The five-minute rule twenty years later, and how flash memory changes the rules

Goetz Graefe

HP Labs, Palo Alto, CA

Abstract

In 1987, Gray and Putzolo presented the five-minute rule, which was reviewed and renewed ten years later in 1997. With the advent of flash memory in the gap between traditional RAM main memory and traditional disk systems, the five-minute rule now applies to large pages appropriate for today's disks and their fast transfer bandwidths, and it also applies to flash disks holding small pages appropriate for their fast access latency.

Flash memory fills the gap between RAM and disks in terms of many metrics: acquisition cost, access latency, transfer bandwidth, spatial density, and power consumption. Thus, within a few years, flash memory will likely be used heavily in operating systems, file systems, and database systems. Research into appropriate system architectures is urgently needed.

The basic software architectures for exploiting flash in these systems are called "extended buffer pool" and "extended disk" here. Based on the characteristics of these software architectures, an argument is presented why operating systems and file systems on one hand and database systems on the other hand will best benefit from flash memory by employing different software architectures.

1 Introduction

In 1987, Gray and Putzolo published their now-famous five-minute rule [GP 87] 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 to perform disk I/O each time the record (or page) is accessed, using appropriate fractions of prices for RAM chips and for 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.

Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007), June 15, 2007, Beijing, China.

Based on then-current prices and performance characteristics of Tandem equipment, they found that the price of RAM memory to hold a record of 1 KB was about equal to the (fractional) price of a disk drive required to access such a record every 400 seconds, which they rounded to 5 minutes. The break-even interval is about inversely proportional to the record size. Gray and Putzolo gave 1 hour for records of 100 bytes and 2 minutes for pages of 4 KB.

The five-minute rule was reviewed and renewed ten years later in 1997 [GG 97]. Lots of prices and performance parameters had changed, e.g., the price for RAM memory had tumbled from $5,000 to $15 per megabyte. Nonetheless, the break-even interval for pages of 4 KB was still around 5 minutes. The first purpose of this paper is to review the five-minute rule after another ten years.

Of course, both prior papers acknowledge 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 re-evaluate the appropriate formulas for their environments and equipment. The values used in this paper, e.g., in Table 1, are meant to be typical for today's technologies rather than universally accurate.

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

Table 1. Prices and performance of flash and disks.

In addition to quantitative changes in prices and performance, qualitative changes already underway will affect the software and hardware architectures of servers and in particular of 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 in hardware [LR 07], deployment in containers housing 1,000s of processors and many TB of data [H 07], 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 [GF 07]. Table 1 and some derived metrics in Table 2 illustrate this point. (From dramexchange.com, dvnation.com, buy.com, seagate.com, and samsung.com; all 4/11/2007).

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 very 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 memory slot, via a SATA disk interface, or via 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 250 GB traditional disk, does the software treat it as 250 GB persistent storage and a 9 GB buffer pool, or as 258 GB persistent storage and a 1 GB buffer pool? The second purpose of this paper is to answer this question, and in fact to argue for different answers in file systems and in database systems.

	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

Table 2. Relative costs for flash memory and disks.

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

Moreover, the characteristics of flash memory suggest some substantial differences in the management of B-tree pages and their allocation. Beyond optimization of page sizes, we argue that B-trees will use different units of I/O for flash memory and for disks. Presenting the case for this design is the third purpose of this paper.

2 Assumptions

Forward-looking research always relies on many assumptions. This section attempts to list the assumptions that lead to our conclusions. Some of the assumptions seem fairly basic while others are more speculative.

One of our assumptions is that file systems and database systems assign to the flash memory between RAM and the disk drive. 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 follows the usual lines, e.g., LRU.

We assume that the administration of such information employs data structures in RAM memory, 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 memory and the 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 by 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.

2.1 File systems

In our research, we assumed a fairly traditional file system. Many file systems differ in some way or another from this model, but it seems that most usage of file systems still follows this model in general.

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 that model, space allocation on disk attempts to employ contiguous disk blocks for each file. Metadata are 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 that 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, a number of our arguments need to be revisited.

2.2 Database systems

We assume fairly traditional database systems with B-tree indexes as the "work horse" storage structure. Similar tree structures capture not only traditional clustered and non-clustered indexes but also bitmap indexes, columnar storage, contents indexes, XML indexes, catalogs (metadata), 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.

Variations such as "second chance" checkpoints or fuzzy checkpoints are included in our assumptions. In addition, "non-logged" (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 [HR 83].

2.3 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 mother board, on a memory DIMM slot, on a PCI board, or within a standard disk enclosure. In all cases, we assume DMA transfers (or something better) between RAM and flash memory. Moreover, we assume that either there is efficient DMA data transfer between flash and disk or there is a transfer buffer in RAM. The size of such 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.

We also assume 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 also used in Table 1. If necessary, the transfer bandwidth can be increased by use of array arrangements as well known for disk drives [CLG 94]. Even redundant arrangement of flash memory may prove advantageous in some cases [CLG 94].

Since the reliability of current NAND flash suffers after 100,000 – 1,000,000 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 similarly often. It is important to recognize the similarity between wear leveling algorithms and log-structured file systems [OD 89, W 01], although the former also move stable, unchanged data such that their locations can also 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, 6 years of continuous and sustained writing at 100 MB/sec overwrites an entire 250 GB disk less 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/s 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 clean-up activity that prepares space for future log writing. The difference is 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 a 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.

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

As mentioned above, we assume page replacement on demand. In addition, there may be automatic policies and mechanisms for prefetch, read-ahead, write-behind.

Based on these considerations, we assume that the contents of a 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. However, limitations in write bandwidth discourage such use. 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.

2.4 Other hardware

In all cases, we assume RAM memory of a substantial size, although probably less than flash memory or disk. The relative sizes should be governed "five-minute rule" [GP 87]. Note that, despite similar transfer bandwidth, the short access latency of flash memory compare to disk results in surprising retention times for data in RAM memory, as discussed below.

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 run-time system) permits highly concurrent maintenance of complex data structures. For example, page replacement heuristics might employ 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.

3 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 Putzolo [GP 87], we focus here 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 for moving data.

Gray and Putzolo introduced the formula BreakEvenIntervalinSeconds = (PagesPerMBofRAM / AccessesPerSecondPerDisk) × (PricePerDiskDrive / PricePerMBofRAM) [GG 97, GP 87]. 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 pages of 4 KB, and the values from Table 1 and Table 2, this produces (256 / 83) × ($80 / $0.047) = 5,248 seconds = 90 minutes = 1½ hours[1]. This compares to 2 minutes (for pages of 4 KB) 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 memory was estimated in 1987 at about $5,000 per MB whereas today the cost is about $0.05 per MB, 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) × ($999 / $0.047) = 876 seconds = 15 minutes. If today's 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 Fitzgerald [GF 07]), 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 memory 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 interestingly, applying the same formula to flash and disk gives (256 / 83) × ($80 / $0.03) = 8,070 seconds = 2¼ hours. Thus, all active data will remain in RAM and flash memory.

Without doubt, 2 hours is longer than any common checkpoint interval, which implies that dirty pages in flash are forced to disk not by page replacement but always by checkpoints. Pages that are updated frequently must be written much more frequently (due to checkpoints) than is optimal based on Gray and Putzolo's formula.

In 1987, Gray and Putzolo speculated 20 years into the future and anticipated a "five-hour rule" for RAM and disks. For records of 1 KB, today's devices suggest 20,978 seconds or a bit less than 6 hours. Their prediction was amazingly accurate.

Page size	1KB	4KB	16KB	64KB	256KB
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	***337***

Table 3. Break-even intervals [seconds].

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.

[1] The "=" sign often indicates rounding in this paper.

"$400" stands for a 32 GB NAND flash drive available in the future for $400 rather than for $999 today.

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 1987 [GP 87] and for 8 KB in 1997 [GG 97]. 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 today's expensive flash memory for page sizes of 64 KB and above (365 and 339 seconds). As the price premium for flash memory decreases, so does the break-even interval (146 and 136 seconds).

The two new five-minute rules promised in the abstract 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.

4 Page movement

In addition to I/O to and from RAM memory, 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 memory. 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 discussed above, replacement policies in both file systems and database systems may rely on LRU and can be implemented with appropriate data structures in RAM memory. As with buffer management in RAM memory, there may be differences due to prefetch, read-ahead, and write-behind, which in database systems 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 very similar to page movement during defragmentation, both in file systems and in database systems. Perhaps the most significant difference is how page movement and current page locations are tracked in these two kinds of systems.

5 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 of 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 and non-clustered indexes on tables, materialized views, and metadata catalogs. Bitmap indexes, columnar storage, and master-detail clustering can readily and efficiently be represented in B-tree indexes [G 07]. Tree structures derived from B-trees are also used for binary large objects ("blobs") and are similar to the storage structures of some file systems [CDR 89, S 81].

For B-trees, moving an individual page can range from very expensive to very cheap. The most efficient mechanisms are usually found in utilities for defragmentation or reorganization. Cost or efficiency result 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 [G 04]. 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 produces two full 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 [G 04]. 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 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 store, moving a page between a flash memory location and an on-disk location adds substantial overhead. Thus, we believe that file system designers will prefer flash memory as extension to the buffer pool rather than 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 "work horse" data structure, B-tree indexes. In comparison to file systems, these mechanisms permit very efficient page movement. Each page movement requires only a fraction of a sequential write (in the recovery log) rather than a full random write.

Moreover, the database mechanisms are also very 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 storage during reboot.

6 Checkpoint processing

To ensure fast recovery after a system failure, database systems employ checkpoints. Their effect is that recovery only needs to consider database activity later than the most recent checkpoint plus some limited activity explicitly indicated in the checkpoint information. This effect is achieved partially by writing dirty pages in buffer pool.

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 even reasonable to flush pages and perform checkpoint activities continuously, starting the next one as soon as one finishes. Thus, many writes to flash memory will soon require 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 memory is kept small due to the presence of flash memory.

If, on the other hand, flash memory is part of the persistent storage, writing to flash memory is sufficient. Write-through to disk is required only as part of page 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 memory 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 managed as part of the persistent storage. In contrast, traditional file systems do not have system-wide checkpoints that flush the recovery log and any dirty data in the buffer pool. Instead, they rely on carefully writing modified file system pages due to the lack of a recovery log protecting file contents.

7 Page sizes

In addition to the tuning based on the five-minute rule, another optimization based on access performance is sizing of B-tree nodes. The optimal page size combines a short access time with a high reduction in remaining search space. Assuming binary search within each B-tree node, the latter is measured by the logarithm of records within a node. This measure was called a node's "utility" in our earlier work [GG 97]. This optimization is essentially equivalent to one described in the original research on B-trees [BM 70].

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

Table 4. Page utility for B-tree nodes on disk.

Table 4 illustrates this optimization for records of 20 bytes, typical if prefix and suffix truncation [BU 77] are employed, and nodes filled at about 70%.

Not surprisingly, the optimal page size for B-tree indexes on modern high-bandwidth disks is much larger than traditional database systems have employed. The access time dominates for all small page sizes, such that additional byte transfer and thus 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 5 minutes. This is the first of the two five-minute rules promised in the abstract.

Page size	Records per 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. Page utility for B-tree nodes on flash memory.

Table 5 illustrates the same calculations for B-trees on flash memory. Due to the lack of mechanical seeking and 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 pages of 4 KB is 351 seconds. This is the second five-minute rule promised in the abstract.

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

Fortunately, O'Neil already designed a space allocation scheme for B-trees in which neighboring leaf nodes usually

reside within the same contiguous extent of pages [O 92]. When a new page is needed for a node split, another page within the same extent is allocated. When extent overflows, half its pages moved to a newly allocated extent.

Using O'Neil's SB-trees, extents of 256 KB are the units of transfer between flash memory and disk, whereas pages of 4 KB are 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 [L 01]. Given that there are at least 3 levels of B-trees and 3 node sizes now, i.e., cache lines, flash memory pages, and disk pages, research into cache-oblivious B-trees [BDC 05] might be very promising.

8 Query processing

Self-similar designs apply both to data structures such as B-trees and to algorithms. For example, sort algorithms already employ algorithms similar to traditional external merge sort in multiple ways, not only to merge runs on disk but also to merge runs in memory, where the initial runs in memory are sized to limit run creation to the CPU cache [G 06, NBC 95].

The same technique might be applied three times instead of two, i.e., runs in memory are merged into runs in flash memory, and for very large sort operations, runs on flash memory are merged into runs on disk. Read-ahead, forecasting, write-behind, and page sizes all deserve a new look in a multi-level 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.

It may be surmised that a variation of this sort algorithm will not only be 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 [RSK 07]. 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 re-evaluation and -design in a three-level memory hierarchy, or even a four-level memory hierarchy is CPU caches are also considered [SKN 94].

Flash memory with its very fast access times may revive interest in index-based query execution [DNB 93, G 03]. Optimal pages size and turn-over times are those derived in the earlier sections.

9 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 20,000,000 entries. With index pages of 128 KB and 4,500 records, a root-to-leaf search requires 2 nodes and thus 256 KB in the buffer pool, although half it 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 3 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 discussed above and already in the original B-tree papers [BM 70]) and moderate buffer pool requirements for each index search. Nonetheless, the example above 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 employs large on-disk pages and a record cache that serves applications, because record cache minimize memory needs yet provide the desired data retention.

The introduction of flash memory with its fast access latency and its small optimal page size may render record 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 employed. 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 thus 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 [G 07, H 78], 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.

10 Directions for future work

Several directions for future research suggest themselves. We plan on pursuing multiple of these in the future.

First, the analyses in this paper are focused on purchasing costs. Other costs could be taken into consideration in order 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 their 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 multi-level memory hierarchies.

Third, Gray and Putzolo offered further rules of thumb, e.g., the ten-byte rule for trading memory and CPU power. These rules also warrant revisiting for both costs and energy. Compared to 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 disk. 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 caches query execution plans, and upper B-tree levels may be prefetched and cached in RAM memory or in flash memory during execution of query plans relying on index navigation.

Fifth, what are secondary effects of introducing flash memory into the memory hierarchy of a database server? For example, short access times permit a lower multi-programming level, because only short I/O operations must be "hidden" by asynchronous I/O and context switching. A lower multi-programming level in turn may reduce contention for memory in sort and hash operations and for locks and latches (concurrency control for in-memory data structures). Should this effect prove significant, 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 memory? Will they become less popular due to very fast traditional database systems using 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 also may provide guidance for log-structured file systems, for wear leveling for flash memory, and for write-optimized B-trees on RAID storage.

11 Summary and conclusions

In summary, 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 large pages moving between RAM and flash memory and for small pages moving between flash memory and disks. For small pages moving between RAM and disk, Gray and Putzolo 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 memory and traditional disk drives in many operating systems, file systems, and database systems.

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

In both models, contents of RAM and of 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 employ 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 be 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 in the buffer pool are flushed to persistent storage. If a dirty page is moved from RAM to the extended buffer pool in flash memory, it cre-

ates 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 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 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 for disks, different B-tree node sizes are optimal. O'Neil's SB-tree exploits two nodes sizes as needed in a multi-level storage hierarchy. The required inexpensive mechanisms for moving individual pages are the same as those required when moving pages between flash memory and disk.

Acknowledgements

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

References

[BDC 05] Michael A. Bender, Erik D. Demaine, Martin Farach-Colton: Cache-Oblivious B-Trees. SIAM J. Comput. 35(2): 341-358 (2005).

[BM 70] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indexes. SIGFIDET Workshop 1970: 107-141.

[BU 77] Rudolf Bayer, Karl Unterauer: Prefix B-Trees. ACM TODS 2(1): 11-26 (1977).

[CDR 89] Michael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. Shekita: Storage Management in EXODUS. Object-Oriented Concepts, Databases, and Applications 1989: 341-369.

[CLG 94] Peter M. Chen, Edward L. Lee, Garth A. Gibson, Randy H. Katz, David A. Patterson: RAID: High-Performance, Reliable Secondary Storage ACM Comput. Surv. 26(2): 145-185 (1994).

[DNB 93] David J. DeWitt, Jeffrey F. Naughton, Joseph Burger: Nested Loops Revisited. PDIS 1993: 230-242.

[G 03] Goetz Graefe: Executing Nested Queries. BTW 2003: 58-77.

[G 04] Goetz Graefe: Write-Optimized B-Trees. VLDB 2004: 672-683.

[G 06] Goetz Graefe: Implementing Sorting in Database Systems. ACM Comput. Surv. 38(3): (2006).

[G 07] Goetz Graefe: Master-detail clustering using merged indexes. Informatik – Forschung und Entwicklung (2007).

[GF 07] Jim Gray, Bob Fitzgerald: FLASH Disk Opportunity for Server-Applications. http://research.microsoft.com/~gray/papers/FlashDiskPublic.doc.

[GG 97] Jim Gray, Goetz Graefe: The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Record 26(4): 63-68 (1997).

[GP 87] Jim Gray, Gianfranco R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. SIGMOD 1987: 395-398.

[H 78] Theo Härder: Implementing a Generalized Access Path Structure for a Relational Database System. ACM TODS 3(3): 285-298 (1978).

[H 07] James Hamilton: An Architecture for Modular Data Centers. CIDR 2007.

[HR 83] Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15(4): 287-317 (1983).

[L 01] David B. Lomet: The Evolution of Effective B-tree Page Organization and Techniques: A Personal Account. SIGMOD Record 30(3): 64-69 (2001).

[LR 07] James R. Larus, Ravi Rajwar: Transactional Memory. Synthesis Lectures on Computer Architecture, Morgan & Claypool (2007).

[NBC 95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, David B. Lomet: AlphaSort: A Cache-Sensitive Parallel External Sort VLDB J. 4(4): 603-627 (1995).

[OD 89] John K. Ousterhout, Fred Douglis: Beating the I/O Bottleneck: A Case for Log-Structured File Systems. Operating Systems Review 23(1): 11-28 (1989).

[O 92] Patrick E. O'Neil: The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. Acta Inf. 29(3): 241-265 (1992).

[RSK 07] Suzanne Rivoire, Mehul Shah, Partha Ranganathan, Christos Kozyrakis: JouleSort: A Balanced Energy-Efficiency Benchmark. SIGMOD 2007.

[S 81] Michael Stonebraker: Operating System Support for Database Management. CACM 24(7): 412-418 (1981).

[SKN 94] Ambuj Shatdal, Chander Kant, Jeffrey F. Naughton: Cache Conscious Algorithms for Relational Query Processing. VLDB 1994: 510-521.

[W 01] David Woodhouse: JFFS: the Journaling Flash File System. Ottawa Linux Symposium, Red Hat Inc, 2001.