On the Use of NAND Flash Memory in High-Performance Relational Databases

by

Daniel Myers

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author . . . . . .
Department of Electrical Engineering and Computer Science
December 10, 2007

Certified by . . . . . .
Samuel R. Madden
ITT Career Development Professor
Thesis Supervisor

Accepted by . . . . . .
Terry P. Orlando
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

On the Use of NAND Flash Memory in High-Performance Relational Databases

by

Daniel Myers

Submitted to the Department of Electrical Engineering and Computer Science
on December 10, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

High-density NAND flash storage has become relatively inexpensive due to the popularity of various consumer electronics. Recently, several manufacturers have released IDE-compatible NAND flash-based drives in sizes up to 64 GB at reasonable (sub-$1000) prices. Because flash is significantly more durable than mechanical hard drives and requires considerably less energy, there is some speculation that large data centers will adopt these devices. As database workloads make up a substantial fraction of the processing done by data centers, it is interesting to ask how switching to flash-based storage will affect the performance of database systems.

We evaluate this question using IDE-based flash drives from two major manufacturers. We measure their read and write performance and find that flash has excellent random read performance, acceptable sequential read performance, and quite poor write performance compared to conventional IDE disks. We then consider how standard database algorithms are affected by these performance characteristics and find that the fast random read capability dramatically improves the performance of secondary indexes and index-based join algorithms. We next investigate using log-structured filesystems to mitigate the poor write performance of flash and find an 8.2x improvement in random write performance, but at the cost of a 3.7x decrease in random read performance. Finally, we study techniques for exploiting the inherent parallelism of multiple-chip flash devices, and we find that adaptive coding strategies can yield a 2x performance improvement over static ones.

We conclude that in many cases flash disk performance is still worse than on traditional drives and that current flash technology may not yet be mature enough for widespread database adoption if performance is a dominant factor. Finally, we briefly speculate how this landscape may change based on expected performance of next-generation flash memories.

Thesis Supervisor: Samuel R. Madden
Title: ITT Career Development Professor

Acknowledgments

I'd like to thank a number of people for their assistance and support during this thesis. At MIT, I'm indebted to Samuel Madden, for initially proposing the project and guiding me during its execution, and to all of my friends and labmates, particularly Daniel Abadi, Jennifer Carlisle, James Cowling, Adam Marcus, and Ben Vandiver. I'm grateful not only for their helpful discussions on flash and databases, but also for their friendship over the past few years, and, in James's case, being a partner in crime on any number of occasions, academic and otherwise, including risking life and limb in service of a side project involving a tandem bicycle. I'd also like to thank Sivan Toledo, who arrived late in the project but whose insight on low-level flash memory performance characteristics I very much appreciate. Finally, I'd like to thank Barbara Liskov, who allowed me to begin working on this topic when I was her student.

Beyond the Institute, I'm grateful for the support of all of my friends, but particularly Sarah Biber, Nicole Nichols, Shiri Azenkot, and Alison Asarnow. Finally, no acknowledgment section would be complete without mentioning my family, but particularly my parents, Alan and Nina Myers, for whose constant support in this and all my endeavors I am singularly fortunate.

This thesis is dedicated to my maternal grandmother, Molly Sumers, whose strength of character I should count myself fortunate ever to possess.

Chapter 1

Introduction

For decades, hard disks have been the storage medium of choice for applications such as databases that require substantial amounts of high-speed, durable storage. After dozens of years of intense research and development, hard disks are a mature technology offering low costs (~$0.33 per gigabyte), high storage densities (750 GB per 3.5" device), and relatively fast read and write performance (~75 MB/s at the high end).

Nonetheless, hard disks remain a less-than-ideal storage technology for high-performance database systems. Largely, this is due to the fact that, unlike virtually every other component in a modern computer system, hard disks are mechanical: data are read and written by moving an arm over platters spinning at thousands of RPM. Given that modern CPUs run at billions of cycles per second, the milliseconds required to rotate disk platters and reposition the disk arm in order to execute an I/O operation impose a substantial cost on accessing data. Moreover, even assuming that a disk can seek to a new location in 4 ms, it can only execute 250 I/O operations per second, which (assuming that each operation transfers 512 bytes) results in a throughput of only 128 KB/s, orders of magnitude too low to support a high-performance transaction processing system. As a result, in order to obtain maximum performance, the cost of a seek must be amortized over a large I/O operation: i.e., to the greatest degree possible, reads and writes must be of sequential blocks of data, not random ones. Indeed, much of database research (and to a large degree systems

research in general) has focused on ways to avoid expensive random I/O.

While various techniques can help mask the poor random I/O performance of hard disks, they cannot eliminate it, and modern database systems still suffer from poor random I/O performance in some situations. Additionally, the mechanical nature of hard disks imposes two further problems, namely excessive power consumption (and electrical costs can comprise up to 70% of a modern data center's operating budget [1]) and vulnerability to environmental shocks.

Over the past several years, the price of solid-state NAND flash memory, which has long been popular in embedded systems, has decreased dramatically, largely as a result of massive demand for persistent storage in cameras, cellular phones, music players, and other consumer electronic devices. Unlike hard disks, NAND flash devices exhibit virtually no seek penalty during random I/O, have relatively low power consumption, and are highly resistant to environmental shocks. While NAND flash remains more expensive per GB than disk ($16 vs $0.33), it is already inexpensive enough to be competitive for performance-critical applications, and prices are expected to further decline (Figure 1-1). On the other hand, however, NAND flash memory has a number of restrictions not seen in conventional disks. In particular, flash memory chips have a two-level hierarchical structure consisting of pages grouped into erase blocks. A page may only be written a small number of times (1-4) between erase cycles, and erasing a page requires erasing all the pages in its erase block. Moreover, erase blocks may only be erased a limited number (10^4 to 10^6) times before failing permanently.

In this thesis, we explore the use of NAND flash technology as a replacement for conventional magnetic hard disks in database applications. In particular, we focus on the use of commercially-available, off-the-shelf "flash disks." These devices package multiple flash memory chips into a single unit that hides the write restrictions of NAND flash memory and provides the same semantics and external interface as a standard hard disk. Our thesis is that the best query processing strategies and algorithms that have been developed for a disk-based environment will no longer be optimal when magnetic disks are replaced with flash memory, and that new techniques

[1] See `http://www.hpl.hp.com/news/2006/oct-dec/power.html`

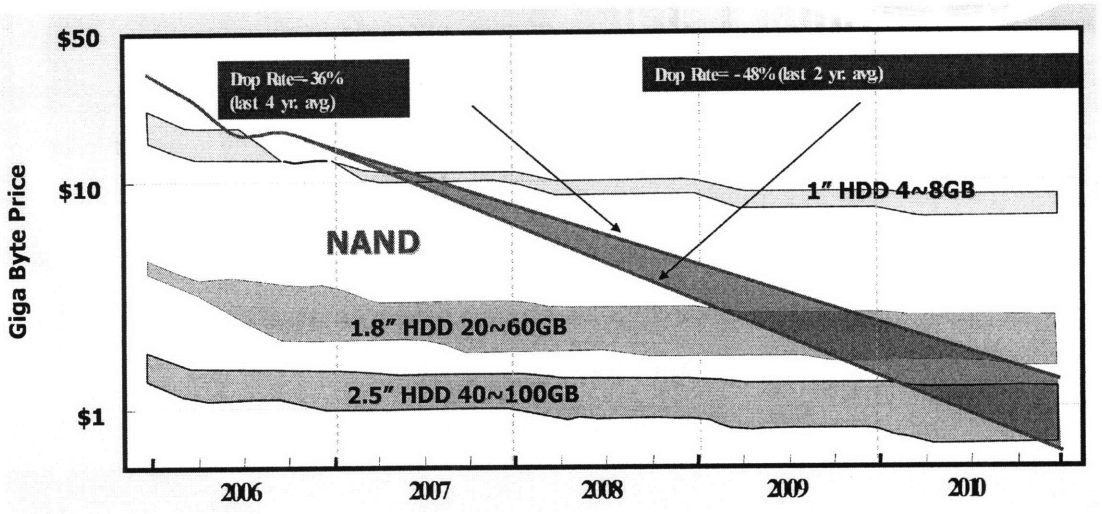


Figure 1-1: Past and projected costs of magnetic and flash memory storage (courtesy Samsung Corporation).

will need to be developed in order to fully realize the potential of these devices.

The rest of this thesis is organized as follows. In the next chapter, we describe NAND flash memory and the techniques that are used to package it into flash disks. In Chapter 3, we benchmark two commercially-available flash disks from major manufacturers and compare their performance to standard magnetic disks. In Chapter 4, we examine how flash disks affect the performance and utility of standard B-tree index structures, and in Chapter 5, we measure the affect of flash disks on join algorithm performance. In Chapter 6, we demonstrate that some of the write-related drawbacks of flash disks can be overcome using a log-structured storage manager, although doing so incurs a significant (3x) impact on read performance. In Chapter 7, we provide an overall analysis of our results on flash disks, and in Chapter 8, we investigate techniques that harness the hardware parallelism inherent in flash disks to improve performance. Finally, we discuss related work in Chapter 9 and conclude in Chapter 10.

Chapter 2

Flash Memory and Flash Disks

In this chapter, we describe the main properties of NAND flash memory and explain how it is packaged to produce "flash disks," devices with standard IDE interfaces and disk-like semantics.

As described in [6], flash memory is a type of nonvolatile, electrically-erasable programmable read-only memory (EEPROM). It is available in two varieties, NOR flash and NAND flash, named for the type of hardware structures that hold the electrical charges representing information. From an application perspective, the major differences between the two types of flash are related to the level at which they can be addressed and the time required to set bits. NOR flash is directly bit-addressable, which allows applications to be run from it directly, as a main memory. NAND flash, by contrast, is block-addressed and must be accessed through a controller, which precludes its use in this manner. In both types of flash, write operations may only clear bits, not set them. In order to set a bit, an erase operation must be performed. In a NOR flash device, an erase operation sets one or more bits, but it is extremely slow: on the order of 15 seconds. By contrast, in a NAND flash device, an erase operation sets all the bits in a superstructure called an "erase block," (described further below, but typically on the order of 256 KB), but it executes considerably more quickly: on the order of 1.5 ms/erase.

Additionally, NAND flash memory is available in considerably higher storage densities and at considerably lower costs than NOR flash. These properties, combined

with its higher-speed erase operations, make it preferable to NOR flash for use in mass storage devices, where the bit-addressability of NOR flash is of no particular advantage. In the rest of this thesis, we will consider only storage devices based on NAND flash memories. In rest of this chapter, we describe in more detail the structure of a NAND flash memory chip and the techniques used to build mass storage devices based on this technology.

2.1 Structure of a NAND Flash Chip

NAND flash memory chips have a two-level hierarchical structure. At the lowest level, bits are organized into pages, typically of ~2 KB each. Pages are the unit of read and write locality in NAND flash: in order to initiate an I/O operation, a command specifying the page ID must first be sent to the flash memory controller, which imposes a fixed setup time irrespective of the number of bits to be read or written. Thus, subsequent bits in the currently-selected page can be read or written far more cheaply than bits from a different page: i.e., flash chips exhibit page-level locality. Unlike as in a disk, however, the penalty for initiating an I/O operation on a page is constant, rather than a function of the previous I/O operation (e.g., there is no advantage to reading pages sequentially).

Pages are grouped into higher-level structures called erase blocks, consisting of ~64 pages each. While pages are the units of reads and writes, erase blocks are the units of erasure. As described above, writes to a page can only clear bits (make them to zero), not set them. In order to set any bit on a given page to 1, an expensive erase operation (~6-7x cost of a write, ~60-70x cost of a read) must be executed on the erase block containing the page; this sets all bits on all pages in the erase block to 1. Additionally, pages may only be written a limited number of times (typically one to four) between erases, regardless of the size or content of the writes. Finally, the number of erase cycles per erase block is limited, and typically ranges from 10,000 to 1,000,000. After the cycle limit has been exceeded, the erase block burns out, and it is impossible to perform further writes to the pages in it.

2.2 Building disk-like devices using NAND flash

Due to the write and erase restrictions of NAND flash memory chips, special techniques are required in order to use them efficiently in mass storage devices. Specifically, consider a naïve approach that simply packages multiple flash memory chips (multiple chips are needed to obtain tens-of-gigabytes capacities) into an enclosure and provides a controller that translates IDE commands into flash memory reads and writes using a static mapping between IDE block numbers and flash memory pages. In particular, consider the case of a write. Given a static mapping from block numbers to page IDs, each write of a block will require the erasure and rewrite of the corresponding page. Moreover, given that erase operations must be executed on entire erase blocks, the contents of any other live pages in the erase block must be read into memory before the erase operation is executed and written back after it completes. Given that erase blocks typically contain 64 pages, such a strategy would impose a massive overhead. Moreover, in the case of a power failure or software crash during the read/erase/rewrite operation, the data that had been read out of the erase block prior to erasure could potentially be lost. Finally, given that each erase block may only be erased a limited number of times before failing, such a strategy would lead to the premature failure of any erase blocks containing "hot" pages.

The drawbacks of this straw-man strategy were quickly realized, and considerable research [15, 22, 4, 7, 8] has focused on more efficient mechanisms to service writes. In general, they all rely on one of two options. First, they might replace the straw-man static map with a dynamic one. The virtual blocks presented to the application can be mapped to multiple (often any) flash memory page, and the device maintains a persistent record of these mappings, using (e.g.) the FTL [7]. Under these schemes, when a virtual block is to be written, a clean page is located on disk, the data are written there directly, and the persistent map is updated to mark the old page as obsolete and to record the new location of the virtual block. In this way, the expense of the read/erase/write strategy may be avoided. This is the approach taken by the flash disks with which we work in this thesis.

Of course, the dynamic strategy will eventually need to execute an erase operation. Every time the location of a virtual block is changed, the page on which it was previously stored becomes obsolete, or a "garbage page." Eventually, the device will consist entirely of pages which are either live (contain the latest version of virtual blocks) or garbage, with no clean pages left to absorb new writes. In this case, a garbage collection operation must be performed in order to make pages available. A garbage collection operation works similarly to the read/erase/write operation of the straw-man strategy: a garbage page is identified, live pages in its erase block are copied elsewhere on the device, and the erase block is erased, converting any garbage pages to clean pages. (As described, this strategy requires that garbage collection occur when at least one erase block's worth of free pages remains on the device.)

Clearly, the efficiency of garbage collection is maximized when the erase block that is erased contains as many garbage pages as possible, as this maximizes the number of clean pages generated while minimizing the copy-out cost of live pages. On the other hand, however, as indicated by [6], there is a competing tension due to the need for wear leveling. Recall that each erase block may only be erased a limited number of times before failing. In order to prolong the lifetime of the device, it is important to spread erasures evenly over the erase blocks in the device. If (for example) certain blocks contain only static data, then they will never be erased under a policy that tries to maximize the number of reclaimed sectors, concentrating the erase load on the remaining erase blocks in the device. As such, various schemes (again detailed in [6]) have been developed to detect and relocate hot and static blocks to achieve more even wear leveling, at the cost of efficiency. It has been shown in [1], however, that a policy that randomly chooses a block to erase is asymptotically optimal under certain (reasonable) assumptions.

Assuming read, write, and erase operations cost r, w, and e μsecs each, that erase blocks contain p pages each, and that the fraction of live pages per erase block is given by c, then the amortized cost of a write operation under both strategies is given in Equations 2.1 (naïve strategy) and 2.2 (dynamic strategy). Figure 2-1 plots these functions for $0 \leq c \leq 1$; note that for live page fractions below 0.9, the dynamic

strategy provides a one-to-two order of magnitude improvement in write performance.

The other option taken to improve write performance on flash memory is to use a log-structured filesystem, as originally proposed for disks by Rosenblum [18]. Here, new data are appended sequentially to a log of changes on disk, Periodically, the log is compacted to delete superseded log records and reclaim space for new updates. YAFFS [15], YAFFS2 [14], and JFFS [22] are examples of this approach. We note briefly that the rational for a log-structured filesystem on a flash device is somewhat different than on a conventional disk. On a conventional disk, an LFS is thought to be beneficial because most reads would be satisfied by a RAM cache (thus avoiding the expense of accessing the log), and appending data sequentially to a log avoids the cost of seeking the disk to update it in place. Flash memory chips, however, do not have seek delays. In this context, the advantage of an LFS is that it is a no-overwrite storage manager: like the dynamic mapping strategy described above, it is a method to avoid the problems associated with the simplistic read/erase/rewrite approach.

$$naive(c) = e + (p * c * (r + w)) + w \tag{2.1}$$

$$dynamic(c) = w + \frac{1}{(p - p * c)} * (e + (p * c) * (r + w)) \tag{2.2}$$

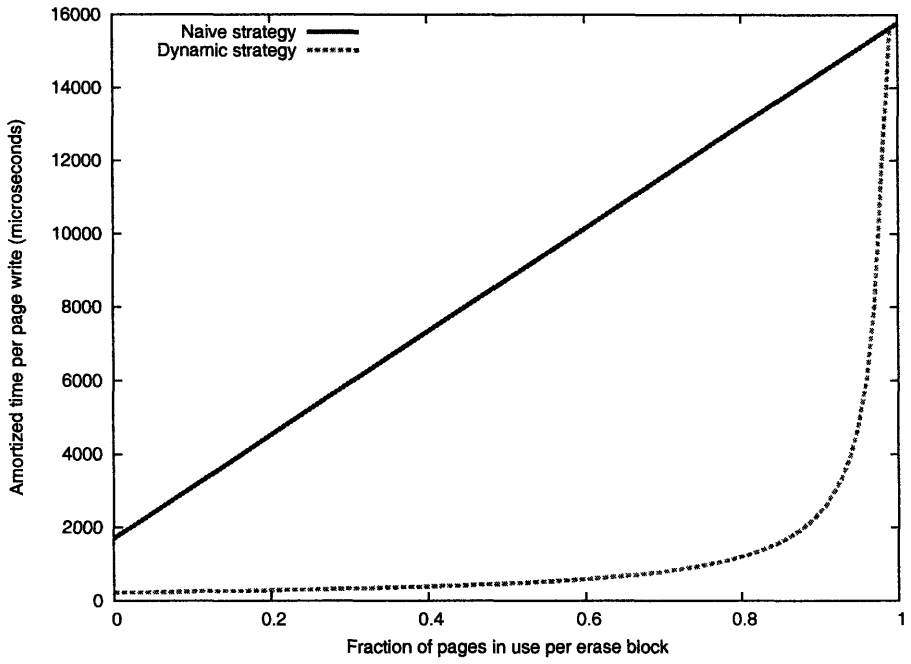


Figure 2-1: Cost of a page write under the naïve and dynamic write strategies as a function of the fraction of pages in use per erase block.

Chapter 3

Device Experiments

In order to assess the potential utility of NAND flash disks for high performance transaction processing, it is necessary to understand the performance delivered by such devices in real-world conditions. In this chapter, we perform benchmarks on two NAND flash disks from two major manufacturers. Specifically, we measure the read and write throughput obtainable under a range of both random and sequential I/O workloads. To provide a point of comparison, we also perform the same experiments using a standard IDE disk.

The flash devices used were 32 GB devices, which we will refer to as device A and device B. The IDE disk was a Western Digital WD2000JB-00REA0, with 8 MB of cache and UDMA-100 support. All devices used the standard ext2 filesystem running with Linux kernel 2.6.17 under Fedora Core 4. The system used had 4 GB of installed RAM. To avoid unnecessary writes, we mounted the filesystems using the "noatime" option, which disables file access time updates.

The test procedure was as follows. For each device, we first created a 20 GB file of zeros (using *dd*) and flushed the filesystem cache. We then executed 30 seconds worth (wall-clock time) of I/O operations, which were either random reads, sequential reads, random writes, or sequential writes. The size of the data read or written with each operation was varied between 1 KB and 1 MB, and when conducting random I/O, we pre-generated a permutation of file offsets to write such that no byte in the file would be written twice and all bytes were equally likely to be written. Writes

Operation	Device A	Device B
Sequential Read	0.44	1.11
Sequential Write	0.33	0.09
Random Read	30.71	68.89
Random Write	0.32	0.21

Table 3.1: Fraction of magnetic disk performance attained by flash disks.

were conducted synchronously.

The results of these experiments are shown in Figure 3-1, which gives the throughput for each workload as a function of I/O operation size, and Figure 3-2, which shows throughput for each device using 4 KB blocks. Based on these data, we propose two sets of observations. First, Table 3.1 compares the speed at which various basic I/O operations can be executed on flash disks as compared to magnetic disks, assuming 4 KB reads and writes (which provided the most operations per second). In particular, we note that magnetic disk obtains between 0.9x and 2.3x the sequential read performance of flash, between 3x and 10x the sequential write performance, and between 3x and 5x the random write performance. Flash, on the other hand, executes random reads between 30x and 69x faster than disk.

Table 3.1 gives a sense of how the performance of existing systems might change were the underlying magnetic storage system replaced with one based on flash. From an algorithm design perspective, though, it is also useful to look at the relative costs of random vs. sequential operations on magnetic disks and flash. Table 3.2 makes that comparison, again assuming 4 KB I/O operations. In particular, it shows that flash imposes only one one-hundredth the performance penalty of a disk for random reads, but that it still imposes a significant penalty for random writes.

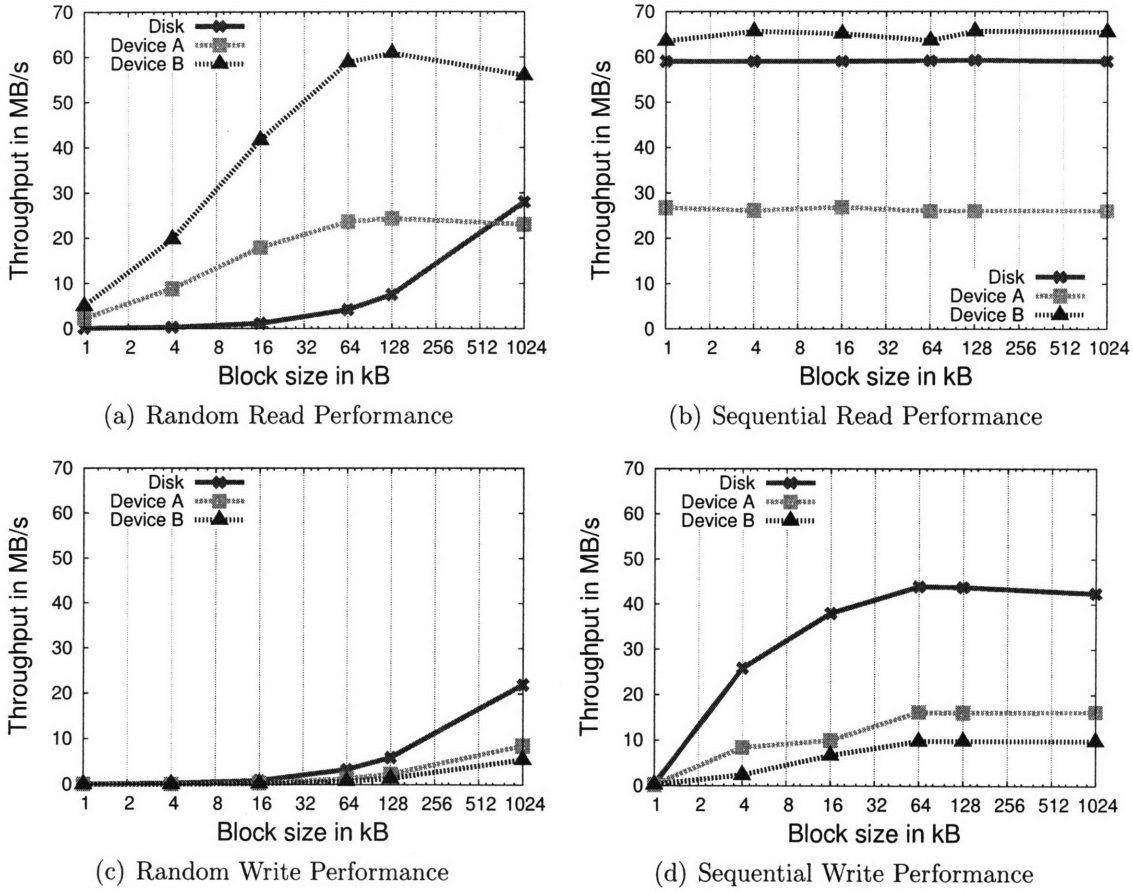
The poor performance of random writes as compared to sequential writes on flash disks is initially perplexing, given that our model of flash memory assumes constant page access costs irrespective of recent I/O history. Based on communications with the manufacturer of device A, one explanation for this behavior is the implementation of the logical–to–physical page mapping strategy, which requires additional copying for random writes but not large sequential ones, which is a behavior not included in

Operation	Flash Slowdown	Disk Slowdown
Read	(2.96, 3.32)	205.40
Write	(112.01, 48.12)	108.86

Table 3.2: Performance penalty of random I/O relative to sequential I/O for disk and flash. Flash values are given as (Device A, Device B).

our initial model of these devices. Additionally, random writes are in general problematic for garbage collection. Assuming devices attempt to locate sequentially-numbered virtual blocks in the same erase block (which will improve sequential overwrite performance), then random writes will tend to leave relatively larger amounts of live data in the same erase block as the pages they invalidate than do sequential writes, and this live data must be copied out during garbage collection before the block may be erased.

Finally, we observe two differences between the flash devices. Namely, device B provides significantly higher random and sequential read throughput than does the device A, whereas device A device enjoys an advantage in write performance. Based on consultations with the manufacturer, the low sequential read throughput of device A is apparently due to the device's controller, not to the flash memory itself.



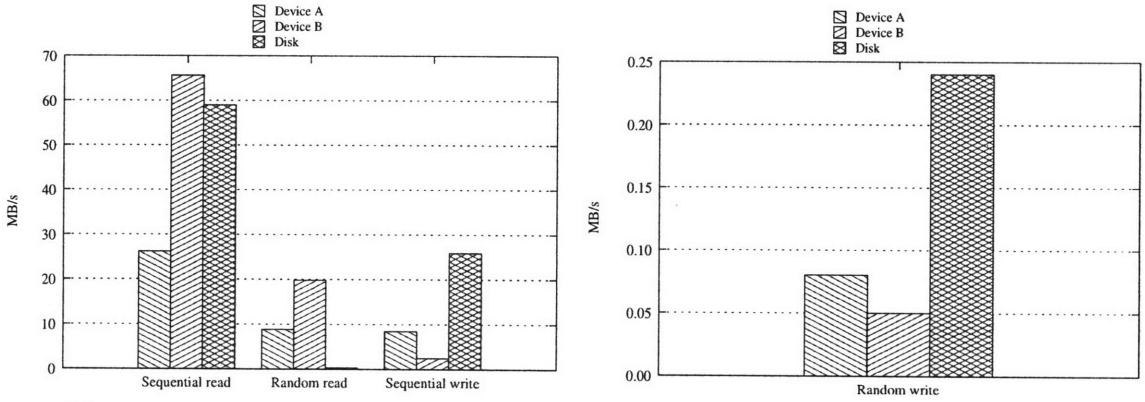
(a) Random Read Performance

(b) Sequential Read Performance

(c) Random Write Performance

(d) Sequential Write Performance

Figure 3-1: Comparison of flash and magnetic disk under various I/O workloads and a range of block sizes.



(a) All workloads except random write

(b) Random write (note altered y-axis range)

Figure 3-2: Comparison of flash and magnetic disk under various I/O workloads with 4 KB blocks.

Chapter 4

Indexes

Having measured the raw performance of flash devices, we now concentrate on how the use of flash disks affects the performance of a variety of database algorithms. In this section, we look in particular at index performance. We consider two aspects of indexes: the cost of creating and accessing them, and their utility in query execution. To examine the former, we measure the time taken to insert and to lookup keys in a B-Tree on both flash and magnetic disk. To examine the latter, we compare the time taken to execute a query using both a sequential scan of a file and an index traversal through an unclustered index on both types of devices.

While previous work has considered the utility of indexes on flash [13], the results presented here are unique in that they consider the performance of indexes on flash disks, not just single, raw flash chips, and in that they are concerned with absolute performance, not energy efficiency (as in [13]).

4.1 Creating and Maintaining Indexes

In order to see how the costs of creating and looking up keys in indexes changes on flash memory, we conducted a series of experiments using the Berkeley DB implementation of B-Trees. We created a range of B-Trees, each containing 2,000,000 records with 8-byte keys and 100-byte records. Following [23], we parametrized the B-Trees by r, which controlled the randomness of the key distribution. For $r = 0$, keys were

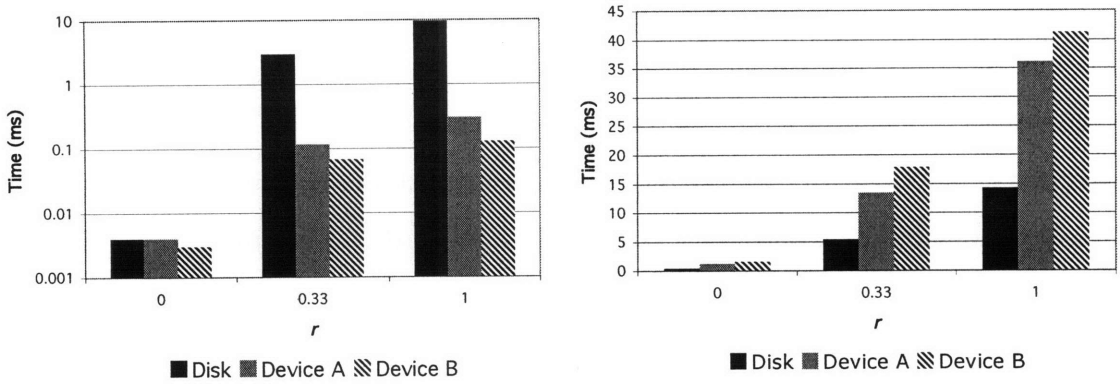
given successive integer values. At $r = 1$, they were chosen uniformly at random, and for intermediate values of r, they were chosen either from the sequential or uniform distribution with probability proportional to r. The B-Tree files occupied between 300 MB and 400 MB on disk, depending on r.

As r increases, the difficulty of the workload for the storage system increases. With $r = 0$, successive inserts tend to fall into the same B-Tree node. This results in multiple consecutive updates to the same page and allows the storage system to perform write coalescing before actually executing a hardware write operation. Alternatively, when $r = 1$, successive writes are randomly distributed throughout the tree, and write coalescing becomes considerably more difficult.

For each value of r, we executed 20,000 inserts followed by 20,000 lookups from the corresponding B-Tree, with keys for both operations chosen from the distribution given by r. The cache was cleared between executing the inserts and the deletes. We discarded the first 10,000 of each operation to allow the cache to warm up, then measured the time taken for the second 10,000. The B-Tree page size was varied between 512 bytes and 64 KB.

We used a Linux kernel parameter to restrict the memory recognized by the operating system to 96 MB, which meant that insufficient memory was available to fully cache either relation. This restriction models the actual situation that would be faced by a production database, in which limited memory is available to devote to a particular query.

The results for $r = 0$, $r = 0.33$, and $r = 1$ at 4096 byte block sizes on all three devices are shown in Figure 4-1. Disk outperforms flash on inserts by a factor of 2.5 to 3.8 across values of r, which is expected given that disk can execute random writes 5 times faster than flash. On queries, however, the situation is reversed: flash outperforms disk by a factor of between 24x and 75x at $r = 0.33$ and $r = 1$ (at $r = 0$ queries take essentially no time on either device, since they all tend to hit in the same B-Tree node as their predecessors). Given the 30-70x advantage in random reads enjoyed over disk by the flash devices, this result is also as expected.



(a) Query performance (note log scale)

(b) Insert performance

Figure 4-1: Time to execute query and insert operations on B-Trees stored on each device over a range of values of r.

4.2 Use of indexes in query execution

Ultimately, index performance only matters if the database system chooses to use indexes during query performance. Thus, a natural question to ask is when an index is likely to be useful during query processing on flash, and whether indexes are more or less useful than in a disk-based database. On disk, there is a well-known rule of thumb stating that a query that will retrieve more than about 1/1000th of the tuples from a file should not use an (unclustered) index, but should just scan the file directly, as the random read costs incurred by following index pointers into the main file become prohibitive. Here, we re-examine that rule of the thumb on a flash device with far faster random reads.

To investigate this question, we created a Berkeley DB queue containing 2,000,000 tuples, each with 4-byte keys and random 100-byte records. The first 4 bytes of each record were interpreted as an integer between 0 and 200,000, and a secondary index was built on this value using a Berkeley DB B-Tree. The main file totaled 412 MB on disk, and the secondary index occupied 42 MB. Again, we restricted available RAM to 96 MB to prevent the operating system from simply caching all the data, although sufficient memory was available to fully cache the secondary index.

Using these files, we ran queries of the form: `SELECT * FROM tuples WHERE record_int < x`, with x varied to control query selectivity. The queries were ex-

ecuted using both direct scans and index traversals, and the execution times for device A, device B, and the conventional disk are shown in Figure 4-2. Note that index scans are uncompetitive on disk for query selectivities over 0.05%, whereas they remain competitive with direct scans until selectivities of roughly 2.25% (device A) to 3.4% (device B) on flash devices, which is a two-order-of-magnitude improvement.

It is critical that a query planner take these new data into account. For example, suppose that a query planner uses the disk-based 0.05% figure to select a direct table scan for a query with 0.05% selectivity running on a flash disk. On the data used here, the flash disk would need 8 to 16 seconds, depending on the device, to execute the query. Had the planner used the correct 2-3% figure and selected an index scan instead, the query would taken between 0.24 and 0.46 seconds (again depending on the device), which is a ~32x speedup. Additionally, it is worth noting that a conventional disk would have required 7.8s to complete the query, which is 16x to 32x longer than the flash devices.

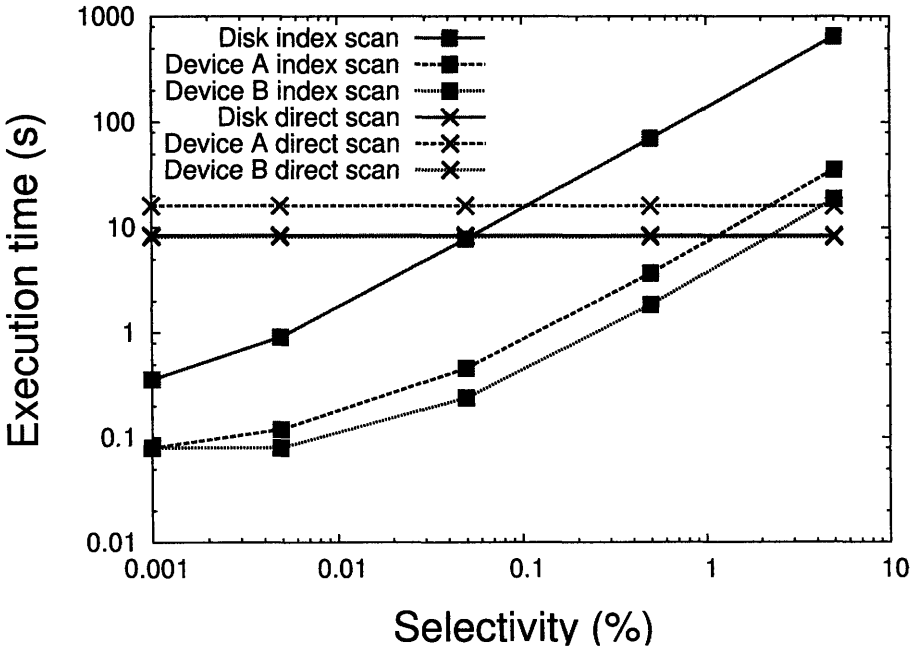


Figure 4-2: Performance of direct table scans vs index scans.

Chapter 5

Joins

Another area of query execution in which secondary storage performance plays a key role is in external memory joins, in which one or both of the relations to be joined is too large to fit in main memory. We investigate the performance of four common external join algorithms:

- Standard nested loops join (NL)
- Index nested loops join (INL)
- Index-based sort-merge join (ISM)
- Hash join

A nested loops join simply scans the entire inner relation for matching tuples once for every tuple in the outer relation. An index nested loops join works similarly, but it requires an index (clustered or unclustered) on the inner relation, and it probes the inner relation for matching tuples through the index rather than scanning it completely. Finally, an index-based sort-merge join uses (clustered or unclustered) indexes on both the inner and outer relations to traverse both in sorted order and perform a sort-merge join.

Conversely, a hash join requires no indexes. Here, we describe the Grace Hash Join algorithm [9]. For each relation in the join, it sequentially scans the relation,

hashes each tuple on the join attribute using some hash function *h1*, and writes each tuple to a partition on disk based on the hash value. These partitions are sized so that a full partition from the outer relation can fit in main memory. Then, once both relations have been partitioned, the hash join algorithm reads in pairs of corresponding partitions and outputs matching tuples, using a second hash function *h2* to probe for matches.

To evaluate the tradeoff between these algorithms on magnetic and flash disks, we performed a primary/foreign key join between the `customers` and `orders` tables using TPC-H data generated at scale 2. At this scale, `customers` contains 300,000 records and `orders` contains 3,000,000. The query was `SELECT * FROM customers AS c, orders AS o WHERE c.custkey` $< x$, where x was varied to control query selectivity.

We generated two copies of the dataset. One was stored using Berkeley DB B-Tree files, and one was stored using Berkeley DB queue files. The B-Tree files provide clustered indices (and thus fast lookups) on the primary keys of the relations and support range queries, but this comes at the cost of more expensive traversals, as B-Tree nodes are generally not contiguous on disk. Conversely, we used queue files to represent database heap files: they are generally contiguous on disk, but they do not support range queries or fast lookups. (Technically, Berkeley DB queue files could have been used to support fast lookups in this case, but we treated them as simple heap files.) In both cases, we used 4 KB pages, and we inserted records into the files in random order so as to fully stress the join algorithms (e.g., given sorted data, a nested loops algorithm can employ early termination optimizations to avoid scanning the whole relation).

Additionally, we created a secondary B-Tree index on `orders.custkey` in the B-Tree dataset and secondary B-Tree indexes on both `orders.custkey` and `customers.custkey` in the Queue dataset. The sizes of all files used in these experiments are shown in Table 5.1.

The NL and INL algorithms used `customers` as the inner relation, as doing so yielded better performance. (When executing nested loop joins with customers as the

Dataset	Orders	Orders idx	Cust.	Cust. idx
Queue	405M	62M	66M	5.9M
B-Tree	453M	62M	74M	N/A

Table 5.1: Sizes of input files used in join algorithm tests.

inner relation, we stopped searching for matching records for a given order as soon as one was found, as only one customer for a given order can exist.)

We again restricted available RAM to 96 MB, and Berkeley DB was configured with 2 MB of cache per open database handle.

The results of this experiment are presented in Figure 5-1, which shows execution time as a function of query selectivity for each join strategy on both datasets and all three storage devices. We do not show the NL results, as the runtimes quickly became unreasonable.

We begin with device A and the Queue dataset (Figure 5-1(a)). At selectivities from zero to roughly 0.45%, a flash-based ISM strategy is the fastest possible join. For disk, the ISM strategy is also the best alternative. Flash, however, can follow pointers from the unclustered indexes into the heap files more rapidly than disk owing to its fast random I/O; indeed, as the selectivity of the query increases, the advantage of flash-based ISM over disk-based ISM increases with it, owing to the increased number pointer lookups executed.

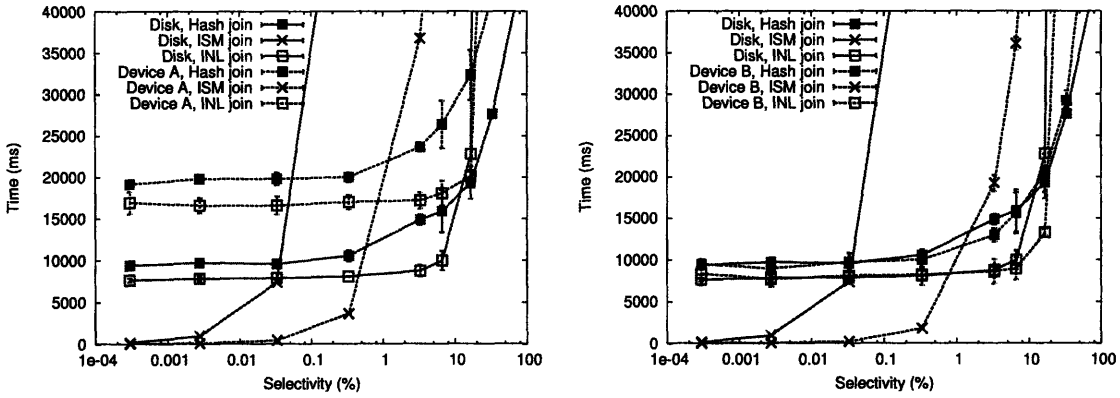
At selectivities from 0.45% to 16%, a disk-based INL strategy is the fastest join, and INL is also the fastest option on flash for most of this range (ISM is faster until roughly 1% selectivity). In this regime, an INL join largely devolves to a sequential scan of the outer relation, and the conventional disk benefits from its superior sequential read performance. Finally, for selectivities above 16%, a hash join is the best strategy for both disk and flash, and the disk outperforms device A, as it has a performance advantage in the sequential reads and writes that are the core of this algorithm.

The results for device B on the Queue dataset (Figure 5-1(b)) are somewhat different, owing to its faster sequential reads. A flash-based ISM strategy is the best join for selectivities up to 0.8%. From 0.8% to 7%, device B and the conventional

disk are both best served by INL and perform roughly equivalently, which is explained by the INL join devolving to a sequential scan of the outer relation and the devices having similar sequential read performance. At 16% selectivity, both devices are still best served by INL, but device B has a performance advantage, as its fast random I/O allows it to quickly index into the inner relation. At higher selectivities, a hash join is the best choice for both devices, and here the disk has a slight performance advantage over device B because it can write out the hash partitions more quickly.

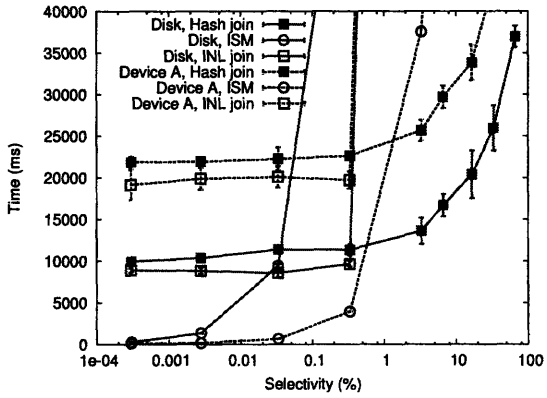
Finally, we consider the two flash devices on the B-Tree dataset (Figures 5-1(c) and 5-1(d)). Here, an ISM join on flash is again the fastest strategy for selectivities up to 0.65% (device A) and 1.25% (device B). At higher selectivities, a hash join becomes the superior strategy for both the flash devices and the disk, again due to the expense of ISM random I/O. Device A always executes the hash join more slowly than the conventional disk, due to its poor sequential read and write performance. Device B, by contrast, executes the hash join as quickly as disk until close to 25% selectivity, owing to its superior sequential read performance (relative to device A). Beyond 25% selectivity, the costs of writing out the larger hash partitions dominates, and the conventional disk becomes the faster device. Note that unlike as in the Queue dataset, an INL join never becomes the best strategy (and is not shown), as sequentially scanning a B-Tree involves substantial random I/O to follow pointers between leaf nodes.

Overall, the conclusion drawn from these results is that the flash devices have better performance than the conventional disk at low selectivities in the presence of indexes, where their fast random I/O can be exploited to maximum advantage. At higher selectivities, device A is hamstrung by its poor sequential read performance relative to disk, although device B remains competitive. At highest selectivities, device B is inferior to disk owing to its disadvantage in writing out the hash partitions.

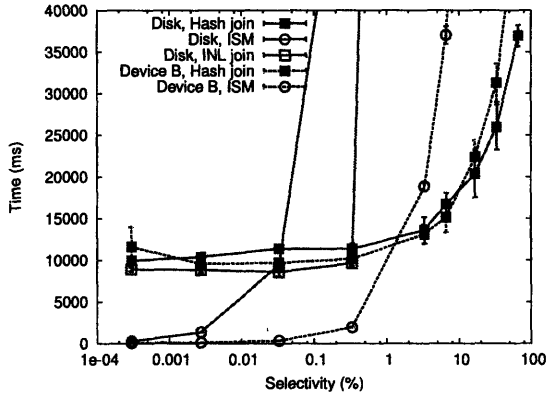


(a) Device A on the Queue dataset

(b) Device B on the Queue dataset



(c) Device A on the B-Tree dataset



(d) Device B on the B-Tree dataset

Figure 5-1: Execution times for external memory join algorithms on the Queue and B-Tree datasets using both flash devices. Points are averages over 3 trials with error bars indicating 1 standard deviation.

Chapter 6

Log-Structured Storage Manager

In order to overcome the poor random write performance of flash disks, we consider using a log-based storage system, which can convert random writes into sequential writes. This is in the spirit of other work on flash storage [23, 11], which also uses a logging approach. Rather than implementing our own log-based system, we investigate the idea by rerunning the device benchmarks described in Chapter 3 using NILFS [10], a log-structured filesystem (LFS) for Linux and a 10 GB file in lieu of a 20 GB one, due to implementation constraints of NILFS. NILFS performs all I/O sequentially by appending block modifications and i-node updates to the end of a log. A table of block numbers of i-nodes is kept in memory and used to lookup the appropriate location of file contents when reads are performed. The test setup was identical to that presented in the previous experiments, except we created the 10 GB file by writing each block in a random order, to better approximate the state of the log after a long series of random updates to the file.

The results for 16 KB I/O operations using device A are presented in Figure 6 (16 KB blocks gave the most operations per second for the LFS). As expected, random write performance has been dramatically improved, by a factor of 8.2. This improvement comes with a significant cost, however. Sequential and random read performance drop by factors of 2.9 and 3.7, respectively. Sequential write performance also decreases slightly, attributable perhaps to the development status of the NILFS filesystem or to additional overhead to update i-nodes.

These results are somewhat encouraging. For a transaction-processing workload consisting mostly of small random reads and writes, a flash disk using LFS provides 4.5x the random read throughput and 8.2x the random write throughput of a disk, whereas a flash disk using a conventional file system provides 16.8x the random read throughput but only 0.33x the random write throughput. Thus, for these devices and write-heavy transaction-processing workloads, a logging file system will provide better performance than a disk; by contrast, for read-heavy transaction-processing workloads, a conventional file system would be preferred. We say the results are only somewhat encouraging due to costs: neglecting power consumption, given that magnetic disks are so much cheaper than flash, an array of multiple disks could still provide superior random I/O at lower cost.

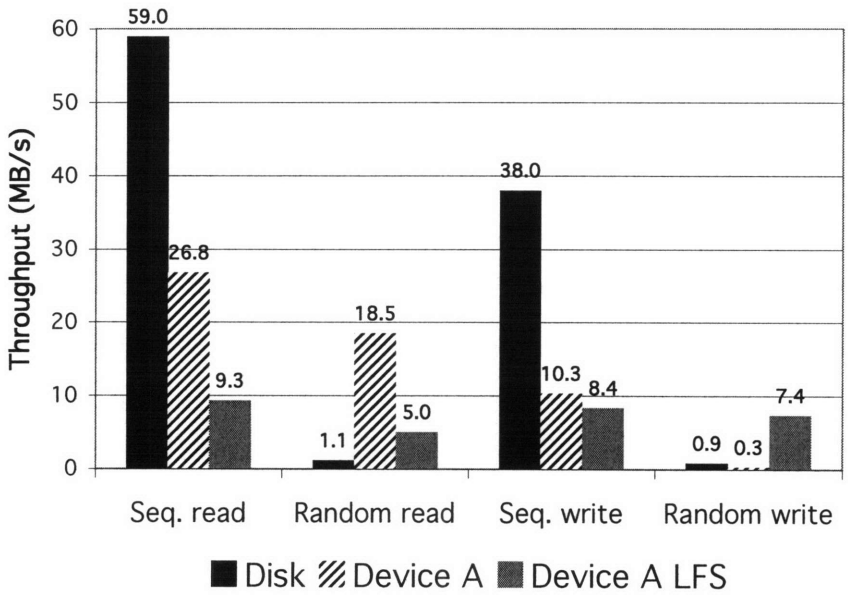


Figure 6-1: Device performance with a logging filesystem and 16 KB I/O operations.

Chapter 7

Discussion

The results in the previous chapters lead to the following two general conclusions. First, flash has a clear advantage over disk when using B-tree indices, due to its superior random read performance. This allows it to answer low-selectivity queries up to an order of magnitude faster than disk if the appropriate index is present. It also substantially improves the utility of index-based join algorithms. When indices are unavailable or selectivity is greater than a few percent, then current flash hardware loses its clear superiority over conventional disks. One of the devices tested (device B) can match or exceed a conventional disk in the sequential scans of relations that are required to answer queries in the absence of indexes, whereas the other (device A) cannot. Additionally, if a query plan requires temporary results to be written to permanent storage (e.g., a hash or sort-merge join), then both flash devices are inferior to conventional disks.

Second, the extremely poor random write performance of current flash hardware implies that an in-place storage manager will likely be unable to provide acceptable performance for update-intensive applications. A log-structured storage manager can be used to substantially improve random write performance, but at the cost of significantly limiting the fast random I/O that was an initial motivation for the use of flash devices; moreover, the cost of the flash device would still be higher than that of an array of conventional disks delivering superior performance.

Based on these observations, we see the following possible applications for current-

generation NAND flash disks in high-performance databases:

- In read-mostly, transaction-like workloads. For example, web applications such as e-mail clients, forums, and account-access front-ends (e.g., applications to view bank account balances or track shipments) all generate workloads with few updates and many small queries.

- In environmentally harsh applications (e.g., field-deployed laptops), or settings where maintenance costs are particularly high (e.g., data centers.) Here, the resilience of flash memory provides a clear advantage over disks: broken disks have even worse write performance than flash!

- In embedded applications, due to their low power consumption.

Ultimately, however, current flash technology seems to be a poor fit for high-update databases if raw performance or price/performance ratio is the primary consideration. Flash disks using an LFS can provide 3x the random write performance of a disk, but flash disks costs 60x as much per GB ($16 vs $0.33), so multiple disk spindles could provide superior performance at lower cost (albeit at higher power consumption). Moreover, as will be discussed further in Section 9.1, were absolute performance critical, a hybrid disk-RAM device that would provide far superior write performance could be built for only 4.5x the cost of a NAND flash disk of equivalent size.

That said, however, both disk and flash technology are constantly evolving, and what is true today may well no longer be true in the near future. For example, Samsung projects that within a year, their flash disks will see two-fold increases in sequential read and write performance, a two-fold improvement in random read performance, and a ten-fold improvement in random write performance [24]. Moreover, costs per gigabyte will continue to decrease (e.g., in 2005, NAND flash cost $45/GB[1], representing a three-fold cost decrease in under two years).

[1]See `http://news.zdnet.co.uk/hardware/0,1000000091,39237444,00.htm`

Such performance improvements and cost reductions, if realized, would give flash devices an across-the-board superiority over conventional disks and would make them attractive for both read-mostly and write-intensive workloads (of course, the fast random read performance would still make the observations in this thesis regarding when to use indexes highly relevant). Given that the write performance of flash is expected to increase, we conjecture that existing update-in-place storage managers are likely to remain the appropriate means by which to store databases on these devices, as log structured schemes impose a significant read performance overhead and add a fair amount of complexity (particularly with regard to free space reclamation).

Finally, even if these performance improvements are realized, as others have noted [11], the logical block abstraction provided by IDE-based flash disks is limiting. Further performance improvements (such as those proposed by [11]) could be realized were databases able to "look under the hood" and directly access flash devices. Databases have considerably more information about their expected I/O access patterns than do most applications and could almost certainly lay their data out more efficiently than the current generic logical block abstraction provided by flash disk controllers can. Additionally, because flash drives are composed of multiple memory chips, it is quite likely databases could extract additional parallelism from flash drives by striping data across several chips and accessing them in parallel, and we address this subject in the next chapter. Both types of optimizations could be done within the existing IDE interface by encoding flash operations such as writes to a specific page or erases of a particular block as reads and writes of special blocks exposed by the device, avoiding the need for expensive new interfaces.

Chapter 8

Exploiting Parallelism

8.1 Overview

The previous chapters have treated flash disks as single, monolithic units. In reality, however, these devices are composed of multiple, independently-addressable flash chips, and a strategy that exploits the parallel nature of the devices may yield performance benefits. In this chapter, we investigate ways to exploit the inherent parallelism of flash disks.

Specifically, there are two ways in which one might take advantage of this parallelism. First, one could stripe data across the devices. I.e., one could split data items into fragments and write one fragment to each flash chip. The latency of reads and writes could then potentially be decreased due to parallel execution. Second, one could replicate data across the devices. This could potentially boost read performance, as reads dependent on data located on a currently-busy flash chip could be satisfied by a read from another chip with a copy of the data. Write performance, on the other hand, might suffer due to the increased write load of updating the replicas.

Our hypothesis is twofold. First, while the above techniques are also applicable to disk arrays, we hypothesize that the tradeoffs between them will be different on flash. Second, we hypothesize that a system which chooses the replication/striping strategy to use as a function of the workload can outperform a system that makes a single static choice. In this chapter, we test these hypotheses using erasure codes

based on Rabin's Information Dispersal Algorithm (IDA) [17]. Specifically, while we continue to expose the ordinary virtual block abstraction that standard flash disks do, rather than map a single virtual block to a single physical page, we will instead fragment or replicate the block and store it on multiple physical pages on different chips.

The rest of this chapter is organized as follows. In the next two sections, we discuss ways by which data are commonly distributed over disk arrays and explain Rabin's IDA. We then describe the discrete event simulator used to test our hypotheses and present experimental results.

8.2 Striping and Replicating Data Across Disks: RAID

The standard way to distribute data across multiple disks is to use one of the levels of RAID originally described in [16]. Of those levels, three are of interest here. First, RAID level 1 mirrors data across multiple disks: k disks are used ($k \geq 2$), and k copies of each block are maintained. RAID level 1 can potentially improve concurrent read performance by executing independent reads simultaneously on different disks but neither improves write performance nor provides additional storage capacity.

Second, RAID level 0 breaks data into chunks and stripes those chunks across two or more disks. RAID level 0 can improve both read and write performance and imposes no storage overhead, but a failure of any disk in the array renders the entire array useless. Finally, RAID level 5 extends RAID level 0 by striping parity information across $k \geq 3$ disks such that the array may be recovered in the event of a single disk failure (two simultaneous failures destroy the array). A RAID-5 array of k disks has the capacity of $k - 1$ disks.

In the present work, we do not simulate RAID levels directly, but rather choose values of n and m for Rabin's IDA (described below) to correspond to RAID levels 0, 1, and 5.

8.3 Striping and Replicating Data Across Flash Chips: Rabin's IDA

In order to distribute virtual blocks across multiple physical pages located on multiple flash chips, we use Rabin's Information Dispersal Algorithm [17]. At a high level, the IDA works by splitting blocks into n fragments of which m are required to reconstruct the block (i.e., $k = n - m$ fragments may be lost). More specifically, if we consider the case of sequences of bytes (rather than of arbitrary symbols) the algorithm is based on matrix multiplication and inversion in the Galois field GF(2^8). First, consider encoding. The algorithm chooses n coding vectors of length m such that any subset of m vectors is linearly independent, which yields an $n * m$ coding matrix A. Second, the input byte sequence F is divided into chunks of length m, which form the columns of a $m * \frac{|F|}{m}$ data matrix B. To generate the coded fragments, the algorithm multiplies $A * B$, yielding an $n * \frac{|F|}{m}$ matrix C in which each of the n rows corresponds to one coded data fragment.

Next, consider reconstruction. Suppose that we have m coded fragments from C and we wish to regenerate the original data matrix. Since we generated C by multiplying B by A, and any m rows of A are linearly independent, it can be shown [17] that we can regenerate B by multiplying the coded fragments available by A^{-1}, the inverse of A. Because the encoding and decoding operations rely only on inner products, the algorithms are lightweight and are able to make use of the vector processing extensions on modern processors [17]; this characteristic is important given that we wish to impose minimal CPU overhead.

The storage overhead for this encoding scheme is given by n/m; by judicious choice of n and m, we can achieve arbitrarily low storage overhead. In the present work, we fix n as the number of flash chips in the flash device (so as to be able to write one fragment to each chip) and vary both m and the number of the generated fragments actually written to provide varying degrees of redundancy. When $m = n$, we have RAID-0-like striping with no redundancy; when $m = 1$ and we write one fragment, we have a single copy; and when $m = 1$ and we write r fragments, we have

r-way replication. On the other hand, the IDA algorithm allows us to express data distribution strategies outside of this standard set, such as $n = 4$, $m = 3$, write 4, which gives 1.33x replication.

8.4 Experimental Setup

To test these hypotheses, we built a discrete event simulator to test various coding strategies under a variety of database workloads. The simulator models a 64 MB flash disk composed of four 16 MB flash chips with 2048 byte pages and 64-page erase blocks. While this simulated device is considerably smaller than the actual disk used in previous chapters, a multi-gigabyte device would have required excessive resources to simulate without providing any additional insight. The flash chips had 20 microsecond page read, 200 microsecond page program, and 1500 microsecond block erase times, which were derived from Samsung product literature [5]. Simulated garbage collection was carried out using the model presented in Section 2.2, with the flash device assumed to be 50% full, which gave an average-case write time of 466 microseconds. For comparison, we also simulated an array of four conventional disks, each with 8,192 2048 byte pages (total array capacity 64 MB), a 6 ms uniform seek time, and 60 MB/s read and write bandwidth.

We note that while the performance of the simulated disk agrees closely with real-world devices, the performance of the simulated flash device differs significantly from the flash disks tested earlier in this thesis. For example, using the 466 microsecond average case write time, a simulated flash chip could sustain 4.2 MB/s of random writes, while the devices tested manage less than 100 KB/s. Even allowing for OS overheads that would preclude executing a write in 466 microseconds, the difference is considerable. There are three potential sources of the discrepancy. First, the simulated device is assumed to be 50% full, so garbage collection operations need to copy out only half an erase block's worth of live pages. The actual devices, however, may not be able to detect that only half of their pages are actually in use (due to the filesystem layered over it), which would cause them to copy out an entire erase block's

worth of pages on each garbage collection event. With 100% space utilization, our simulator gives read and write times that are close to those observed experimentally by other groups (e.g., ~15 ms/write) [12]

Second, the devices tested may be buffering writes for tens of milliseconds before executing them, presumably in an attempt to perform write coalescing. As our benchmarks were single-threaded, such buffering would reduce the throughput by delaying the next write. Further experiments (not shown) demonstrated that device B could support two concurrent random write benchmarks each with the same throughput as a single benchmark instance, indicating that higher throughput is possible.

Finally, the devices may be managing flash memory in blocks of 256 or 512 KB, to decrease the mapping overhead, which would require them to copy 256 to 512 KB per write if the device is completely full [21].

The simulated devices were tested under a workload designed to resemble that of a transaction processing system; namely, a mix of random reads and writes of single pages. The system was configured with a number of read worker processes and a number of write worker processes, each of which generated a read or write to a page in the device chosen uniformly at random, waited for the operation to complete, slept for an interval, and then issued another request. Separate base sleep times existed for each class of workers (read or write) and the actual time slept was chosen uniformly at random from the interval [0.75*SLEEPTIME, 1.5*SLEEPTIME]. For each set of parameters (number of writers, number of readers, writer base sleep time, reader base sleep time), we simulated running the system on both the flash disk and the disk array using all possible coding strategies with $n = 4$ (i.e., require from $m = 1$ to 4 fragments to reconstruct and write between the number of fragments required and 4 fragments to the device). A write request returned to the application once m fragments had been written, and fragments of a single virtual block were always written to different chips or disks. Only the I/O time was considered (the CPU time for encoding and decoding fragments was not taken into account). The simulation was considered to have converged after a run of eight, ten-second epochs showed no more than a 1.0% change in the total operations/s rate from the end of the first epoch to the end of the

eighth.

Finally, we note that by using the IDA to produce fragments of 2 KB virtual blocks, we produce fragments smaller than the 2 KB physical pages. Here, we assume that either a small in-memory buffer exists to coalesce these small writes into full (or close to full) page writes, or that the underlying flash devices supports partial page programs. Thus, when computing the time taken for a write, we use a linear model that charges a worker process with only the time to write fragments, not entire pages. For reads, we charge each worker process with the time taken to read an entire physical page (including a full seek, on disk) as most likely the requesting worker was the only worker with an interest in the contents of the page, precluding amortization.

An alternative design, similar but not explored here, would be to provide larger (e.g., 8 KB) virtual blocks so that the IDA-produced fragments would be closer in size to a full physical page.

8.5 Results

Using the procedure described above, we simulated a transaction processing system with 1, 2, 5, or 10 write processes and 5, 25, or 50 read processes. The write process base delay was 200 microseconds, and the read process base delay was varied over 200, 1500, and 3000 microseconds.

Here, we show results from three cases: 5 readers, 200 microsecond base write delay (high write load); 25 readers, 1500 microsecond base write delay (mixed load); and 50 readers, 3000 microsecond base write delay (high read load). In each case, there were five writer workers. Figure 8-1 shows the degree of replication in use by the optimal coding strategy for each configuration of workload and device, and Figure 8-2 shows the number of fragments required to reconstruct a block under the optimal strategy in each configuration.

Figures 8-1 and 8-2 show that while the optimal strategy for a disk array is workload-invariant, the optimal strategy for a flash device depends on the workload. The optimal strategy on a disk array always uses a one-copy approach: each virtual

block is written to exactly one page. On a flash device, however, write-dominated workloads are best served by a RAID-1-like strategy that fragments blocks evenly among all chips without any replication (thus requiring all fragments to reconstruct), while read-dominated workloads benefit from a replication strategy that places full copies of blocks on multiple chips. These differences are naturally explained by the differences between the two types of storage devices. Disk arrays incur substantial seek delays to begin accessing a block regardless of the amount of data to be retrieved from it, and in the presence of a random write workload, updating multiple locations becomes prohibitively expensive. Moreover, due to the high fixed seek cost, parallel speedups cannot be realized, as reading or writing a fraction of a block takes almost as long as writing the entire block. Flash devices, on the other hand, have essentially no seek penalty, which allows redundancy to be maintained for read-heavy workloads even in the presence of some updates. By contrast, data are written to flash devices more slowly than they are read. Thus, given the low cost of initiating an I/O operation, under write-heavy workloads a striping strategy that reduces the amount of data to be written to any chip can improve performance.

Figure 8-3 shows the performance of the optimal coding strategy (in operations/s) relative to three standard strategies on a flash device. "Stripe" is essentially RAID-0, "one-copy" is the strategy described in previous chapters of this thesis, and "replication" is 2-way replication. Note that the optimal coding strategy is essentially no better than any of the standard strategies; indeed, it differs only in the mixed-workload case, and there only by a negligible amount. There is no single strategy that is competitive with the optimal strategy across all workloads, however. Replication is equivalent under the read-heavy workload but delivers roughly half the performance of the optimal strategy under the write-heavy workload, one-copy is never equivalent, and striping is equivalent under the write-heavy and mixed workloads but is 20% slower under the read-heavy workload.

Figure 8-4 presents the same comparison on a disk array, and here, by contrast, single-copy is always equivalent to the optimal strategy (since indeed it is the optimal strategy as described above). Moreover, Figure 8-4 clearly illustrates the vast

performance advantage that the simulated flash device has over the simulated disk array in this setting: regardless of workload or replication strategy, the flash device outperforms the disk array by two to three orders of magnitude. Indeed, these results suggest that significant improvements can be realized in the actual devices currently available and tested in earlier chapters.

8.6 Conclusion

We thus conclude that while there is little observed advantage in using Rabin's IDA over standard RAID levels (and the CPU requirements of the IDA, while slight, are not modeled here and might eliminate the small advantage observed), significant (up to 2x) performance improvements can be achieved on flash devices by varying the layout of data across flash chips.

We close with a few caveats. First, while these experiments chose a single layout for the entire database, in practice, one would likely choose a layout for individual items (e.g., tables) in the database instead. I.e., one might use a high degree of replication to encode a table which is mostly read and a low degree of replication to encode one that is mostly written. Second, the simulations presented here assumed the existence of an in-memory map to map virtual blocks to fragments on permanent storage. In practice, this may not be practical for large devices. The existing FTL used by flash disks could be extended to support our coding strategies in a straightforward manner, however: rather than mapping a virtual block to a single page, it would simply need to support mapping a virtual block to multiple pages.

Finally, we note that the results presented here assume a workload composed entirely of the small random reads and writes typical of a transaction processing system. Other systems might exhibit a larger degree of sequential I/O and might yield different conclusions.

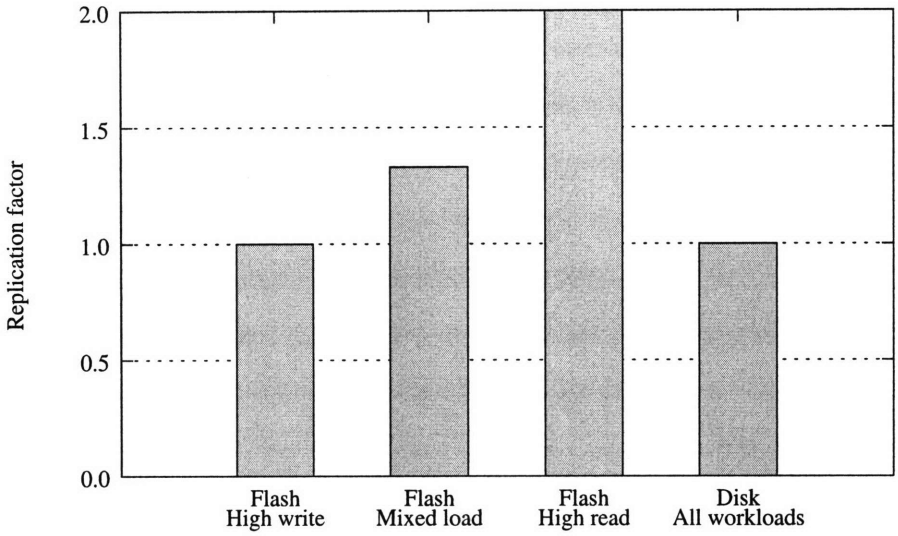


Figure 8-1: Degree of replication provided by the optimal coding strategy for both a flash device and a disk array.

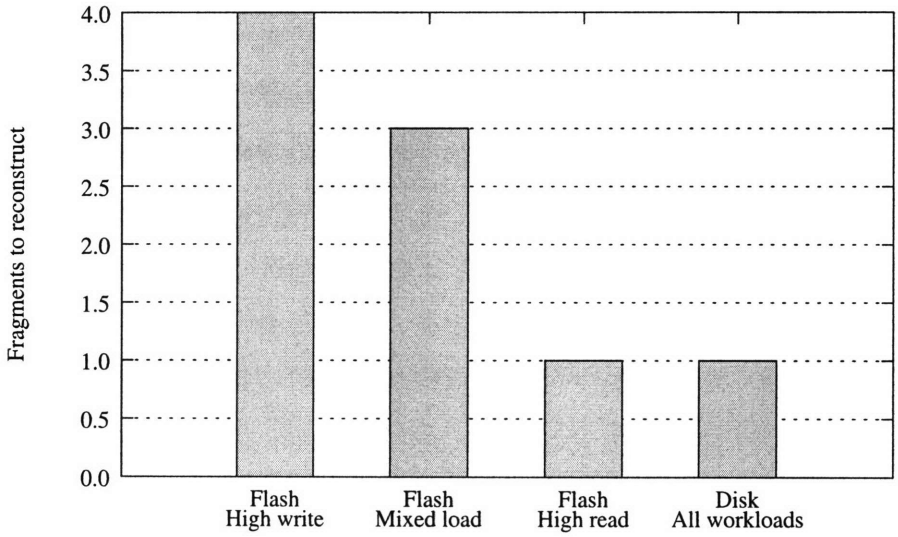


Figure 8-2: Number of fragments required by the optimal coding strategy to reconstruct a block for both a flash device and a disk array.

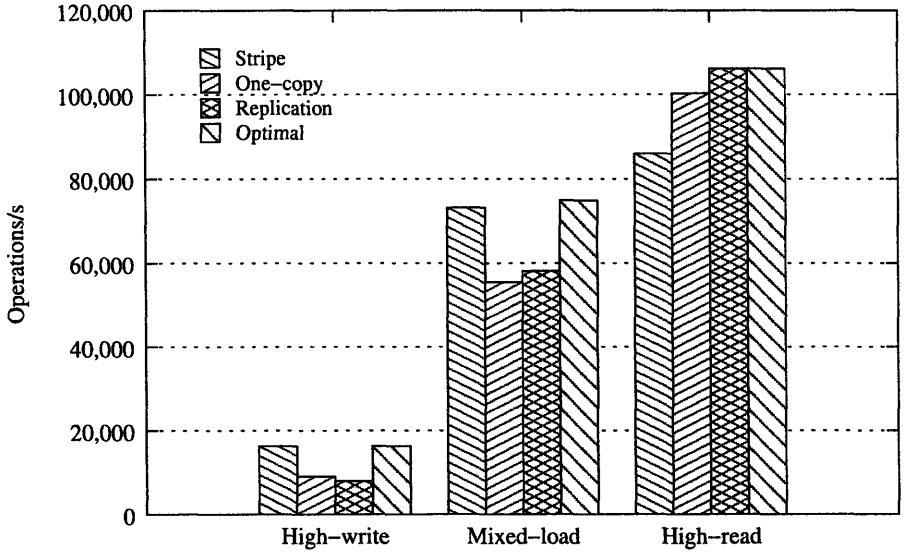


Figure 8-3: Performance of the optimal coding strategy relative to three standard strategies on a flash device.

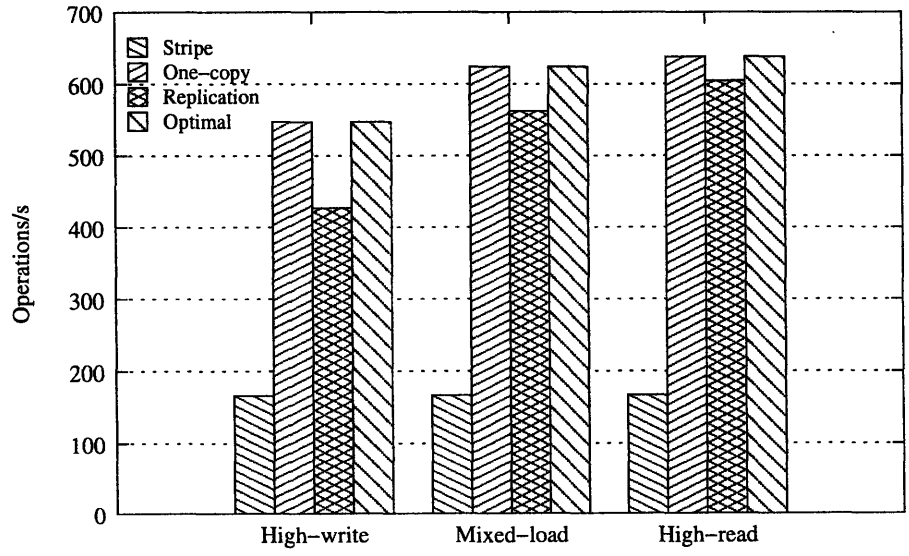


Figure 8-4: Performance of the optimal coding strategy relative to three standard strategies on a disk array.

Chapter 9

Related Work

In this chapter, we discuss two major classes of previous research related to the present work: attempts to mask the mechanical seek latency of conventional magnetic disks, and prior attempts to build database systems using flash memory.

9.1 Hiding Disk Latency

The problem of masking the delay of a disk seek has been extremely well studied in the research literature, in both the general systems and database communities, and we cannot possibly review the entire body of work here. Generally, though, the need to avoid disk seeks can be seen throughout the design of modern database systems, from B-tree data structures that use extremely high-arity trees to minimize the number of I/O operations required to descend to a leaf, to hash join algorithms that rewrite entire relations on the fly to make them more amenable to sequential I/O, to query plans that scan entire relations rather than incur the random I/O penalty of an unclustered index traversal.

Clever algorithms can go only so far, however, and a second approach to improving database I/O performance is the RAM disk, of which the Texas Memory Systems RamSan product line [20] is an example. Such systems provide a substantial quantity (10-1000 GB) of DRAM memory with a battery backup and an equivalent amount of magnetic storage. In the event of a power failure, sufficient battery power will

be available to copy the contents of the RAM to disk, avoiding data loss. These devices have the advantage of extremely high performance–RAM is far faster than either flash or magnetic memory–but at a steep price: a 16 GB RamSan, for example, costs $25,000, compared to $270 for a 16 GB NAND flash disk. This figure is somewhat misleading, however, as a RamSan device includes high-performance network interfaces (e.g., Infiniband) that far outclass anything contained in a flash disk. A fairer comparison would be the cost of eight 2 GB DDR2 RAM chips (total $1,072), with perhaps another $100-$200 for a controller, battery, and backup disk, for a total cost in the range of $1200, or roughly 4.5x the cost of an equivalent flash disk. (These are resale prices, but the comparison holds assuming similar markups on both NAND flash disks and RAM chips.) These figures suggest that a RAM/disk/battery hybrid might well be a more appropriate device than a NAND flash disk if pure performance were the only consideration, although a flash device would exhibit lower power consumption and higher physical robustness.

9.2 Previous Work on Flash Memory

There has been considerably previous work on algorithms and data structures for flash memories, which is well summarized in a survey paper of the same title [6]. Generally, this work forms the basis for the techniques considered in this paper; e.g., the dynamic block remapping strategy to handle writes or the use of logging filesystems to avoid expensive random writes.

9.2.1 Previous Work on Flash Memory Databases

Work to date on relational databases on flash has been conducted along three major axes. First, several authors have considered the problem of scaling relational databases down to function in flash-based, resource-constrained environments such as smart cards [3, 19]; this work is outside the scope of the present study, which focuses on high-performance systems in resource-rich environments.

Second, several authors have considered the problem of index design on flash

memory. In [23], Wu et al. present a log-structured B-tree design. In contrast to standard implementations, in which B-tree nodes are overwritten with new data on each modification, the B-tree implementation proposed by Wu et al. writes a log record for each change to a node. All nodes in the B-tree share a common log output buffer in memory, which is flushed to disk as it fills. This approach converts slow random I/O into fast sequential I/O, dramatically improving update performance. The Wu scheme has two drawbacks, however. First, it appears to scale poorly as B-tree size increases. The Wu approach maintains an in-memory map with an entry for every node in the B-tree that gives the locations of the node's log records on flash; as the size of the B-tree increases, so to does this map. Second, the Wu approach makes read operations potentially more expensive, as log records may need to be read from multiple locations on disk.

To resolve this latter problem, Nath and Kansal [13] propose a hybrid scheme in which updates to B-tree nodes are either performed in-place or logged, depending on the access pattern seen by the given node (e.g., frequently written but infrequently read nodes should have their updates logged, while frequently read but infrequently written nodes should be updated in place). They prove that the algorithm used for switching update modes is 3-competitive, which is a lower bound. They still require an in-memory node translation table to locate the pages containing log records for each node, however. Additionally, their work appears to be more concerned with embedded systems than with high performance, and their experimental evaluations use a flash chip of only 128 MB.

Finally, beyond indexes, Lee and Moon have proposed a general logging framework for a database storage manager, which they call In-Page Logging [11]. The key idea in this scheme is to co-locate log records in the same erase block as the database pages whose updates they record. Such a design has the advantage of making garbage collection efficient: once the log pages have been filled and the data pages must be updated and rewritten, the old data pages and log pages can be erased in a single operation. Additionally, the authors claim that the reduced write/erase load attained by this system will improve overall performance.

There are, however, a few concerns with this scheme. First, it seems likely that In-Page Logging would have a substantial impact on read performance (though the paper does not explicitly evaluate read overheads), as the authors propose locating 16 log pages in each erase block, and thus each read of a data page would require reading 16 log pages, any of which might contain relevant log records. Second, the In-Page Logging scheme requires the ability to co-locate specific pages within the same erase block, which is not supported by current generation flash disks, nor (to the best of our knowledge) a feature planned by any flash disk manufacturer.

9.2.2 Previous work on IDA and Disk Arrays

This thesis is not the first work to propose using Rabin's IDA to distribute data over multiple disks in an array. In [2], Bestavros concluded that the IDA was the best way to take advantage of redundancy in disk arrays to improve performance, which differs from the results presented here. The strategies against which they compared the IDA, however, are not the same as those in the present work. Second, the paper dates from the late 1980s, and disk performance has since evolved significantly. Finally, the work was purely analytical and did not include an analysis of contention between concurrent processes in the system, which this work does.

Chapter 10

Conclusion

This thesis has made the following contributions. First, we have measured the performance of a new class of "flash disk" storage devices under a set of realistic I/O workloads and characterized those regions of the space in which these devices provide a performance advantage over conventional disks. Second, we have evaluated a number of core database algorithms (indexes and joins) on these devices and demonstrated how the new performance characteristics of flash memory change the way a query optimizer ought generate query plans. We showed that current flash disks provide excellent random read performance, which enables them to considerably outperform disks at index-based "needle-in-a-haystack" queries, but that their poor write performance renders them unattractive for update-intensive workloads.

Additionally, we explored the use of a log-structured storage manager to replace the standard in-place update approach and found that while a log-structured approach can significantly improve random write performance, it does so at the expense of the fast random reads that make flash devices unique. We conclude that log-structured approaches can make flash devices tolerable in update-intensive workloads if other considerations force their use, but that given the lackluster performance and high cost of flash memory, disk is otherwise preferable.

We also explored ways in which the inherent parallelism of flash devices may be exploited. We evaluated a variety of strategies for fragmenting and replicating virtual blocks across multiple flash chips, and we demonstrated that flash chips and

disk arrays require different strategies. Moreover, we demonstrated that, on flash devices, an adaptive strategy that chooses the degree of fragmentation/replication based on access patterns can yield up to a 2x speedup over a static strategy.

Finally, we speculated on future trends in NAND flash performance and cost and suggest that within the next year or two, improved flash disks may present a compelling performance argument for use in both read-mostly and update-intense workloads, although at present, they are best suited to read-mostly applications. We noted that further performance improvements might be realized were databases able to access flash disks via a lower-level interface than the current IDE-disk emulation provided by flash drive manufacturers.

Bibliography

[1] Avraham Ben-Aroya. Competitive analysis of flash-memory algorithms. Master's thesis, Tel-Aviv University, April 2006.

[2] A. Bestavros. IDA-based disk arrays. In *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, Miami Beach, FL, 1991.

[3] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling down database techniques for the smartcard, 2000.

[4] M.-L. Chiang and R.-C. Chang. Cleaning policies in mobile computers using flash memory. *The Journal of Systems and Software*, 48(3):213–231, 1999.

[5] Samsung Corporation. *K9XXG08UXM specifications*, November 2005.

[6] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[7] Intel Corporation. Understanding the flash translation layer (FTL) specification. Technical report, Intel Corporation, 1998.

[8] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[9] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[10] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006.

[11] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 55–66. ACM, 2007.

[12] Itay Malinger. The performance characteristics of logical-block-address flash-memory storage devices. Master's thesis, Tel-Aviv University, July 2007.

[13] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In Tarek F. Abdelzaher, Leonidas J. Guibas, and Matt Welsh, editors, *IPSN*, pages 410–419. ACM, 2007.

[14] Aleph One. Yaffs 2 specification and development notes. http://www.yaffs.net/yaffs-2-specification-and-development-notes.

[15] Aleph One. Yaffs: Yet another flash filing system. http://www.aleph1.co.uk/yaffs/index.html.

[16] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM.

[17] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.

[18] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[19] Michal Spivak and Sivan Toledo. Storing a persistent transactional object heap on flash memory. *SIGPLAN Not.*, 41(7):22–33, 2006.

[20] Texas Memory Systems. Ramsan-400 details. http://www.superssd.com/products/ramsan-400/indexb.htm.

[21] Sivan Toledo. Personal communication, December 2007.

[22] David Woodhouse. Jffs: The journaling flash file system. http://sourceware.org/jffs2/jffs2-html/.

[23] C. H. Wu, L. P. Change, and T. W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *Proc. 9th Intl. Conf. on Real-Time and Embedded Computing Systems and Applications*, 2003.

[24] Michael Yang. Personal communication, May 2007. Samsung Corporation.