Improving Database Performance Using a Flash-Based Write Cache

Yi Ou and Theo Härder

University of Kaiserslautern
{ou,haerder}@cs.uni-kl.de

Abstract. The use of flash memory as a write cache for a database stored on magnetic disks has been so far largely ignored. In this paper, we explore how flash memory can be efficiently used for this purpose and how such a write cache can be implemented. We systematically study the design alternatives, algorithms, and techniques for the flash-based write cache and evaluate them using trace-driven simulations, covering the most typical database workloads.

1 Introduction

Flash memory[1] is popularly used in a variety of data storage devices, such as compact flash cards, secure digital cards, flash SSDs, flash PCIe cards, etc., primarily due to its non-volatility and high density. Flash-based storage devices (or flash devices for short) can be manufactured with some interesting properties, such as low-power consumption, small form factor, shock resistance, etc. which make them attractive to a large range of applications.

Due to the physical constraints of flash memory such as erase-before-program and wear-out of memory cells [1], flash devices typically implement an additional layer, the flash translation layer (FTL) [2], on top of the flash memory to support typical host-to-device interfaces.

The function of magnetic disks (HDDs), the currently dominating mass storage devices, relies on mechanical moving parts, which is one of the major threats to the device reliability and typically the bottleneck of the entire system performance. In contrast, flash devices do not contain mechanical moving parts. Therefore, they allow much faster random access (up to two orders of magnitude) and much higher reliability.

Flash memory and flash devices have received a lot of attention from the database research community due to the fact that existing database systems are designed and optimized for HDDs that have quite different performance characteristics than flash devices, e. g., flash SSDs [3].

To exploit the performance potential of flash memory and flash devices in a database environment, flash-specific query processing techniques [4,5], index structures [6,7,8], and buffer management algorithms [9,10,11] have been proposed. Efficient use of flash devices for update propagation [12], logging and recovery [13], and transaction processing [14] has also been studied.

[1] We focus on the NAND type flash memory due to its high density and low cost.

Modern flash devices are much faster than HDDs, even for random write workloads. At the same time, they are also much more expensive than HDDs in terms of price per unit capacity. Some researchers realized that completely replacing HDDs with flash devices is not a cost-effective solution [15], instead, using flash as an intermediate tier between RAM-based memory and HDDs [16,17] can bridge the access-time gap between the two tiers without introducing much higher cost. However, it is doubtful whether such an abrupt architecture change will be widely accepted in practice, given the trend in industry to keep the whole working set in RAM for maximum performance [18,19].

1.1 Flash-Based Write Cache

In this paper, we study the use of flash as a write cache for databases based on HDDs. The flash-based write cache (or the flash-based second-tier cache) is a page cache layer between the RAM-based buffer pool (or first-tier cache) and the database stored in HDDs (the third tier). The accesses to all three tiers are via page-oriented interfaces, i. e., in units of database pages.

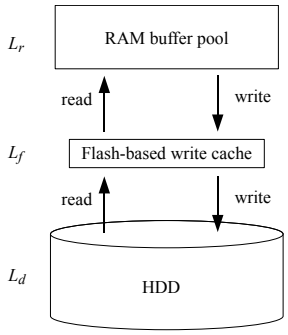


Fig. 1: Database storage system with flash-based write cache

The architecture of a three-tier database storage system are illustrated in Figure 1. It is very similar to the three-layer architecture (3LA) introduced in [17]. To be comparable, we adopt the notation of [17] to denote the RAM-based buffer pool as L_r, the flash-based write cache as L_f, and the third tier as L_d. Their capacities, in number of pages, are denoted as $|L_r|$, $|L_f|$, and $|L_d|$, respectively. As opposed to [17], which assumes that

$$|L_r| \leq |L_f| \leq |L_d| \tag{1}$$

we drop the constraint $|L_r| \leq |L_f|$ and assume that

$$|L_r| \ll |L_d| \text{ and } |L_r| \sim |L_f| \tag{2}$$

Note, according to Equation 2, $|L_r| > |L_f|$ is acceptable.

In practical systems, flash is used in different forms such as flash SSDs or flash PCIe cards. In our system, there is no constraint on which a specific form should

be used. Instead, various forms of flash devices are abstracted into *page-oriented devices with flash performance characteristics*. We consider four different costs for device access in our system: the costs of flash read, flash sequential write, flash random write, and the cost of disk access, denoted as C_{fr}, C_{fw}, $C_{f\tilde{w}}$, and C_d, respectively.

In such a configuration, dirty[2] pages evicted from the RAM buffer pool are first written to the flash-based write cache and later propagated to HDDs. Depending on specific replacement policies, the number of disk writes can be substantially reduced in this way (see Section 3). Even as a write cache, L_f should ideally keep the "warm" pages, i. e., pages that are not so hot to be kept in L_r, but warmer than the remaining database pages. Hence, whenever a page request can be satisfied from L_f, a cost benefit of $C_d - C_{fr}$ can be obtained.

There are multiple benefits of using such a write cache: 1. Compared with a volatile write cache, it provides higher reliability and protection against data loss at power failure. 2. Due to the cost advantage of flash to RAM, the write cache can be much larger and potentially more efficient than volatile and expensive battery-backed RAM-based write cache (The same argument in favor of flash applies to energy saving [17]). 3. A dedicated write cache improves the write response time and offloads write workloads from HDDs and can asynchronously propagate them to HDDs, i. e., the read performance can also benefit from the write cache. 4. Potential page hits in the write cache reduce the number of expensive disk writes.

Using flash for a write cache, the wear-out problem of flash cells seems to be a concern. However, with proper wear-leveling techniques, the life time of flash memory becomes quite acceptable. For example, an SLC flash memory module typically has a write endurance of 100,000 program/erase cycles (Write endurance of one million cycles has already been reported [20]). With perfect wear-leveling, i. e., all flash pages are programmed and erased at equal frequency, 10 GB flash memory can have a life span of 27.4 years[3] under a daily write workload of 100 GB (factor 10 of its capacity).

1.2 Contribution

Our major contributions are:

- The use of flash devices as a write cache for a database stored on HDDs has been so far largely ignored. To the best of our knowledge, our work is the first one that considers this usage.
- We systematically study the algorithms and techniques for flash-based write caches and evaluate them using trace-driven simulations, covering the most typical database workloads.

[2] In contrast to its use in transactional contexts, we denote modified pages as "dirty", as long as they are not written to disk.

[3] $100000 \times 10/(100 \times 365) = 27.4$

1.3 Organization

The remainder of this paper is organized as follows: Section 2 discusses related works. Section 3 presents and discusses the algorithms and techniques for the flash-based write cache. Section 4 reports our empirical study. The concluding remarks and future works are presented in Section 5.

2 Related Work

[16] is one of the pioneer works studying flash-aware multi-level caching. The authors identified three page-flow schemes in a three-level caching hierarchy with flash as the mid-tier and proposed flash-specific cost models for those schemes. In contrast, contribution [17] presented a detailed three-tier storage system design and performance analysis. In this study, the experiments have shown for certain range of applications, by reducing the amount of energy-hungry RAM-based memory and using a much larger amount of flash as the mid-tier, that system performance and energy efficiency can be both improved at the same time.

Canim et al. [21] proposed a temperature-aware replacement policy for managing the SSD-based mid-tier, based on the access statistics of disk regions (page groups). In [22], the authors studied three design alternatives of an SSD-based mid-tier, which differ mainly in the way how to deal with dirty pages evicted from the first-tier, e. g., write through or write back.

As a general assumption, all these approaches use a flash-based mid-tier being much larger than the first tier. As a consequence, both clean pages and dirty pages are cached in the mid-tier. In contrast, we focus on a configuration where the flash is used as a write cache, i. e., only dirty pages are cached in the mid-tier. More important, the above mentioned works only consider the cost asymmetry of reads and writes on flash, while the cost asymmetry between random and sequential flash writes are ignored, which can significantly impact the system performance, according to our experiments (see Section 4).

Using flash as a write cache has also been studied by Li et al. [23]. However, in their configuration, the database is completely stored in L_f and no L_d is considered, in contrast to the three-tier system (Figure 1) being studied by us. Their basic idea is to exploit the performance advantage of focused writes over random ones, by directing all the writes generated by L_r to a small logical flash area and reordering the writes so that they can be written back to their actual destinations on the same flash device, but in more efficient write patterns.

The authors of [24] and [25] have taken an approach that is somehow the "opposite" of ours. They consider the use of HDDs as the write cache for flash SSDs, based on the argument that the write performance of HDDs is better than that of some flash SSDs and HDDs don't have the wear-out problem. However, we believe that compared to HDDs, flash devices can be made much more reliable and they have a much higher performance potential. In fact, even the random write performance of many mid-range SSDs is now superior to that of the enterprise HDDs.

3 Algorithms and Techniques

Two design decisions are critical to the performance of flash-based caches:

- When should a page be admitted into the cache? This is specified by cache-admission strategies.
- How should admitted pages be written to the cache? This is specified by cache-writing strategies.

3.1 Cache-Admission Strategies

Following the architecture shown in Figure 1, there are only two cases where pages can be admitted into L_f: 1. *Admit-On-Read (AOR)*, i. e., when the read function of L_f is called; 2. *Admit-On-Write (AOW)*, i. e., when the write function of L_f is called.

The first case happens when a buffer fault occurs for page p in L_r. If p is not found in L_f either, it will be fetched from L_d and forwarded to L_r. After that, the newly fetched p can be admitted into L_f.

The second case happens when a dirty page p is evicted from L_r and the write function on L_f is called to write p back. Conceptually, pages admitted in this case are all dirty pages even if L_f is non-volatile, i. e., they need to be propagated to L_d at the latest, when they are evicted from L_f.

The GLB algorithm discussed in [17] actually uses a third cache-admission strategy, which we call *Admit-On-Eviction* (AOE), because it admits every page (either clean or dirty) evicted from L_r into L_f. However, AOE is not considered in this paper due to two problems: 1. It requires a write operation on L_f even on a cache hit in L_f (because the page currently hit has to be exchanged with a page from L_r); 2. It violates the transparency of L_f, because it requires for L_f an extension of the interface shown in Figure 1.

Cache-admission strategies are orthogonal to replacement polices, although classical second-tier cache algorithms such as LOC [17] and MQ [26] implicitly use AOR. In contrast, AOW is the cache-admission strategy used by our flash-based write cache.

3.2 Cache-Writing Strategies

According to the characteristics of flash memory, we consider two cache-writing strategies: *sequential cache write* (SCW) and *random cache write* (RCW). With SCW, pages are written to the flash media in a strictly sequential fashion (thus each write has a cost of C_{fw}, in contrast to $C_{f\tilde{w}}$ in case of RCW), as illustrated in Figure 2. If the flash-based cache has n pages and the most recently updated cache slot is $curr$, then the next cache slot to be used is given by $next = (curr+1) \bmod n$. If an earlier version of a newly admitted page p is already in L_f, it has to be *invalidated*, i. e., if multiple versions of p exist in L_f, only the newest version is valid.

Because updates to the flash media are sequential and the slots are updated with equal frequency, SCW enjoys two advantages: *write performance and wear leveling without the need of an FTL*. However, due to the constraint of strict sequential writes, SCW does not allow much flexibility in the choice of replacement victims, which is always predetermined by the *next* pointer. Even if the page cached at the slot pointed to by *next* is a "warm" page, we have to write it back to the disk in order to make room for the page to be cached.

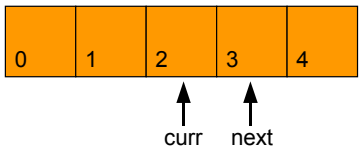


Fig. 2: Sequential cache write. Illustrated is a flash-based write cache with 5 pages. The slot number is marked at the bottom-left corner of the cache slots. Slot 2 is just written, the next slot to be used is slot 3.

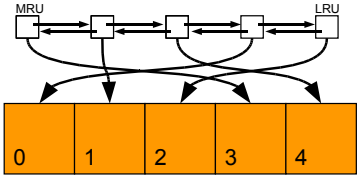


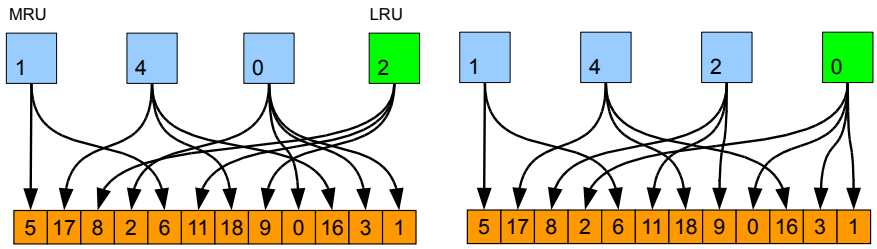
Fig. 3: Random cache write. In the example, the cache slots are ordered by their reference recency (as doubly linked list maintained in volatile memory), which implies the (random) order they are to be overwritten: 2, 0, 4, 1, 3

Random cache write (RCW), as the name suggests, allows writing to the flash-based cache in a random fashion. If a earlier version of a newly admitted page p is already in L_f, it has to be *overwritten*. The cost of writing to the cache is higher than in the SCW case, and wear-leveling mechanisms such as FTL become necessary. However, in contrast to SCW, RCW does not impose any restriction on the replacement policy. For example, the cache slots can be ordered by their reference recency (as shown in Figure 3) or frequency, upon which the replacement decision can be made, as in the classical buffer management algorithms.

3.3 Track-Aware Algorithms

Seeking is the most expensive mechanical movement made by a magnetic disk – typically a few milliseconds for modern disks. If the information about which page belongs to which track, i. e., the page-to-track mapping function t that maps a logical page number a to its corresponding track number $t(a)$, is known to the flash-based write cache, it is possible to further improve the disk write performance by minimizing the number of seeks. Assume that the mapping function t is given by $t(a) = a/4$, then pages 0, 1, 2, and 3 have the track number 0, pages 4 to 7 have the track number 1, and so on. Cache algorithms making use of this track information are called *track-aware* algorithms.

Virtual track is an in-memory data structure used by the track-aware algorithms. It contains the pointers to the set of pages belonging to the same track.



(a) By reference recency (LRU) **(b)** Densest track first

Fig. 4: Virtual tracks (depicted as large rectangles with track number in the bottom-left corner) ordered by reference recency (a) and by density (b). Replacement victims are shown in green. Page numbers of the cached pages are depicted as the numbers inside the flash slots.

When a free cache slot is needed and all slots are currently occupied, a virtual track can be chosen as the replacement victim, and the m pages pointed to by it are all flushed to disk at once, freeing m pages and requiring only one seek. This technique is called *coalesced flushing (CF)*.

The set of virtual tracks can be ordered by their reference recency, i. e., the time that a page belonging to the track is referenced (i. e., read or updated). When a replacement victim is needed, the least-recently-referenced virtual track is chosen, similar to the LRU replacement policy. Note if the page-to-track mapping function is $t(a) = a$ (one page per track), the described algorithm degenerates to the classical page-oriented LRU (see Figure 3). Therefore, we refer to this algorithm also as LRU whenever there is no ambiguity. An example runtime state of the algorithm is shown in Figure 4a.

Virtual tracks can also be ordered by their density, i. e., by the number of pointers they contain. The replacement victim is then the densest track, as illustrated in Figure 4b. This replacement policy is called *densest track first* (DTF).

Using the track information, we can further reduce the number of seeks using a *piggy-backing (PB)* technique. When the flash-based cache has to serve a read request for a page belonging to track t, a seek to t is very likely inevitable, but the cache can flush all pages pointed to by the corresponding virtual track without enforcing further seeks. Figure 5 illustrates a PB example.

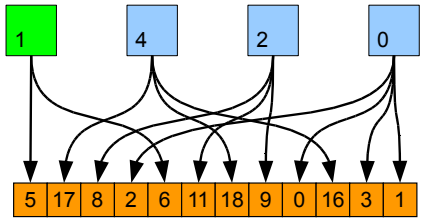


Fig. 5: Piggy-backing. In this sample, a read request for page 7 is to be served, thus a seek to track 1 is needed; in this case, the cache flushes pages 5 and 6 pointed to by virtual track 1.

3.4 Compatibility Matrix

The cache-write strategies SCW and RCW are orthogonal to the track-aware techniques. With SCW, it is also possible to perform coalesced flushing and piggy-backing. However, replacement policies, such as REC and DTF, are only compatible with RCW. Table 1 lists the compatibility relationships between the discussed techniques.

Table 1: Compatibility matrix of the techniques

	SCW	*RCW*
LRU, DTF, etc.		√
CF	√	√
PB	√	√

Table 2: Device access costs

metric	high-end	low-end
C_{fr}	0.105 ms	0.165 ms
C_{fw}	0.106 ms	0.153 ms
$C_{f\tilde{w}}$	0.133 ms	7.972 ms
C_d	4.464 ms	

3.5 Logging and Recovery Implications

The data structures used by the algorithms discussed in this section, e. g., the table mapping logical page numbers to their physical locations, are stored in a small RAM area. To prevent from data loss in case of a system crash, the logical page number should be stored in the page (either in the payload or in the page header) to be able to restore the mapping table by a recovery procedure.

For SCW, it is possible that multiple versions of the same page exist in the cache at recovery. The solution is to store a version number in the page which increments whenever the page is updated. At recovery, only the version with the highest version number generates an entry in the mapping table. The log sequence number (LSN) [27] can serve as the version number. In this case, no extra space is required.

Because of the persistence of the mid-tier, all our 2-step update propagation approaches can be combined with the classical logging methods, the WAL principle (Write Ahead Log), and the recovery-oriented concepts (Atomic/NoAtomic, Steal/NoSteal, Force/NoForce) for mapping database changes from volatile to non-volatile storage [28]. As a consequence, the 2-step mechanism can not only be used to accelerate the propagation of data pages to disk, but can also be applied to log information. Such a practice automatically minimizes transaction latency caused by commit processing, i.e., the 2PC protocol, because all synchronous writes are first directed to the flash layer and not directly to HDDs.

4 Experiments

We implemented our algorithms and used trace-driven simulations to evaluate their performance and study their behavior under various workloads. Our test

system consists of a disk layer supporting the block-device interface and collecting disk-access statistics, and a cache layer implementing the introduced algorithms and collecting flash-access statistics. Our traces contain the block-level accesses (physical page requests), collected with the help of the PostgreSQL database engine under TPC-C (100 warehouses) and TPC-H (scale factor: 10) benchmark workloads. The PostgreSQL engine was used to collect the buffer traces (logical page requests), which were then fed to and filtered by an LRU buffer pool of 10,000 pages (i. e., $|L_r| = 10,000$), and the resulting sequence of physical page requests make the block-level traces used in our experiments.

A test program parses the traces and generates block read and write requests, which are then served by the cache layer, either using the cached pages whenever possible or by accessing the disk layer if necessary. According to our three-tier storage architecture in Figure 1, only the bottom two tiers are used in our experiments. The observations made under the TPC-C workload are very similar to those made under the TPC-H workload. This means that our observations are not specific to the workload. For improved clarity, we choose to only report the experimental results collected using the TPC-C trace.

Based on our cost model introduced in Section 1.1 and with the variables n_{fr}, n_{fw}, $n_{f\tilde{w}}$, and n_d as the numbers for the related flash read, flash sequential write, flash random write, and disk accesses, we can define the performance metric *virtual execution time* (v) as:

$$v = n_{fr} \times C_{fr} + n_{fw} \times C_{fw} + n_{f\tilde{w}} \times C_{f\tilde{w}} + n_d \times C_d \tag{3}$$

The actual values for device access costs, listed in Table 2, are obtained using device benchmarks on two SSDs (high-end and low-end) and a magnetic disk. According to our device benchmark, the high-end SSD has a very good random write performance: its average time of serving a random page write is only 25% slower than that of a sequential page write. In contrast, for the low-end SSD, the random writes are slower than the sequential writes by a factor of 50. The remarkable difference in the device performance characteristics can be explained by substantial differences in the proprietary FTL implementations.

4.1 AOR vs. AOW

We first compare AOR with AOW. Figure 6 shows their virtual execution times relative to the no-cache configuration. The replacement policy in both cases was LRU. The cache size $|L_f|$ was scaled by a factor of 4 from 1,000 to 16,000 pages. In this range, AOR suffers from the problem of duplicate caching. Hence, we can expect that AOW has a better performance. As indicated by the results, AOW clearly outperforms AOR, meaning that caching only dirty pages is quite efficient in our configuration, where the second-tier cache is of comparable size of the first-tier cache ($|L_r| = 10,000$, see also Equation 2). For this reason, this paper focus on AOW, and all algorithms evaluated in the remainder of the section use the AOW strategy.

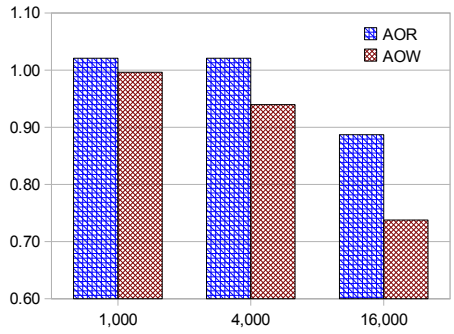


Fig. 6: Virtual execution times of AOR and AOW relative to the no-cache case

4.2 SCW vs. RCW

To study the cache-write strategies SCW and RCW, we used the performance metrics both of the high-end and the low-end SSD. The difference in device performance characteristics are reflected in our test results shown in Figure 7, where the virtual execution times of SCW and RCW with a cache of 4,000 pages relative to the no-cache configuration are shown. The replacement policy used in RCW was LRU. On the high-end SSD, RCW performance was superior because its higher hit ratio compensated the slightly higher cost of random flash writes. On the low-end SSD, however, RCW is much slower than SCW, because the latter only does sequential writes, which can be handled rather efficiently even by the low-end SSD.

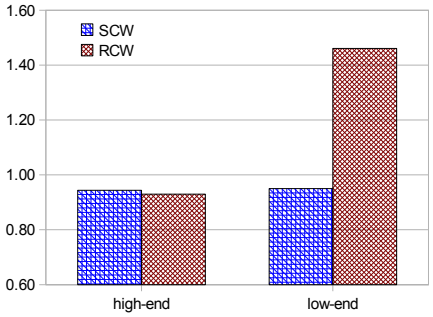


Fig. 7: Virtual execution times relative to the no-cache case

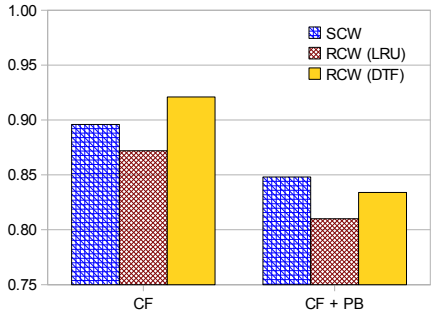


Fig. 8: Number of seeks relative to the non-track-aware LRU

4.3 CF and PB

We used the performance metrics of the high-end SSD to study the track-aware techniques discussed in Sect. 3.3. The goal of those techniques is to minimize the number of disk seeks. Figure 8 shows the numbers of seeks of various configurations relative to LRU (Admit-On-Write, without using track information) for

a cache size of 4,000 pages. The page-to-track mapping function t used in the simulation was $t(a) = a/32$.

The results reveal that the track-aware technique CF clearly reduced the number of seeks (up to 12% for LRU), and the combination CF+PB achieved even more significant improvements (up to 18% for LRU). Another observation is that the simple replacement policy LRU, combined with the track-aware techniques, achieved remarkably good performance.

5 Conclusion and Future Work

Based on our experimental results, we can conclude that:

- A small-sized (relative to the RAM buffer pool size) flash-based write cache can substantially improve storage system performance.
- Cache-writing strategies can significantly impact system performance, depending on the flash device implementation. For low-end flash devices with poor random write performance or raw flash memory, it is better to use SCW to handle the wear-leveling problem and random write problem natively in the database software.
- The page-to-track information, if available, can be used to further improve disk access performance.

The sector-to-track relation on modern disks can be much more sophisticate than the mapping function used in our simulation. The fact that most enterprise databases are hosted on RAID also adds complexity requiring further investigation. As future work, we plan to investigate the discussed algorithms and techniques in a larger variety of configurations and on real devices. We will also examine the correlation between the strategies and workload characteristics.

6 Acknowledgement

We are grateful to anonymous referees for valuable comments. This research is supported by the German Research Foundation and the Carl Zeiss Foundation.

References

1. E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.
2. T.S. Chung, D.J. Park, S. Park, D.H. Lee, S.W. Lee, and H.J. Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332–343, 2009.
3. L. Bouganim, B.T. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR'09*, 2009.
4. D. Tsirogiannis, S. Harizopoulos, et al. Query processing techniques for solid state drives. In *SIGMOD*, pages 59–72. ACM, 2009.
5. Y. Li, S.T. On, J. Xu, B. Choi, and H. Hu. DigestJoin: Exploiting fast random reads for flash-based joins. In *MDM'09*, pages 152–161. IEEE, 2009.

6. S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *Int. Conf. on Information Processing in Sensor Networks*, pages 410–419, 2007.
7. Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE'09*, pages 1303–1306. IEEE, 2009.
8. S. Yin, P. Pucheral, and X. Meng. A sequential indexing scheme for flash-based embedded systems. In *EDBT'09*, pages 588–599. ACM, 2009.
9. S. Park, D. Jung, et al. CFLRU: a replacement algorithm for flash memory. In *CASES*, pages 234–241, 2006.
10. Y. Ou, T. Härder, et al. CFDC: a flash-aware replacement policy for database buffer management. In *SIGMOD Workshop DaMoN*, pages 15–20, 2009.
11. P. Jin, Y. Ou, T. Härder, and Z. Li. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. *Data & Knowledge Eng.*, 72:83–102, 2012.
12. S.W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD'07*, pages 55–66. ACM, 2007.
13. S. Chen. FlashLogging: exploiting flash devices for synchronous logging performance. In *SIGMOD'09*, pages 73–86. ACM, 2009.
14. S. On, J. Xu, B. Choi, H. Hu, and B. He. Flag Commit: Supporting efficient transaction recovery on flash-based DBMSs. *IEEE Transactions on Knowledge and Data Engineering*, (99):1–1, 2011.
15. D. Narayanan, E. Thereska, et al. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys*, pages 145–158. ACM, 2009.
16. I. Koltsidas and S. D. Viglas. The case for flash-aware multi-level caching. Technical Report, 2009.
17. Y. Ou and T. Härder. Trading memory for performance and energy. In *DASFAA'11*, pages 241–253. Springer-Verlag, 2011.
18. J. Ousterhout, P. Agrawal, D. Erickson, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.
19. H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD'09*, pages 1–2. ACM, 2009.
20. Micron Technology, Inc. Micron collaborates with sun microsystems to extend lifespan of flash-based storage, achieves one million write cycles. `http://investors.micron.com/releasedetail.cfm?ReleaseID=440650`.
21. M. Canim, G.A. Mihaila, et al. SSD bufferpool extensions for database systems. In *VLDB*, pages 1435–1446, 2010.
22. J. Do, D.J. DeWitt, D. Zhang, J.F. Naughton, et al. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD'11*, pages 1113–1124. ACM, 2011.
23. Y. Li, J. Xu, B. Choi, and H. Hu. StableBuffer: optimizing write performance for dbms applications on flash devices. In *CIKM'10*, pages 339–348, New York, NY, USA, 2010. ACM.
24. G. Soundararajan, V. Prabhakaran, et al. Extending SSD lifetimes with disk-based write caches. In *USENIX FAST'10*. USENIX Association, 2010.
25. P. Yang, P. Jin, and L. Yue. Hybrid storage with disk based write cache. In *DASFAA'11*, pages 241–253. Springer-Verlag, 2011.
26. Y. Zhou, Z. Chen, et al. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.
27. C. Mohan, D. J. Haderle, et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.
28. T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 12 1983.