 1975.

[Ozkarahan86] Ozkarahan, E. *Database Machines and Database Management*, Prentice-Hall, 1986.

[Pang93] Pang, H., Carey, M.J. and Livny, M. “Partially Preemptible Hash Joins” *SIGMOD*, May 1993.

[Pang93a] Pang, H. Carey, M.J. and Livny, M. “Memory-Adaptive External Sorting” *VLDB*, August 1993.

[Pasquale94] Pasquale, J. and Anderson, E. “Container Shipping: Operating System Support for I/O-Intensive Applications” *IEEE Computer* 27, March 1994.

[Patterson88] Patterson, D.A., Gibson, G. and Katz, R.H., “A Case for Redundant Arrays of Inexpensive Disks” *SIGMOD*, June 1988.

[Patterson95] Patterson, R.H. et al. “Informed Prefetching and Caching” *SOSP*, 1995.

[Paulin97] Paulin, J. “Performance Evaluation of Concurrent OLTP and DSS Workloads in a Single Database System” *Master’s Thesis*, Carleton University, November 1997.

[Perry88] Perry, Tekla S. “‘PostScript’ prints anything: a case history” *IEEE Spectrum*, May 1988.

[PostgreSQL99] PostgreSQL DBMS, *www.postgresql.org*, February 1999.

[Prabhakar97] Prabhakar, S., Agrawal, D., Abbadi, A.E., Singh, A. and Smith, T. “Browsing and Placement of Images on Secondary Storage” *IEEE International Conference of Multimedia Computer Systems*, June 1997.

[Proebsting97] Proebsting, T.A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T. and Watterson, S.A. “Toba: Java For Applications A Way Ahead of Time Compiler” *Technical Report TR97-01*, University of Arizona, January 1997.

[Pu95] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J. and Zhang, K. “Optimistic Incremental Specialization: Streamlining a Commercial Operating System”, *SOSP*, December 1995.

[Quest97] Quest Project, IBM Almaden Research Center “Quest Data Mining Project” *www.almaden.ibm.com/cs/quest*, December 1997.

[Ramakrishnan98] Ramakrishnan, R. *Database Management Systems*, McGraw-Hill, 1998.

[Ranganathan98] Ranganathan, P., Gharachorloo, K., Adve, S.V. and Barroso, L.A. “Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors” *ASPLOS*, October 1998.

[Riedel96] Riedel, E. and Gibson, G. “Understanding Customer Dissatisfaction With Underutilized Distributed File Servers” *Fifth Goddard Conference on Mass Storage Systems and Technologies*, September 1996.

[Riedel97] Riedel, E. and Gibson, G. “Active Disks - Remote Execution for Network-Attached Storage” *Technical Report CMU-CS-97-198*, Carnegie Mellon University, December 1997.

[Riedel98] Riedel, E., Gibson, G. and Faloutsos, C. “Active Storage For Large-Scale Data Mining and Multimedia” *VLDB*, August 1998.

[Riedel98a] Riedel, E., van Ingen, C. and Gray, J. “Sequential I/O Performance in Windows NT” *2nd USENIX Windows NT Symposium*, August 1998.

[Romer96] Romer, T.H., Lee, D., Voelker, G.M., Wolman, A., Wong, W.A., Baer, J., Bershad, B.N. and Levy, H.M. “The Structure and Performance of Interpreters” *ASPLOS*, October 1996.

[Rosenblum91] Rosenblum, M. and Ousterhout, J.K., “The Design and Implementation of a Log-Structured File System” *SOSP*, 1991.

[Ruemmler91] Ruemmler, C. and Wilkes, J., “Disk Shuffling”, *Technical Report HPL-CSP-91-30*, Hewlett-Packard Labs, 1991

[Ruemmler93] Ruemmler, C. and Wilkes, J. “UNIX disk access patterns” *Winter USENIX*, January 1993.

[Ruemmler94] Ruemmler, C. and Wilkes, J. “An Introduction to Disk Drive Modeling” *IEEE Computer* 27 (3), March 1994.

[Sachs94] Sachs, M.W., Leff, A., and Sevigny, D., “LAN and I/O Convergence: A Survey of the Issues”, *IEEE Computer*, December 1994.

[Salzberg90] Salzberg, B., Tsukerman, A., Gray, J., Uern, S. and Vaughan, B. “FastSort: A distributed single-input single-output external sort” *SIGMOD*, May 1990.

[Satya81] Satyanarayanan, M. “A Study of File Sizes and Functional Lifetimes” *SOSP*, December 1981.

[Schlumberger97] Schlumberger Limited “First-Ever Java-Based Smart Card Demonstrated by Schlumberger” *News Release*, April 1997.

[Schmidt95] Schmidt, F. *The SCSI Bus and IDE Interface: Protocols, Applications and Programming*, Addison-Wesley, April 1995.

[Schneider89] Schneider, D.A. and DeWitt, D.J. “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment” *SIGMOD*, 1989.

[Schneider90] Schneider, D.A. and DeWitt, D.J. “Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines” *VLDB*, 1990.

[Schuster79] Schuster, S.A., Nguyen, H.B., Ozkarahan, E.A. and Smith, K.C. “RAP.2 - An Associative Processor for Databases and Its Applications” *IEEE Transactions on Computers* 28 (6), June 1979.

[Seagate97] Seagate Technology, Inc. “Cheetah: Industry-Leading Performance for the Most Demanding Applications”, *News Release*, 1998.

[Seagate98] Seagate Technology, Inc. “Storage Networking: The Evolution of Information Management” *White Paper*, November 1998.

[Seagate98a] Seagate Technology “Fibre Channel: The Preferred Performance Path” *White Paper*, November 1998.

[Segall95] Segall, E., Riedel, E., Bruegge, B., Steenkiste, P. and Russell, A. “Heterogeneous Distributed Environmental Modeling” *SIAM* 29 (1), 1995.

[Selinger79] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A. and Price, T.G. “Access Path Selection in a Relational Database Management System” *SIGMOD*, 1979.

[Seltzer96] Seltzer, M., Endo, Y., Small, C. and Smith, K. “Dealing With Disaster: Surviving Misbehaved Kernel Extensions” *OSDI*, October 1996.

[Senator95] Senator, T.E., Goldberg, H.G., Wooten, J., Cottini, M.A., Khan, A.F.U., Klinger, C.D., Llamas, W.M., Marrone, M.P. and Wong, R.W.H. “The Financial Crimes Enforcement Network AI System (FAIS): Identifying potential money laundering from reports of large cash transactions” *AI Magazine* 16 (4), Winter 1995.

[Sha94] Sha, L., Rajkumar, R. and Sathaye, S.S. “Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems” *Proceedings of the IEEE* 82 (1), January 1994.

[Shekita90] Shekita, E. and Zwilling, M. “Cricket: A Mapped Persistent Object Store” *Persistent Object Sytems Workshop*, September 1990.

[Shriver97] Shriver, E. “Performance Modeling for Realistic Storage Devices” *PhD Thesis*, New York University, May 1997.

[Shriver98] Shriver, E., Merchant, A. and Wilkes, J. “An analytic behavior model for disk drives with readahead caches and request reordering” *SIGMETRICS*, June 1998.

[Shugart87] Shugart, Al “5 1/4-in. drive replaces minifloppy with reliable Winchester” *Electronic Design*, April 1987.

[Siemens97] Siemens Microelectronics, Inc. “Siemens’ New 32-bit Embedded Chip Architecture Enables Next Level of Performance in Real-Time Electronics Design” *News Release*, September 1997.

[Siemens98] Siemens Microelectronics, Inc. “Siemens Announced Availability of TriCore-1 For New Embedded System Designs” *News Release*, March 1998.

[Sienknecht94] Sienknecht, T.F., Friedrich, R.J., Martinka, J.J. and Friedenbach, P.M. “The Implications of Distributed Data in a Commercial Environment on the Design of Hierarchical Storage Management” *Performance Evaluation* 20 (1-3), 1994.

[Sirer96] Sirer, E.G., Savage, S., Pardyak, P., DeFouw, G.P. and Bershad, B.N. “Writing an Operating System in Modula-3” *Workshop on Compiler Support for System Software*, February 1996.

[Small95] Small, C. and Seltzer, M. "A Comparison of OS Extension Technologies" *USENIX Technical Conference*, January 1996.

[Smirni96] Smirni, E., Aydt, R.A., Chien, A.A., and Reed, D.A. "I/O Requirements of Scientific Applications: An Evolutionary View" *Fifth IEEE Conference on High Performance Distributed Computing*, August 1996.

[Smith79] Smith, D.C.P. and Smith, J.M. “Relational DataBase Machines” *IEEE Computer*, March 1979.

[Smith94] Smith, K.A. and Seltzer, M.I. “File Layout and File Systems Performance” *Technical Report TR-35-94*, Harvard University, 1994.

[Smith95] Smith, S.M. and Brady, J.M. “SUSAN - A New Approach to Low Level Image Processing” *Technical Report TR95SMS1c*, Oxford University, 1995.

[Spasojevic93] Spasojevic, M. and Satyanarayanan, M. “A Usage Profile and Evaluation of a Wide-Area Distributed File System” *Winter USENIX*, 1993.

[Spasojevic96] Spasojevic, M. and Satyanarayanan, M. “An Empirical Study of a Wide-Area Distributed File System” *ACM TOCS*, May 1996.

[Srivastava94] Srivastava, A., and Eustace, A. “ATOM: A system for building customized program analysis tools” *Technical Report TN-41*, Digital Western Research Lab, 1994.

[Steere99] Steere, D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C. and Walpole, J. “A Feedback-driven Proportion Allocator for Real-Rate Scheduling” *OSDI*, February 1999.

[Stonebraker86] Stonebraker, M. and Rowe, L.A. “The Design of Postgres” *SIGMOD*, May 1986.

[Stonebraker97] Stonebraker, M. “Architectural Options for Object-Relational DBMSs” *White Paper*, Informix Software, Inc., 1997.

[StorageTek94] Storage Technology Corporation “Iceberg 9200 Storage System: Introduction”, *STK Part Number 307406101*, 1994.

[StorageTek99] Storage Technology Corporation “StorageTek Takes Lead In Worldwide SAN Deployments” *News Release*, July 1999.

[Stricker95] Stricker, T., Stichnoth, J., O’Hallaron, D., Hinrichs, S. and Gross, T. “Decoupling Synchronization and Data TRansfer in Message Passing Systems of Parallel Computers” *International Conference on Supercomputing*, July 1995.

[Su79] Su, S.Y.W., Nguyen, L.H., Emam, A. and Lipovski, G.J. “The Architectural Features and Implementation Techniques of the Multicell CASSM” *IEEE Transactions on Computers* 28 (6), June 1979.

[Subhlok93] Subhlok, J., Stichnoth, J.M., O’Hallaron, D.R. and Gross, T. “Exploiting Task and Data Parallelism on a Multicomputer” *ACM Symposium on Principle & Practices of Parallel Programming*, May 1993.

[Subhlok94] Subhlok, J., O’Hallaron, D., Gross, T., Dinda, P. and Webb, J. “Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs” *Supercomputing ‘94*, November 1994.

[Sun98] Sun Microsystems “Sun Enterprise 10000 Server (Starfire)” *Technical White Paper*, September 1998.

[Sun99] Sun Microsystems “Sun Unleashes Jini Connection Technology” *News Release*, January 1999.

[Sun99a] Sun Microsystems “Cathay Pacific Airways Builds Data Warehouse - Key Customer Information System Relies on Sun Hardware” *Customer Success Story*, June 1999.

[Szalay99] Szalay, A.S. and Brunner, R.J. “Astronomical Archives of the Future: A Virtual Observatory” *Future Generation Computer Systems*, In Press, 1999.

[Tennenhouse96] Tennenhouse. D.L., et al. “A Survey of Active Network Research” *SIGOPS*, 1996.

[TPC93] Transaction Processing Performance Council, *Quarterly Report* 6, July 1993.

[TPC97a] TPC-C Rev. 3.2 Rating for an HP NetServer LX Pro C/S, Transaction Processing Performance Council, *www.tpc.org*, January 1997.

[TPC97b] TPC-C Rev. 3.2 Rating for a Dell PowerEdge 6100, Transaction Processing Performance Council, *www.tpc.org*, March 1997.

[TPC97c] TPC-C Rev. 3.3 Rating for a Digital AlphaServer 1000A 5/500, Transaction Processing Performance Council, *www.tpc.org,* April 1997.

[TPC97d] TPC-C Rev. 3.2.3 Rating for an IBM RS/6000 Workgroup Server F50 C/S, Transaction Processing Performance Council, *www.tpc.org,* April 1997.

[TPC97e] TPC-C Rev. 3.3 Rating for an IBM RS/6000 Enterprise Server J50 C/S, Transaction Processing Performance Council, *www.tpc.org,* May 1997.

[TPC97f] TPC-C Rev. 3.3.2 Rating for an HP 9000 V2200 Enterprise Server C/S, Transaction Processing Performance Council, *www.tpc.org*, September 1997.

[TPC98] Transaction Processing Performance Council, “TPC Benchmark D (Decision Support) Standard Specification 1.3.1”, *www.tpc.org*, February 1998.

[TPC98a] TPC-D Rev. 1.3.1 Rating for a Digital AlphaServer 8400 5/625 12 CPUs using Oracle8, Transaction Processing Performance Council, *www.tpc.org*, May 1998.

[TPC98b] TPC-C Rev. 3.3 Rating for a Compaq ProLiant 5500-6/400 C/S, Transaction Processing Performance Council, *www.tpc.org,* September 1998.

[TPC99a] TPC-C Rev. 3.4 Rating for an HP NetServer LH 4r C/S, Transaction Processing Performance Council, *www.tpc.org*, January 1999.

[TPC99b] TPC-C Rev. 3.4 Rating for an IBM Netfinity 7000 M10 C/S, Transaction Processing Performance Council, *www.tpc.org*, January 1999.

[TPC99c] TPC-C Rev. 3.4 Rating for a Dell PowerEdge 6350 C/S, Transaction Processing Performance Council, *www.tpc.org,* March 1999.

[TPC99d] TPC-C Rev. 3.4 Rating for an HP 9000 N4000 Enterprise Server C/S, Transaction Processing Performance Council, *www.tpc.org*, April 1999.

[TPC99e] TPC-C Rev. 3.4 Rating for an IBM RS/6000 Enterprise Server H70 C/S, Transaction Processing Performance Council, *www.tpc.org*, May 1999.

[TPC99f] Transaction Processing Performance Council, “TPC Benchmark H (Decision Support) Standard Specification 1.2.1”, *www.tpc.org*, June 1999.

[TPC99g] Transaction Processing Performance Council, “TPC Benchmark R (Decision Support) Standard Specification 1.2.0”, *www.tpc.org*, June 1999.

[Turley96] Turley, J. “ARM Grabs Embedded Speed Lead” *Microprocessor Reports* 2 (10), February 1996.

[VanMeter96] Van Meter, R., Holtz, S. and Finn G. “Derived Virtual Devices: A Secure Distributed File System Mechanism”, *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies,* September 1996.

[Veritas99] Veritas Software Corporation “Veritas Software and Other Industry Leaders Demonstrate SAN Solutions” *News Release*, May 1999.

[Virage98] Virage “Media Management Solutions” *www.virage.com*, February 1998.

[Volanschi96] Volanschi, E.-N., Muller, G. and and Consel, C. “Safe Operating System Specialization: The RPC Case Study” *Workshop on Compiler Support for System Software*, February 1996.

[vonEicken92] von Eicken, T., Culler, D., Goldstein, S.C. and Schauser, K. “Active Messages: A Mechanism for Integrated Communication and Computation” *ISCA*, May 1992.

[vonEicken95] von Eicken, T., Basu, A., Buch, V. and Vogels, W. “U-Net: A User-Level Network Interface for Parallel and Distributed Computing” *SOSP*, December 1995.

[vonNeumann63] von Neumann, J. *Collected Works*, Pergamon Press, 1963.

[Wactlar96] Wactlar, H.D., Kanade, T., Smith, M.A. and Stevens, S.M. “Intelligent Access to Digital Video: Informedia Project” *IEEE Computer*, May 1996.

[Wahbe93] Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L. "Efficient Software-Based Fault Isolation" *SOSP*, December 1993.

[Waldspurger94] Waldspurger, C.A. and Weihl, W.E. “Lottery Scheduling: Flexible Proportional-Share Resource Management” *OSDI*, November 1994.

[Wang99] Wang, R.Y., Anderson, T.E. and Patterson, D.A. “Virtual Log Based File Systems for a Programmable Disk” *OSDI*, February 1999.

[Webb93] Webb, J. “Latency and Bandwidth Consideration in Parallel Robotic Image Processing” *Supercomputing ‘93*, November 1993.

[Welling98] Welling, J. “Fiasco: A Package for fMRI Analysis” *www.stat.cmu.edu/~fiasco*, January 1998.

[Wetherall96] Wetherall, D.J. and Tennenhouse, D.L. “The ACTIVE IP Option” *ACM SIGOPS European Workshop*, September 1996.

[Widom95] Widom, J. “Research Problems in Data Warehousing” *CIKM*, November 1995.

[Wilkes79] Wilkes, M.V. and Needham, R.M. *The Cambridge CAP Computer and Its Operating System*, 1979.

[Wilkes92] Wilkes, J. “Hamlyn - an interface for sender-based communications” *Technical Report HPL-OSR-92-13*, Hewlett-Packard Laboratories, November 1992.

[Wilkes95] Wilkes, J., Golding, R., Staelin, C. and Sullivan, T. “The HP AutoRAID hierarchical storage system” *SOSP*, December 1995.

[Worthington94] Worthington, B.L., Ganger, G.R. and Patt, Y.N. “Scheduling Algorithms for Modern Disk Drives” *SIGMETRICS*, May 1994.

[Worthington95] Worthington, B.L., Ganger, G.R., Patt, Y.N. and Wilkes, J. “On-Line Extraction of SCSI Disk Drive Parameters” *SIGMETRICS*, May 1995.

[Yang93] Yang, B., Webb, J., Stichnoth, J., O’Hallaron, D. and Gross, T. “Do & Merge: Integrating Parallel Loops and Reductions” *Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[Yao85] Yao, A.C. and Yao, F.F. “A General Approach to D-Dimensional Geometric Queries” *ACM STOC*, May 1985.

[Zeller90] Zeller, H. and Gray, J. “An Adaptive Hash Join Algorithm for Multiuser Environments” *VLDB*, 1990.

[Zhang97] Zhang, T., Ramakrishnan, R. and Livny, M. “BIRCH: A New Data Clustering Algorithm and Its Applications” *Data Mining and Knowledge Discovery* 1 (2), 1997.

Appendix A: Benchmark Details

This appendix contains a summary description of the tables and queries for the TPC-D benchmark, as used by the Active Disk prototype.

1.1 Details of TPC-D Queries and Schemas

A listing of the schemas for the tables used in the TPC-D benchmark, as well as the full SQL text of the queries discussed in Chapters 4, 5, and 6.

1.1.1 Tables

The `lineitem` table is the largest table in the benchmark (a factor of 5x larger than the next-largest table) and contains a listing of:

`l_orderkey`	identifier
`l_partkey`	identifier
`l_suppkey`	identifier
`l_linenumber`	integer
`l_quantity`	decimal
`l_extendedprice`	decimal
`l_discount`	decimal
`l_tax`	decimal
`l_returnflag`	char, 1
`l_linestatus`	char, 1
`l_shipdate`	date
`l_commitdate`	date
`l_receiptdate`	date
`l_shipinstruct`	char, 25
`l_shipmode`	char, 10
`l_comment`	varchar, 44

for each item sold by the company.

The `order` table is the next largest table and contains a listing of:

o_orderkey	identifier
o_custkey	identifier
o_orderstatus	char, 1
o_totalprice	decimal
o_orderdate	date
o_orderpriority	char, 15
o_clerk	char, 15
o_shippriority	integer
o_comment	varchar, 79

for each order processed.

The `part` table contains a listing of:

p_partkey	identifier
p_name	varchar, 55
p_mfgr	char, 25
p_brand	char, 10
p_type	varchar, 25
p_size	integer
p_container	char, 10
p_retailprice	decimal
p_comment	varchar, 79

for each unique part in the database.

The `supplier` table contains a listing of:

s_suppkey	identifier
s_name	char, 25
s_address	varchar, 40
s_nationkey	identifier
s_phone	char, 15
s_acctbal	decimal
s_comment	varchar, 101

for each supplier of parts.

The `partsupp` table contains a listing of:

ps_partkey	identifier
ps_suppkey	identifier
ps_availqty	integer
ps_supplycost	decimal
ps_comment	varchar, 199

matching parts and suppliers.

The `customer` table contains a listing of:

`c_custkey`	identifier
`c_name`	char, 25
`c_address`	varchar, 40
`c_nationkey`	identifier
`c_phone`	char, 15
`c_acctbal`	decimal
`c_mktsegment`	char, 10
`c_comment`	varchar, 117

for each customer.

The `nation` table contains a listing of:

`n_nationkey`	identifier
`n_name`	char, 25
`n_regionkey`	identifier
`n_comment`	varchar, 152

for placing countries in geographic regions

The `region` table contains a listing of:

`r_regionkey`	identifier
`r_name`	char, 25
`r_comment`	varchar, 152

for each order processed. There are only a small number of countries and regions in the database, so both of these tables are very small.

Finally, the optional `time` table is used to map dates to date strings for systems that do not handle such conversions internally:

`t_timekey`	date
`t_alpha`	char, 10
`t_year`	integer
`t_month`	integer
`t_week`	integer
`t_day`	integer

for each unique date that appears in the database.

1.1.2 Query 1 - Aggregation

The business question for Query 1 is to provide a summary report of all the items shipped as of a particular date. This date is chosen within 60 and 120 days of the end date in the database, so about 95% of the data items must be scanned to answer this query. Several items are summarized, including total list price, total amount charged, average price, and average discount. The query text as used in the PostgreSQL prototype is:

```
select l_returnflag, l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from lineitem
where l_shipdate <= '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

where a constant end date is used to simplify processing.

1.1.3 Query 9 - Join

The business question is to total up the profit, by year and country of origin, for all parts matching a particular string. The prototype uses only a sub-query from the larger query to perform a two-way join using the text:

```
select sum(l_quantity), count(*)
from part, lineitem
where p_partkey = l_partkey
and p_name like '%green%'
group by n_name, t_year
order by n_name, t_year desc
```

which extracts a total quantity and count for parts containing the word "green".

A full 5-way join is performed using Variant B of the full query text, as:

```
select n_name, t_year,
sum(l_extprice*(1-l_disc)-ps_supplycost*l_quantity) as sum_profit
from part, supplier, lineitem, partsupp, order, nation, time
where s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and t_alpha = o_orderdate
and s_nationkey = n_nationkey
and p_name like '%green%'
group by n_name, t_year
order by n_name, t_year desc
```

which again finds all the "green" items, but further summarizes the profit by year and country of origin.