

1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997

The Five- minute Rule

Goetz Graefe, Hewlett-Packard Laboratories

20

20 Years Later

and How Flash Memory Changes the Rules

In 1987, Jim Gray and Gianfranco Putzolu published their now-famous five-minute rule¹ for trading off memory and I/O capacity. Their calculation compares the cost of holding a record (or page) permanently in memory with the cost of performing disk I/O each time the record (or page) is accessed, using appropriate fractions of prices for RAM chips and disk drives. The name of their rule refers to the break-even interval between accesses. If a record (or page) is accessed more often, it should be kept in memory; otherwise, it should remain on disk and read when needed.

Based on then-current prices and performance characteristics of Tandem equipment, Gray and Putzolu found that the price of RAM to hold a 1-KB record was about equal to the (fractional) price of a disk drive required to access such a record every 400 seconds, which they rounded to five minutes. The break-even

1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007

*The old rule continues
to evolve, while flash memory
adds two new rules.*

The Five-minute Rule 20 Years Later

and How Flash Memory Changes the Rules

interval is about inversely proportional to the record size. Gray and Putzolu gave one hour for 100-byte records and two minutes for 4-KB pages.

The five-minute rule was reviewed and renewed 10 years later in 1997.² Lots of prices and performance parameters had changed (e.g., the price of RAM had tumbled from \$5,000 to \$15 per megabyte). Nonetheless, the break-even interval for 4-KB pages was still around five minutes. The first purpose of this article is to review the five-minute rule after another 10 years.

Of course, both previous papers acknowledged that prices and performance vary among technologies and devices at any point in time (e.g., RAM for mainframes versus minicomputers, SCSI versus IDE disks, etc.). Therefore, interested readers are invited to reevaluate the appropriate formulas for their environments and equipment. The values used here (in table 1) are meant to be typical for 2007 technologies rather than universally accurate.

In addition to quantitative changes in prices and performance, qualitative changes already under way will affect the software and hardware architectures of servers and, in particular, database systems. Database software will change radically with the advent of new technologies: virtualization with hardware and software support, as well as higher utilization goals for physical machines; many-core processors and transactional memory supported both in programming environments and hardware;³ deployment in containers housing thousands of processors and many terabytes of data;⁴ and flash memory

that fills the gap between traditional RAM and traditional rotating disks.

Flash memory falls between traditional RAM and persistent mass storage based on rotating disks in terms of acquisition cost, access latency, transfer bandwidth, spatial density, power consumption, and cooling costs.⁵ Table 1 and some derived metrics in table 2 illustrate this point.

Given that the number of CPU instructions possible during the time required for one disk I/O has steadily increased, an intermediate memory in the storage hierarchy is desirable. Flash memory seems to be a highly probable candidate, as has been observed many times by now.

Many architecture details remain to be worked out. For example, in the hardware architecture, will flash memory be accessible via a DIMM slot, a SATA (serial ATA) disk interface, or yet another hardware interface? Given the effort and delay in defining a new hardware interface, adaptations of existing interfaces are likely.

A major question is whether flash memory is considered a special part of main memory or a special part of persistent storage. Asked differently: if a system includes 1-GB traditional RAM, 8-GB flash memory, and a 250-GB traditional disk, does the software treat it as 250 GB of persistent storage and a 9-GB buffer pool, or as 258 GB of persistent storage and a 1-GB buffer pool? The second purpose of this article is to answer this question and, in fact, to argue for different answers in file systems and database systems.

Many design decisions depend on the answer to this question. For example, if flash memory is part of the buffer pool, pages must be considered “dirty” if their contents differ from the equivalent page in persistent storage. Synchronizing the file system or checkpointing a database must force disk writes in those cases. If flash memory is part of persistent storage, these write operations are not required.

Designers of operating systems and file systems will

want to use flash memory as an extended buffer pool (extended RAM), whereas database systems will benefit from flash memory as an extended disk (extended persistent storage). Multiple aspects of file systems and database systems consistently favor these two designs.

Moreover, the characteristics of flash memory sug-

TABLE 1

Prices and Performance of Flash and Disks

	RAM	Flash disk	SATA disk
Price and capacity	\$3 for 8x64 Mbit	\$999 for 32 GB	\$80 for 250 GB
Transfer bandwidth	----	66 MB/s API	300 MB/s API
Access latency	----	0.1 ms	12 ms average
Active power	----	1 W	10 W
Idle power	----	0.1 W	8 W
Sleep power	----	0.1 W	1 W

gest some substantial differences in the management of B-tree pages and their allocation. Beyond optimization of page sizes, B-trees can use different units of I/O for flash memory and disks. Presenting the case for this design is the third purpose of this article.

ASSUMPTIONS

Forward-looking research always relies on many assumptions. This section lists the assumptions that led to the conclusions put forth in this article. Some of the assumptions are fairly basic, whereas others are more speculative.

One assumption is that file systems and database systems assign flash memory to a level between RAM and the disk drives. Both software systems favor pages with some probability that they will be touched in the future but not with sufficient probability to warrant keeping them in RAM. The estimation and administration of such probabilities follow the usual lines (e.g., LRU, or least recently used).

We assume that the administration of such information employs data structures in RAM, even for pages whose contents have been removed from RAM to flash memory. For example, the LRU chain in a file system's buffer pool might cover both RAM and flash memory, or there might be two separate LRU chains. A page is loaded into RAM and inserted at the head of the first chain when it is needed by an application. When it reaches the tail of the first chain, the page is moved to flash memory and its descriptor to the head of the second LRU chain. When it reaches the tail of the second chain, the page is moved to disk and removed from the LRU chain. Other replacement algorithms would work *mutatis mutandis*.

Such fine-grained LRU replacement of individual pages is in contrast to assigning entire files, directories, tables, or databases to different storage units. It seems that page replacement is the appropriate granularity in buffer pools. Moreover, proven methods exist to load and replace buffer-pool contents entirely automatically, without assistance from tuning tools and without directives by users or administrators. An extended buffer pool in flash memory should

exploit the same methods as a traditional buffer pool. For truly comparable and competitive performance and administration costs, a similar approach seems advisable when flash memory is used as an extended disk.

FILE SYSTEMS

The research for this article assumed a fairly traditional file system. Many file systems differ in one way or another from this model, but most still generally adhere to it.

Each file is a large byte stream. Files are often read in their entirety, their contents manipulated in memory, and the entire file replaced if it is updated at all. Archiving, version retention, hierarchical storage management, data movement using removable media, etc. all seem to follow this model as well.

Based on this model, space allocation on disk attempts to use contiguous disk blocks for each file. Metadata is limited to directories, a few standard tags such as a creation time, and data structures for space management.

Consistency of these on-disk data structures is achieved by careful write ordering, fairly quick write-back of updated data blocks, and expensive file-system checks after any less-than-perfect shutdown or media removal. In other words, we assume the absence of transactional guarantees and transactional logging, at least for file contents. If log-based recovery is supported for file contents such as individual pages or records within pages, then a number of the arguments presented here need to be revisited.

DATABASE SYSTEMS

We assume fairly traditional database systems with B-tree indexes as the workhorse storage structure. Similar tree

Relative Costs for Flash Memory and Disks

	NAND Flash	SATA disk
Price and capacity	\$999 for 32 GB	\$80 for 250 GB
Price per GB	\$31.20	\$0.32
Time to read a 4-KB page	0.16 ms	12.01 ms
4-KB reads per second	6,200	83
Price per 4-KB read per second	\$0.16	\$0.96
Time to read a 256-KB page	3.98 ms	12.85 ms
256-KB reads per second	250	78
Price per 256-KB read per second	\$3.99	\$1.03

[Derived metrics from <http://www.dramexchange.com>, <http://www.dvnation.com>, <http://www.buy.com>, <http://www.seagate.com>, and <http://www.samsung.com>; all 4/11/2007]

TABLE 2

The Five-minute Rule 20 Years Later

and How Flash Memory Changes the Rules

structures capture not only traditional clustered and nonclustered indexes but also bitmap indexes, columnar storage, contents indexes, XML indexes, catalogs (meta-data), and allocation data structures.

With respect to transactional guarantees, we assume traditional write-ahead logging of both contents changes (such as inserting or deleting a record) and structural changes (such as splitting B-tree nodes). Efficient log-based recovery after failures is enabled by checkpoints that force dirty data from the buffer pool to persistent storage.

Other assumptions include variations such as “second-chance” or fuzzy checkpoints. In addition, nonlogged (allocation-only logged) execution is permitted for some operations such as index creation. These operations require appropriate write ordering and a “force” buffer-pool policy.⁶

FLASH MEMORY

We assume that hardware and device drivers hide many implementation details such as the specific hardware interface to flash memory. For example, flash memory might be mounted on the computer’s motherboard, a DIMM slot, a PCI board, or within a standard disk enclosure. In all cases, DMA (direct memory access) transfers (or something better) are assumed between RAM and flash memory. Moreover, the assumption is that there is either efficient DMA data transfer between flash and disk or a transfer buffer in RAM. The size of such a transfer buffer should be, in a first approximation, about equal to the product of transfer bandwidth and disk latency. If it is desirable that disk writes should never delay disk reads, the increased write-behind latency must be included in the calculation.

Another assumption is that transfer bandwidths of flash memory and disk are comparable. While flash write bandwidth has lagged behind read bandwidth, some products claim a difference of less than a factor of two (e.g., Samsung’s Flash-based solid-state disk in table 1). If necessary, the transfer bandwidth can be increased by using array arrangements, as is well known for disk drives.⁷ Even redundant arrangement of flash memory may prove advantageous in some cases.⁸

Since the reliability of current NAND flash suffers after 100,000 to 1 million erase-and-write cycles, we assume that some mechanisms for “wear leveling” are provided. These mechanisms ensure that all pages or blocks of pages are written with about the same frequency. It is important to recognize the similarity between wear-leveling algorithms and log-structured file systems,^{9,10} although the former also move stable, unchanged data such that their locations can absorb some of the erase-and-write cycles.

Also note that traditional disk drives do not support more write operations, albeit for different reasons. For example, six years of continuous and sustained writing at 100 MB per second overwrites an entire 250-GB disk fewer than 80,000 times. In other words, assuming a log-structured file system as appropriate for RAID-5 or RAID-6 arrays, the reliability of current NAND flash seems comparable. Similarly, overwriting a 32-GB flash disk 100,000 times at 30 MB per second takes about 3½ years.

In addition to wear leveling, we assume that there is an asynchronous agent that moves fairly stale data from flash memory to disk and immediately erases the freed-up space in flash memory to prepare it for write operations without further delay. This activity also has an immediate equivalence in log-structured file systems—namely, the cleanup activity that prepares space for future log writing. The difference is that disk contents must merely be moved, whereas flash contents must also be erased before the next write operation at that location.

In either file systems or database systems, we assume separate mechanisms for page tracking and page replacement. A traditional buffer pool, for example, provides both, but it uses two different data structures for these two purposes. The standard design relies on an LRU list for page replacement and on a hash table for tracking pages (i.e., which pages are present in the buffer pool and in which buffer frames). Alternative algorithms and data structures also separate page tracking and replacement management.

The data structures for the replacement algorithm are assumed to be small and high-traffic, and are therefore kept in RAM. We also assume that page tracking must be as persistent as the data; thus, a buffer pool’s hash table is reinitialized during a system reboot, but tracking information for pages on a persistent store such as a disk must be stored with the data.

As previously mentioned, we assume page replacement on demand. There may also be automatic policies and mechanisms for prefetch, read-ahead, and write-behind.

Based on these considerations, we assume that the contents of flash memory are pretty much the same,

whether the flash memory extends the buffer pool or the disk. The central question is, therefore, not what to keep in cache but how to manage flash memory contents and its lifetime.

In database systems, flash memory can also be used for recovery logs, because its short access times permit very fast transaction commit. Limitations in write bandwidth discourage such use, however. Perhaps systems with dual logs can combine low latency and high bandwidth, one log on a traditional disk and one log on an array of flash chips.

OTHER HARDWARE

In all cases, RAM is assumed to be a substantial size, although probably less than flash memory or disk. The relative sizes should be governed by the five-minute rule.¹¹ Note that despite similar transfer bandwidth, the short access latency of flash memory compared with disk results in surprising retention times for data in RAM.

Finally, we assume sufficient processing bandwidth as provided by modern many-core processors. Moreover, we believe that forthcoming transactional memory (in hardware and in the software runtime system) permits highly concurrent maintenance of complex data structures. For example, page replacement heuristics might use priority queues rather than bitmaps or linked lists. Similarly, advanced lock management might benefit from more complex data structures. Nonetheless, we do not assume or require data structures more complex than those already in common use for page replacement and location tracking.

THE FIVE-MINUTE RULE

If flash memory is introduced as an intermediate level in the memory hierarchy, relative sizing of memory levels requires renewed consideration. Tuning can be based on purchasing cost, total cost of ownership, power, mean time to failure, mean time to data loss, or a combination of metrics. Following Gray and Putzolu,¹² this article focuses on purchasing cost. Other metrics and appropriate formulas to determine relative sizes can be derived similarly (e.g., by replacing dollar costs with energy use for caching and moving data).

Gray and Putzolu introduced the following formula:^{13,14}

$$\text{BreakEvenIntervalInSeconds} = (\text{PagesPerMBofRAM} / \text{AccessesPerSecondPerDisk}) \times (\text{PricePerDiskDrive} / \text{PricePerMBofRAM})$$

It is derived using formulas for the costs of RAM to hold a page in the buffer pool and of a (fractional) disk to perform I/O every time a page is needed, equating these two costs, and solving the equation for the interval between accesses.

Assuming modern RAM, a disk drive using 4-KB pages, and the values from tables 1 and 2, this produces:

$$(256 / 83) \times (\$80 / \$0.047) = 5,248 \text{ seconds} = 90 \text{ minutes} = 1\frac{1}{2} \text{ hours}$$

(The “=” sign indicates rounding in this article.) This compares with two minutes (for 4-KB pages) 20 years ago.

If there is a surprise in this change, it is that the break-even interval has grown by less than two orders of magnitude. Recall that RAM was estimated in 1987 at about \$5,000 per megabyte, whereas the 2007 cost is about \$0.05 per megabyte, a difference of five orders of magnitude. On the other hand, disk prices have also tumbled (\$15,000 per disk in 1987), and disk latency and bandwidth have improved considerably (from 15 accesses per second to about 100 on SATA and about 200 on high-performance SCSI disks).

For RAM and flash disks of 32 GB, the break-even interval is

$$(256 / 6,200) \times (\$999 / \$0.047) = 876 \text{ seconds} = 15 \text{ minutes}$$

If the 2007 price for flash disks includes a “novelty premium” and comes down closer to the price of raw flash memory—say, to \$400 (a price also anticipated by Gray and Fitzgerald¹⁵), then the break-even interval is 351 seconds = 6 minutes.

An important consequence is that in systems tuned using economic considerations, turn-over in RAM is about 15 times faster (90 minutes / 6 minutes) if flash memory rather than a traditional disk is the next level in the storage hierarchy. Much less RAM is required, resulting in lower costs for purchase, power, and cooling.

Perhaps most interesting, applying the same formula to flash and disk results in the following:

$$(256 / 83) \times (\$80 / \$0.03) = 8,070 \text{ seconds} = 2\frac{1}{4} \text{ hours}$$

Thus, all active data will remain in RAM and flash memory.

Without a doubt, two hours is longer than any common checkpoint interval, which implies that dirty pages in flash are forced to disk not by page replacement but by checkpoints. Pages that are updated frequently must be

The Five-minute Rule 20 Years Later

and How Flash Memory Changes the Rules

written much more frequently (because of checkpoints) than is optimal based on Gray and Putzolu's formula.

In 1987, Gray and Putzolu speculated 20 years into the future and anticipated a "five-hour rule" for RAM and disks. For 1-KB records, prices and specifications typical in 2007 suggest 20,978 seconds, or just under six hours. Their prediction was amazingly accurate.

All break-even intervals are different for larger page sizes (e.g., 64 KB or even 256 KB). Table 3 shows the break-even intervals, including ones cited above, for a variety of page sizes and combinations of storage technologies. "\$400" stands for a 32-GB NAND flash drive available in the future for \$400 rather than for \$999 in 2007.

The old five-minute rule for RAM and disk now applies to page sizes of 64 KB (334 seconds). Five minutes had been the approximate break-even interval for 1 KB in 1987¹⁶ and for 8 KB in 1997.¹⁷ This trend reflects the different rates of improvement in disk-access latency and transfer bandwidth.

The five-minute break-even interval also applies to RAM and the expensive flash memory of 2007 for page sizes of 64 KB and above (365 seconds for 64 KB and 339 seconds for 256 KB). As the price premium for flash memory decreases, so does the break-even interval (146 and 136 seconds, respectively).

The two new five-minute rules promised here are indicated with values in ***bold italics*** in table 3. We will come back to this table and these rules in the discussion on optimal node sizes for B-tree indexes.

TABLE 3

Break-even Intervals (seconds)

Page size	1 KB	4 KB	16 KB	64 KB	256 KB
RAM-SATA	20,978	5,248	1,316	334	88
RAM-flash	2,513	876	467	365	339
Flash-SATA	32,253	8,070	2,024	513	135
RAM-\$400	1,006	351	187	146	136
\$400-SATA	80,553	20,155	5,056	1,281	<i>337</i>

PAGE MOVEMENT

In addition to I/O to and from RAM, a three-level memory hierarchy also requires data movement between flash memory and disk storage. The pure mechanism for moving pages can be realized in hardware (e.g., by DMA transfer), or it might require an indirect transfer via RAM. The former case promises better performance, whereas the latter design can be realized entirely in software without novel hardware. On the other hand, hybrid disk manufacturers might have cost-effective hardware implementations already available.

The policy for page movement is governed or derived from demand-paging and LRU replacement. As already discussed, replacement policies in both file systems and database systems may rely on LRU and can be implemented with appropriate data structures in RAM. As with buffer management in RAM, there may be differences resulting from prefetch, read-ahead, and write-behind. In database systems these may be directed by hints from the query execution layer, whereas file systems must detect page-access patterns and worthwhile read-ahead actions without the benefit of such hints.

If flash memory is part of the persistent storage, page movement between flash memory and disk is similar to page movement during defragmentation, both in file systems and in database systems. The most significant difference is how page movement and current page locations are tracked in these two kinds of systems.

TRACKING PAGE LOCATIONS

The mechanisms for tracking page locations are quite different in file systems and database systems. In file systems, pointer pages keep track of data pages or runs of contiguous data pages. Moving an individual page may require breaking up a run. It always requires updating and then writing a pointer page.

In database systems, most data is stored in B-tree indexes, including clustered (primary, nonredundant) and nonclustered (secondary, redundant) indexes on tables, materialized views, and database catalogs. Bitmap indexes, columnar storage, and master-detail clustering can readily and efficiently be represented in B-trees.¹⁸ Tree structures derived from B-trees are also used

for blobs (binary large objects) and are similar to the storage structures of some file systems.^{19,20}

For B-trees, moving an individual page can be very expensive or very cheap. The most efficient mechanisms are usually found in utilities for defragmentation or reorganization. Cost or efficiency results from two aspects of B-tree implementation: namely, maintenance of neighbor pointers and logging for recovery.

First, if physical neighbor pointers are maintained in each B-tree page, moving a single page requires updating two neighbors in addition to the parent node. If the neighbor pointers are logical using *fence keys*, only the parent page requires an update during a page movement.²¹ If the parent page is in memory, perhaps even pinned in the buffer pool, recording the new location is rather like updating an in-memory indirection array. The pointer change in the parent page is logged in the recovery log, but there is no need to force the log immediately to stable storage because this change is merely a structural change, not a database contents change.

Second, database systems log changes in the physical database, and in the extreme case both the deleted page image and the newly created page image are logged. Thus, an inefficient implementation fills two log pages whenever a single data page moves from one location to another. A more efficient implementation only logs allocation actions and delays de-allocation of the old page image until the new image is safely written in its intended location.²² In other words, moving a page from one location (e.g., on persistent flash memory) to another location (e.g., on disk) requires only a few bytes in the database recovery log.

The difference between file systems and database systems is the efficiency of updates enabled by the recovery log. In a file system, the new page location must be saved as soon as possible by writing a new image of the pointer page. In a database system, only a single log record or a few short log records must be added to the log buffer. Thus, the overhead for a page movement in a file system is writing an entire pointer page using a random access, whereas a database system adds a log record of a few dozen bytes to the log buffer that will eventually be written using large sequential write operations.

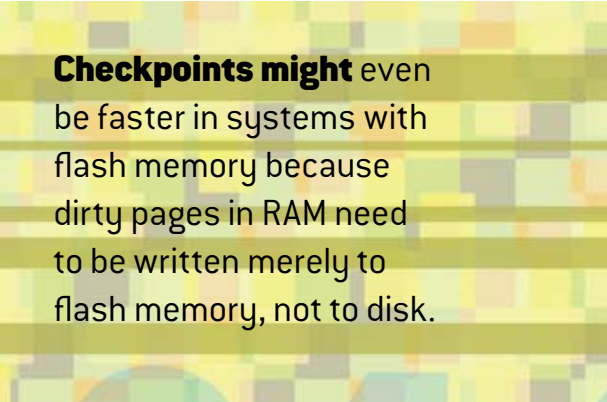
If a file system uses flash memory as persistent storage, moving a page between a flash memory location and an on-disk location adds substantial overhead. Thus, most file-system designers will probably prefer flash memory as an extension to the buffer pool rather than as an extension of the disk, thus avoiding this overhead.

A database system, however, has built-in mechanisms that can easily track page movements. These mechanisms are inherent in the “workhorse” data structure, B-tree indexes. Compared with file systems, these mechanisms permit efficient page movement, each one requiring only a fraction of a sequential write (in the recovery log) rather than a full random write.

Moreover, the database mechanisms are also reliable. Should a failure occur during a page movement, database recovery is driven by the recovery log, whereas a file system requires checking the entire volume during reboot.

CHECKPOINT PROCESSING

To ensure fast recovery after a system failure, database systems employ checkpoints. Their effect is that recovery needs to consider database activity later than the most recent checkpoint plus some limited activity explicitly



Checkpoints might even be faster in systems with flash memory because dirty pages in RAM need to be written merely to flash memory, not to disk.

indicated in the checkpoint information. The main effort is writing dirty pages from the buffer pool to persistent storage.

If pages in flash memory are considered part of the buffer pool, dirty pages must be written to disk during database checkpoints. Common checkpoint intervals are measured in seconds or minutes. Alternatively, if checkpoints are not truly points but intervals, it is reasonable to flush pages and perform checkpoint activities continuously, starting the next checkpoint as soon as one finishes. Thus, performing many writes to flash memory soon requires a write to disk, and flash memory as intermediate level in the memory hierarchy fails to absorb write activity. This effect may be exacerbated if, as discussed in the previous section, RAM is kept small because of the presence of flash memory.

If, on the other hand, flash memory is considered persistent storage, writing to flash memory is sufficient. Write-through to disk is required only as part of page

The Five-minute Rule 20 Years Later

and How Flash Memory Changes the Rules

replacement (i.e., when a page's usage suggests placement on disk rather than in flash memory). Thus, checkpoints do not incur the cost of moving data from flash memory to disk.

Checkpoints might even be faster in systems with flash memory because dirty pages in RAM need to be written merely to flash memory, not to disk. Given the very fast random access in flash memory relative to disk drives, this difference might speed up checkpoints significantly.

To summarize, database systems benefit if the flash memory is treated as part of the system's persistent storage. In contrast, traditional file systems do not have system-wide checkpoints that flush the recovery log and

any dirty data from the buffer pool. Instead, they rely on carefully writing modified file-system pages because of the lack of a recovery log protecting file contents.

PAGE SIZES

In addition to tuning based on the five-minute rule, another optimization based on access performance is sizing of B-tree nodes. The optimal page size minimizes the time spent on I/O during a root-to-leaf search. It balances a short access time (i.e., the desire for small pages) with a high reduction in remaining search space (i.e., the desire for large pages).

Assuming binary search within each B-tree node, the reduction in remaining search space is measured by the logarithm of the number of records within each node. This measure was called a node's *utility* in our earlier work.²³ This optimization is essentially equivalent to one described in the original research on B-trees.²⁴

Table 4 illustrates this optimization for records of 20 bytes—typical with prefix and suffix truncation²⁵—and nodes filled at about 70 percent. Not surprisingly, the

optimal node size for B-tree indexes on modern high-bandwidth disks is much larger than the page sizes employed in traditional database systems. With those disks, the access time dominates for all small page sizes, such that additional byte transfer and, therefore, additional utility are almost free.

B-tree nodes of 256 KB are very near optimal. For those, table 3 indicates a break-even time for RAM and disk of 88 seconds. For a \$400 flash disk and a traditional rotating hard disk, table 3 indicates 337 seconds or just over five minutes. This is the first of the two new five-minute rules presented here.

Table 5 illustrates the same calculations for B-trees on flash memory. Because of the lack of mechanical seeking and

TABLE 4

Page Utility for B-tree Nodes on Disk

Page size	Records / page	Node utility	Access time	Utility / time
4 KB	140	7	12.0ms	0.58
16 KB	560	9	12.1ms	0.75
64 KB	2,240	11	12.2ms	0.90
128 KB	4,480	12	12.4ms	0.97
256 KB	8,960	13	12.9ms	1.01
512 KB	17,920	14	13.7ms	1.02
1 MB	35,840	15	15.4ms	0.97

Page Utility for B-tree Nodes on Flash Memory

Page size	Records / page	Node utility	Access time	Utility / time
1 KB	35	5	0.11ms	43.4
2 KB	70	6	0.13ms	46.1
4 KB	140	7	0.16ms	43.6
8 KB	280	8	0.22ms	36.2
16 KB	560	9	0.34ms	26.3
64 KB	2,240	11	1.07ms	10.3

TABLE 5

rotation, the transfer time dominates even for small pages. The optimal page size for B-trees on flash memory is 2 KB, much smaller than for traditional disk drives.

In Table 3, the break-even interval for 4-KB pages is 351 seconds. This is the second new five-minute rule.

The implication of two different optimal page sizes is that any uniform node size for B-trees on flash memory and traditional rotating hard disks is suboptimal. Optimizing page sizes for both media requires a change in buffer management, space allocation, and some of the B-tree logic.

Fortunately, Patrick O'Neil of the University of Massachusetts has already designed a space allocation scheme for B-trees in which neighboring leaf nodes usually reside within the same contiguous extent of pages.²⁶ When a new page is needed for a node split, another page within the same extent is allocated. When an extent overflows, half its pages are moved to a newly allocated extent. In other words, the "split in half when full" logic of B-trees is applied not only to pages containing records but also to contiguous disk extents containing pages.

Using O'Neil's SB-trees (S meaning sequential), extents of 256 KB can be the units of transfer between flash memory and disk, whereas pages of 4 KB can be the unit of transfer between RAM and flash memory.

Similar notions of self-similar B-trees have also been proposed for higher levels in the memory hierarchy (e.g., in the form of B-trees of cache lines for the indirection vector within a large page).²⁷ Given that there are at least three levels of B-trees and three node sizes now (i.e., cache lines, flash memory pages, and disk pages), research into cache-oblivious B-trees²⁸ might be very promising.

QUERY PROCESSING

Self-similar designs apply both to data structures such as B-trees and to algorithms. For example, sort algorithms already use algorithms similar to traditional external merge sorts in multiple ways, to merge runs not only on disk but also in memory, where the initial runs in memory are sized to limit run creation to the CPU cache.^{29,30}

The same technique might be applied three times instead of twice: first, cache-sized runs in memory are merged into memory-sized runs in flash memory; second, in very large sort operations, memory-sized runs on flash memory are merged into runs on disk; and third, runs on disk are merged to form the final sorted result. Read-ahead, forecasting, write-behind, and page sizes all deserve a new look in a multilevel memory hierarchy consisting of cache, RAM, flash memory, and traditional disk drives. These page sizes can then inform the break-

even calculation for page retention versus I/O and thus guide the optimal capacities at each memory level.

We can surmise that a variation of this sort algorithm will be not only fast but also energy efficient. While energy efficiency has always been crucial for battery-powered devices, research into energy-efficient query processing on server machines is only now beginning.³¹ For example, both for flash memory and for disks, the energy-optimal page sizes might well differ from the performance-optimal page sizes.

The I/O pattern of external merge sort is similar (albeit in the opposite direction) to the I/O pattern of external partition sort, as well as to the I/O pattern of partitioning during hash join and hash aggregation. The latter algorithms, too, require reevaluation and redesign in a three-level memory hierarchy, or even a four-level memory hierarchy if CPU caches are also considered.³²

Flash memory with its very fast access times may revive interest in index-based query execution.^{33,34} Optimal page sizes and turn-over times are those derived in the earlier sections.

RECORD AND OBJECT CACHES

Page sizes in database systems have grown over the years, although not as fast as disk-transfer bandwidth. On the other hand, small pages require less buffer-pool space for each root-to-leaf search. For example, consider an index with 20 million entries. With index pages of 128 KB and 4,500 records, a root-to-leaf search requires two nodes and, thus, 256 KB in the buffer pool, although half of that (the root node) can probably be shared with other transactions. With index pages of 8 KB and 280 records per page, a root-to-leaf search requires three nodes or 24 KB in the buffer pool, or one order of magnitude less.

In the traditional database architecture, the default page size is a compromise between efficient index search (using large B-tree nodes as previously discussed and already in the original B-tree papers³⁵) and moderate buffer-pool requirements for each index search. Nonetheless, this example requires 24 KB in the buffer pool for finding a record of perhaps only 20 bytes, and it requires 8 KB of the buffer pool for retaining these 20 bytes in memory. An alternative design uses large on-disk pages and a record cache that serves applications, because record caches minimize memory needs yet provide the desired data retention. In-memory databases used as front ends for traditional disk-based databases represent a specific form of record caches.

The introduction of flash memory with its fast access latency and its small optimal page size may render record

The Five-minute Rule 20 Years Later

and How Flash Memory Changes the Rules

caches obsolete. With the large on-disk pages in flash memory and only small pages in the in-memory buffer pool, the desired compromise can be achieved without the need for two separate data structures (i.e., a transacted B-tree and a separate record cache).

In object-oriented applications that assemble complex objects from many tables and indexes in a relational database, the problem may be either better or worse, depending on the B-tree technology. If traditional indexes are used with a separate B-tree for each record format, assembling a complex object in memory requires many root-to-leaf searches and, therefore, many B-tree nodes in the buffer pool. If records from multiple indexes can be interleaved within a single B-tree based on their common search keys and sort order^{36,37} (e.g., on object identifier plus appropriate additional keys), very few or even a single B-tree search may suffice. Moreover, the entire complex object may be retained in a single page within the buffer pool.

DIRECTIONS FOR FUTURE WORK

Several directions for future research suggest themselves. First, the analyses presented in this article are focused on purchasing costs. Other costs could also be taken into consideration to capture total cost of ownership. Perhaps most interestingly, a focus on energy consumption may lead to different break-even points or even entirely different conclusions. Along with CPU scheduling, algorithms for staging data in the memory hierarchy, including buffer-pool replacement and compression, may be the software techniques with the highest impact on energy consumption.

Second, the five-minute rule applies to permanent data and its management in a buffer pool. The optimal retention time for temporary data such as run files in sorting and overflow files in hash join and hash aggregation may be different. For sorting, as for B-tree searches, the goal should be to maximize the number of comparisons per unit of I/O time or per unit of energy spent on I/O. Focused research may lead to new insights about query processing in multilevel memory hierarchies.

Third, Gray and Putzolu offered further rules of

thumb, such as the 10-byte rule for trading memory and CPU power. These rules also warrant revisiting for both costs and energy. Compared with 1987, the most fundamental change may be that CPU power should be measured not in instructions but in cache line replacements. Trading off space and time seems like a new problem in this environment.

Fourth, what are the best data-movement policies? One extreme is a database administrator explicitly moving entire files, tables, and indexes between flash memory and traditional disks. Another extreme is automatic movement of individual pages, controlled by a replacement policy such as LRU. Intermediate policies may focus on the roles of individual pages within a database or on the current query processing activity. For example, catalog pages may be moved after schema changes to facilitate fast recompilation of all cached query execution plans, and upper B-tree levels may be prefetched and cached in RAM or in flash memory during execution of query plans relying on index-to-index navigation.

Fifth, what are the secondary effects of introducing flash memory into the memory hierarchy of a database server? For example, short access times permit a lower multiprogramming level, because only short I/O operations must be “hidden” by asynchronous I/O and context switching. A lower multiprogramming level in turn may reduce contention for memory in sort and hash operations and for locks (concurrency control for database contents) and latches (concurrency control for in-memory data structures). Should this effect prove significant, the effort and complexity of using a fine granularity of locking may be reduced.

Sixth, how will flash memory affect in-memory database systems? Will they become more scalable, affordable, and popular based on memory inexpensively extended with flash memory rather than RAM? Will they become less popular because very fast traditional database systems use flash memory instead of (or in addition to) disks? Can a traditional code base using flash memory instead of traditional disks compete with a specialized in-memory database system in terms of performance, total cost of ownership, development and maintenance costs, time to market of features and releases, etc.?

Finally, techniques similar to generational garbage collection may benefit storage hierarchies. Selective reclamation applies not only to unreachable in-memory objects but also to buffer-pool pages and favored locations on permanent storage. Such research may also provide guidance for log-structured file systems, wear leveling for flash memory, and write-optimized B-trees on RAID storage.

SUMMARY AND CONCLUSIONS

The 20-year-old five-minute rule for RAM and disks still holds, but for ever-larger disk pages. Moreover, it should be augmented by two new five-minute rules: one for small pages moving between RAM and flash memory and one for large pages moving between flash memory and traditional disks. For small pages moving between RAM and disk, Gray and Putzolu were amazingly accurate in predicting a five-hour break-even point 20 years into the future.

Research into flash memory and its place in system architectures is urgent and important. Within a few years, flash memory will be used to fill the gap between traditional RAM and traditional disk drives in many operating systems, file systems, and database systems.

Flash memory can be used to extend RAM or persistent storage. These models are called “extended buffer pool” and “extended disk” here. Both models may seem viable in operating systems, file systems, and database systems. Because of the characteristics of these systems, however, they will employ different usage models.

In both models, the contents of RAM and flash will be governed by LRU-like replacement algorithms that attempt to keep the most valuable pages in RAM and the least valuable pages on traditional disks. The linked list or other data structure implementing the replacement policy for the flash memory will be maintained in RAM.

Operating systems and file systems will use flash memory mostly as transient memory (e.g., as a fast backup store for virtual memory and as a secondary file-system cache). Both of these applications fall into the extended buffer-pool model. During an orderly system shutdown, the flash memory contents might be written to persistent storage. During a system crash, however, the RAM-based description of flash-memory contents will be lost and must be reconstructed by a contents analysis very similar to a traditional file-system check. Alternatively, flash-memory contents can be voided and reloaded on demand.

Database systems, on the other hand, will employ flash memory as persistent storage, using the extended-disk model. The current contents will be described in persistent data structures (e.g., parent pages in B-tree indexes). Traditional durability mechanisms—in particular, logging and checkpoints—ensure consistency and efficient recovery after system crashes. An orderly system shutdown has no need to write flash-memory contents to disk.

There are two reasons for these different usage models for flash memory. First, database systems rely on regular checkpoints during which dirty pages are flushed from

the buffer pool to persistent storage. Moving a dirty page from RAM to the extended buffer pool in flash memory creates substantial overhead during the next checkpoint. A free buffer must be found in RAM, the page contents must be read from flash memory into RAM, and then the page must be written to disk. Adding such overhead to checkpoints is not attractive in database systems with frequent checkpoints. Operating systems and file systems, on the other hand, do not rely on checkpoints and thus can exploit flash memory as an extended buffer pool.

Second, the principal persistent data structures of databases, B-tree indexes, provide precisely the mapping and location-tracking mechanisms needed to complement frequent page movement and replacement. Thus, tracking a data page when it moves between disk and flash relies on the same data structure maintained for efficient database search. In addition to avoiding buffer descriptors, etc., for pages in flash memory, avoiding indirection in locating a page also makes database searches as efficient as possible.

Finally, as the ratio of access latencies and transfer bandwidth is very different for flash memory and disks, different B-tree node sizes are optimal. O’Neil’s SB-tree exploits two nodes sizes as needed in a multilevel storage hierarchy. The required inexpensive mechanisms for moving individual pages are the same as those required when moving pages between flash memory and disk. ◻

ACKNOWLEDGMENTS

This article is dedicated to Jim Gray, who suggested this research and helped me and many others many times in many ways. Barb Peters, Lily Jow, Harumi Kuno, José Blakeley, Mehul Shah, and the DaMoN 2007 reviewers suggested multiple improvements after reading earlier versions of this article.

REFERENCES

1. Gray, J., Putzolu, G.R. 1987. The 5-minute rule for trading memory for disk accesses and the 10-byte rule for trading memory for CPU time. *SIGMOD Record* 16(3): 395-398.
2. Gray, J., Graefe, G. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record* 26(4): 63-68.
3. Larus, J.R., Rajwar, R. 2007. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool.
4. Hamilton, J. 2007. An architecture for modular data centers. CIDR (Conference on Innovative Data Systems Research).

The Five-minute Rule 20 Years Later

and How Flash Memory Changes the Rules

5. Gray, J., Fitzgerald, B. Flash Disk Opportunity for Server-Applications; <http://research.microsoft.com/~gray/papers/FlashDiskPublic.doc>.
6. Härder, T., Reuter, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys* 15(4): 287-317.
7. Chen, P.M., Lee, E.L., Gibson, G.A., Katz, R.H., Patterson, D.A. 1994. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys* 26(2): 145-185.
8. See reference 7.
9. Ousterhout, J.K., Douglass, F. 1989. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review* 23(1): 11-28.
10. Woodhouse, D. 2001. JFFS: the Journaling Flash File System. Ottawa Linux Symposium. Red Hat Inc.
11. See reference 1.
12. See reference 1.
13. See reference 2.
14. See reference 1.
15. See reference 5.
16. See reference 1.
17. See reference 2.
18. Graefe, G. 2007. Master-detail clustering using merged indexes. *Informatik – Forschung und Entwicklung* 21 (3-4): 127-145.
19. Carey, M.J., DeWitt, D.J., Richardson, J.E., Shekita, E.J. 1989. Storage management in EXODUS. *Object-Oriented Concepts, Databases, and Applications*: 341-369.
20. Stonebraker, M. 1981. Operating system support for database management. *Communications of the ACM* 24(7): 412-418.
21. Graefe, G. 2004. Write-optimized B-trees. *Vldb*: 672-683.
22. See reference 21.
23. See reference 2.
24. Bayer, R., McCreight, E.M. 1970. Organization and maintenance of large ordered indexes. *SIGFIDET Workshop*: 107-141.
25. Bayer, R., Unterauer, K. 1977. Prefix B-trees. *ACM Transactions on Database Systems* 2(1): 11-26.
26. O’Neil, P.E. 1992. The SB-Tree: An index-sequential structure for high-performance sequential access. *Acta Informatica* 29(3): 241-265.
27. Lomet, D.B. 2001. The evolution of effective B-tree page organization and techniques: A personal account. *SIGMOD Record* 30(3): 64-69.
28. Bender, M.A., Demaine, E.D., Farach-Colton, M. 2005. Cache-oblivious B-trees. *SIAM Journal on Computing* 35(2): 341-358.
29. Graefe, G. 2006. Implementing sorting in database systems. *ACM Computing Surveys* 38(3).
30. Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D.B. 1995. AlphaSort: A cache-sensitive parallel external sort. *Vldb Journal* 4(4): 603-627.
31. Rivoire, S., Shah, M., Ranganathan, P., Kozyrakis, C. 2007. JouleSort: A balanced energy-efficiency benchmark. *SIGMOD*.
32. Shatdal, A., Kant, C., Naughton, J.F. 1994. Cache-conscious algorithms for relational query processing. *Vldb*: 510-521.
33. DeWitt, D.J., Naughton, J.F., Burger, J. 1993. Nested loops revisited. *PDIS*: 230-242.
34. Graefe, G. 2003. Executing nested queries. *BTW*: 58-77.
35. See reference 24.
36. See reference 18.
37. Härder, T. 1978. Implementing a generalized access path structure for a relational database system. *ACM Transactions on Database Systems* 3(3): 285-298.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

GOETZ GRAEFE (Goetz.Graefe@HP.com) joined Hewlett-Packard Laboratories after seven years as an academic researcher and teacher followed by 12 years as a product architect and developer at Microsoft, and was recently named an HP Fellow. Within the field of database management, his work bridges compile-time query optimization, runtime query execution, and indexing technologies. His Volcano research project was awarded the 10-year Test-of-Time Award at ACM SIGMOD 2000 for work on query execution, and he received the inaugural Influential Paper Award at the IEEE International Conference on Data Engineering in 2005 for his work on query optimization.

© 2008 ACM 1542-7730/08/0700 \$5.00

This article was originally published in Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007), June 15, 2007, Beijing, China.