Data Management Over Flash Memory

Ioannis Koltsidas
IBM Research
Zurich, Switzerland
iko@zurich.ibm.com

Stratis D. Viglas
School of Informatics
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

Flash SSDs are quickly becoming mainstream and emerge as alternatives to magnetic disks. It is therefore imperative to incorporate them seamlessly into the enterprise. We present the salient results of research in the area, touching all aspects of the data management stack: from the fundamentals of flash technology, through storage for database systems and the manipulation of SSD-resident data, to query processing.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems; D.4.2 [**Storage Management**]: Secondary storage

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Flash memory has emerged as a high-performing and viable alternative to magnetic disks for data-intensive applications. In the near future, commodity hardware is expected to incorporate both flash SSDs and magnetic disks as storage media. In light of this development, fundamental principles of data storage and management need to be revisited, as all existing database systems and algorithms have been designed with disks consisting of rotating platters in mind. We present (a) what makes flash SSDs different from magnetic disks, (b) what challenges arise when flash technology is introduced in a database system context, (c) the recent results in this fresh research area, and (d) an outlook of existing problems and the things to come. The most important characteristics of flash SSDs can be summarized as follows:

- **I/O interface**. At the operating system level, flash SSDs behave like magnetic disks, as they are accessed through the same I/O interface. The on-disk controller maps system commands to flash memory operations.
- **No mechanical latency.** SSDs are purely electronic devices and have no mechanical moving parts. The time needed to access a block on an SSD is independent of its position on the physical medium.
- **I/O asymmetry**. Due to the electrical properties of flash chips, reading is faster than writing, even when writing to clean pages (typically twice as fast or more).
- **Erase-before-write limitation.** The most important constraint of SSDs is due to the erase-before-write limitation of flash pages: a sector cannot be overwritten. Rather, the whole flash block to which it belongs has to be erased first. This operation is orders of magnitude more expensive than both reads and writes.
- **Wear leveling.** The disk controller employs techniques to alleviate the effect of flash chip wearing on the lifetime of the device. Such techniques spread writes evenly across all flash chips in the device and across all flash blocks in the same chip, thereby prolonging the overall lifetime of the device.
- **Physical properties and energy efficiency.** Being purely electronic devices, SSDs have low power consumption and generate little heat when in use. They are, thus, ideal for mobile devices. Energy efficiency is also desirable in enterprise environments [17]. The lack of mechanical parts gives SSDs additional advantages: they incur faster start-up times, while they are resistant to extreme environmental conditions.

As evident from the list, flash SSDs require rethinking most aspects of established data management techniques.

2. FLASH-BASED DATA MANAGEMENT

To showcase the challenges in incorporating flash technology into database systems, we follow a bottom-up approach. First, we present the hardware and storage aspects of flash disks and how they are different from magnetic ones. We then focus on their performance aspects and show how they can be used to efficiently cater for database-specific storage needs. Moving up the processing stack, we deal with main-memory buffering and caching issues, before we finish with query execution over flash-enabled database systems.

2.1 The inner workings

Flash memory is a type of electronic memory that stores information in arrays of memory cells, called *flash cells*. Depending on the sensing technology, flash cells are divided into (a) *Single-Level Cells* (SLC), which can sense only the presence or absence of current, thereby storing only one bit of information, and (b) *Multi-Level Cells* (MLC), which can additionally detect the amount of current flow, typically utilizing four levels of voltage. As a result, MLC devices can store two bits of information, and are thus denser than SLC

SIGMOD'11, June 12–16, 2011, Athens, Greece.

ones. However, both reading and writing to an MLC takes longer than to an SLC, as there are more voltage levels to deal with and the cell can endure fewer write/erase cycles. Therefore, SLC devices are used for high performance and high-endurance purposes, with MLC ones catering for higher storage densities. *Solid State Drives*, or SSDs, also known as *flash disks*, package multiple flash memory chips into a single enclosure. Such devices also include a controller supported by several DRAM buffers. The controller translates system read and write requests to actual read, erase and write operations on the chips. The controller also employs algorithms and techniques to hide the complexities of flash memory from the user and improve I/O performance.

2.2 SSD performance evaluation

In [6], the authors study the performance characteristics of flash disks. They propose a benchmarking methodology for flash devices, provide an I/O benchmark that takes into account the particular characteristics of devices, and use it to evaluate a multitude of SSDs. A small number of performance indicators are found adequate to accurately capture the performance characteristics of SSDs. The impact of SSDs on algorithms is discussed in [3]. The authors found SSDs to yield a better random read but a much worse random write performance than magnetic disks. The algorithms designed for both main memory and magnetic disks were found to perform sub-optimally over flash memory, as they are not optimized for the medium. In accordance to [6], aligning write requests to block boundaries substantially improved random write performance; block alignment, however, did not help at random and sequential reads and random writes. Similarly, [37] proposes the *erase-once-write-many* model for analyzing algorithms over flash memory. In [24], the authors classify SSDs based on their architectural characteristics and performance, and identify which class better fits different uses. For instance, low-end SSDs are better used for logging [26]. Additionally, the authors present ways to improve the performance of each class for the workload the devices of the class are expected to process.

2.3 Accelerating random writes

The erase-before-write limitation makes random writes inefficient. To minimize this effect, flash controllers employ a software layer called the *Flash Translation Layer* (FTL). Its main purpose is to provide logical-to-physical address mapping, power-off recovery, and wear-leveling. Both the hardware and software components of SSDs affect the overall performance [1]. Different FTL algorithms are studied in [11]; with regard to their logical-to-physical address mapping they are categorized into three categories: sector mapping algorithms, block mapping algorithms, and hybrid mapping algorithms. A hybrid mapping scheme based on block-level associativity is presented in [20]. The potential drawbacks of the approach were lifted in [25] with the introduction of fully-associative logical to physical mappings. In [5], it is argued that increasing the amount of volatile RAM on flash disks is the only way to achieve acceptable random access write performance. Finally, in [30] the authors present a page-level FTL scheme based on lazy updates.

2.4 Write performance vs. wear leveling

To allow for low-latency write operations, an SSD requires some block to be clean upon the write request. The more clean blocks are available, the more write requests can be absorbed without waiting for an erasure. Therefore, the controller tries to continuously reclaim invalidated (or, obsolete) sectors. This procedure is referred to as *garbage collection* or *block reclamation*. The goals of write performance and wear leveling are often competing with garbage collection [15]. Assuming a block of relatively static data over time, *i.e.,* the block is not overwritten, it is best if the garbage collector does not touch that block. In the interest of wear leveling, on the other hand, by reclaiming the static block, its data can be moved to another erase unit that has been heavily erased in the past, thereby reducing future wear. Hence, garbage collection and wear leveling have contradicting goals. The additional write operations occurring due to garbage collection and wear leveling are referred to as *write amplification*. A probabilistic analysis of write amplification is presented in [18]. The results show that write amplification heavily affects both the write performance and the aging of the SSD.

2.5 Database storage

FTL algorithms designed for file systems are not well-suited for DBMSs: random write operations in file systems are mostly required for metadata [23]. DBMSs, however, perform random writes that are scattered over the whole address space. To improve write efficiency for databases, *in-page logging* (IPL) has been proposed [23]: changes made to a data page are not written directly to disk, but to log records for that page. Changes are logged on a per-page basis, while each data page and its log records are located in the same physical block of the disk *i.e.,* in the same erase unit. The authors also propose an IPL-based recovery mechanism for transactions that minimizes the cost of system recovery.

To avoid reading unnecessary attributes during scan selections and projections, the authors of [41] advocate storing relations using the PAX layout [2]. An operator termed *FlashScan* is introduced, which reads projected attributes of a selection and produces tuples in row format. Substantial performance improvement over a row store was observed, more so for queries of low selectivity and projectivity.

2.6 Hybrid systems

Hybrid systems employ both SSDs and HDDs for persistent storage. A system utilizing inexpensive SSDs along with HDDs is presented in [22]: data with read-intensive workloads are placed on the SSD, while data with write-intensive workloads are placed on the HDD. Similarly, the authors of [42] study adaptive load balancing via block allocations and migrations. They argue that the system is at an optimal state when it reaches a Wardrop equilibrium, *i.e.,* the response times of the storage devices are equal. An object placement advisor for a commercial DBMS over a hybrid system is described in [7]. The advisor decides which data should be placed on the SSD to maximize performance. The decision is reduced to a knapsack problem, which they attack using heuristics to achieve significant gains in performance. On the other hand, in [38] the authors use the HDD as a non-volatile write cache for the SSD: all writes are appended to a log on the HDD, so that all HDD writes are sequential. When the log is full, writes are merged back into the SSD. The same approach is independently followed in [16].

2.7 Main memory buffering

A lot of research has focused on buffer management over SSDs, *i.e.,* for when flash memory is used for persistent storage and a subset of the pages are cached in main memory.

In [19] the authors propose BPLRU, a scheme for the on-disk buffer cache of flash disks. This scheme treats the buffer as a write cache and groups RAM buffers in blocks that are equal in size to the flash erase-unit; page replacement is performed with erase-unit granularity (using LRU). Similarly, the authors of [36] propose that the buffer cache choose for replacement a clean page over a dirty one and therefore trade the number of writes with the number of reads. In [21, 22], this concept was generalized for a system in which the same buffer pool holds pages from both the SSD and the HDD. In that case, not only the dirtiness of the page, but also the read/write costs of the storage media and the access history of the page are considered when choosing a page to replace. In [35] the authors extend the ideas of [36]: they not only aim to minimize the number of writes to the SSD, but they also aim to exploit the spatial locality of victim pages. Similarly, [39] presents a technique for transforming random writes of dirty pages into sequential ones. This is achieved by creating a shim layer within the storage manager of a DBMS that writes dirty pages, evicted by the buffer manager, sequentially in multiples of the erase block size. Hence, the physical medium is utilised in an append-only manner.

2.8 Caching in flash memory

In [32], the authors discuss how SSDs can be incorporated in the enterprise storage hierarchy in terms of performance, capacity, power consumption and reliability. They describe an automated tool that decides in an offline fashion the optimal storage hardware configuration for a specific workload, using multiple metrics. Another piece of work relevant to caching in flash memory —which, however, is not geared towards database workloads— is outlined in [27] and implemented in the ZFS filesystem [40]. The SSD is used as a cache for the magnetic disk with the goal of improving the performance of random read workloads. Data flow schemes and metadata management alternatives for such a 3-tier system are studied in [21], where it is found that optimal design decisions vary widely across workloads, SSDs and cache sizes.

The authors of [9] present a flash-resident bufferpool extension for a commercial DBMS. The secondary bufferpool monitors the temperature of pages using heat maps: main memory evictees are admitted to the flash bufferpool only if they are found hot enough. This way, the flash cache is not polluted with cold data. A similar bufferpool design is presented in [16] and implemented inside an open-source DBMS: the SSD is used as a cyclic buffer, so that the write pattern to the bufferpool is more flash-friendly. Both approaches demonstrate promising experimental results.

2.9 Indexing

In [33], the authors focus on SSD-friendly access method design. In addition to avoiding in-place updates and random writes, the designers should avoid sub-block deletions and employ *semi-random* writes as well. In [44], the authors propose storing B^+-tree nodes as a sequence of log records spread over multiple disk blocks. Similarly, the authors of [29] propose the FD-tree, a flash-friendly B^+-tree variant hierarchical index structure. On the same topic, [34] proposes a self-tuning B^+-tree that provides indexing functionality to the storage manager of a flash-based DBMS. An efficient implementation of R-trees over the FTL of an SSD is presented in [43], aiming not only to improve performance but also reduce energy consumption. The implications of solid state storage on spatial index structures is studied in [14]. In [28], the authors propose a hash index for flash-resident data that avoids deleting records in place. Another hash index, termed *MicroHash*, is proposed in [45]. In the same context, the authors of [33] study the problem of efficiently maintaining a large random sample of a data stream. A scalable transactional record manager using raw flash memory is presented in [4]: a distributed log-structured multiversioned and unpartitioned database is stored on flash and shared over the network by many database servers. A high-throughput persistent key-value store is also presented in [12]; in this case, however, the SSD is used as a cache between DRAM and backend HDDs.

2.10 Query execution

The use of indexes during query execution is discussed in [31], where the author studies the impact of predicate selectivity and join algorithms over a hybrid system with both an SSD and an HDD. Query execution over flash memory is also studied in [41]. The authors propose a join operator, termed *FlashJoin*, that aims (a) to reduce the I/O cost of join evaluation by minimizing the number of passes over the participating tables and (b) to minimize the I/O required to fetch attributes for the query result. The most common *ad hoc* join algorithms are revisited in [13] with respect to processing over flash memory. The authors show which previous results for HDDs continue to hold for flash drives and what modifications in thinking about query execution are necessary for SSDs. In [8] the authors focus on membership queries and study how Bloom filters can be adapted for SSDs. They break a large Bloom filter into multiple page-sized subfilters, thereby improving the locality of bit operations when the filter is stored on flash. At the same time, they amortize the I/O cost by buffering read and write requests to the same sub-filter and executing them in batches.

The authors of [26] experimentally evaluated the overhead of secondary operations during query execution (*e.g.,* logging, temporary storage) and found it to be substantial. They suggest that to improve the performance of transaction processing systems, one should not optimize only for primary structures and operations, but for the secondary ones too. Transactional logging using flash memory is also studied in [10]. Instead of an internal SSD, the author proposes a solution that utilizes multiple USB flash drives, since they have comparable performance at a much lower price.

3. OUTLOOK

A great deal of research remains to be done on solid state drives, especially in the context of database systems. With the price, performance, and capacity characteristics of SSDs constantly changing, the term *solid state drive* incorporates multiple classes of device. The only major common characteristic of all these devices is the excellent random read performance. The remaining characteristics range within more than two orders of magnitude across different devices. Some SSDs are more than an order of magnitude slower than HDDs at random writes, while other SSDs dominate HDDs in both random read and write throughput and latency.

The most important question to answer is what is the best use of each class of device in a DBMS. The answer also depends on the amount of DRAM available and the presence, number, size, speed, and configuration of the underlying HDDs. Alternatives include (a) using the SSD as persistent storage, either in combination with HDDs or by itself,

(b) using the SSD as a cache for the HDDs, (c) using the SSD as a transactional log, (d) using the HDD as a log-structured write cache for the SSD, (e) using the SSD as a temporary buffer for specific query evaluation algorithms (*e.g.,* sorting), and, of course, (f) any combination of the above.

Lately, SSD manufacturers offer a richer interface to the users, most notably by allowing explicit invalidation of logical pages, using the TRIM command. Also, in recent products, the controller logic and the FTL algorithm for the disk run at the operating system level, *i.e.,* using the host CPU and DRAM. Giving users more power and moving critical operations outside of the SSD enclosure is particularly interesting for the database community, as currently all flash devices are geared towards user filesystems.

4. REFERENCES

[1] N. Agrawal *et al.* Design tradeoffs for SSD performance. In *USENIX Annual Technical*, 2008.

[2] A. Ailamaki *et al.* Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3), 2002.

[3] D. Ajwani *et al.* Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *WEA 2008*.

[4] P. A. Bernstein *et al.* Hyder - a transactional record manager for shared flash. 2011.

[5] A. Birrell *et al.* A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.

[6] L. Bouganim *et al.* uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[7] M. Canim *et al.* An object placement advisor for DB2 using solid state storage. *Proc. VLDB Endow.*, 2(2), 2009.

[8] M. Canim *et al.* Buffered Bloom filters on solid state storage. In *ADMS*, 2010.

[9] M. Canim *et al.* SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3(2), 2010.

[10] S. Chen. FlashLogging: exploiting flash devices for synchronous logging performance. In *SIGMOD 2009*.

[11] T.-S. Chung *et al.* System software for flash memory: A survey. In *EUC*, 2006.

[12] B. Debnath *et al.* Flashstore: High throughput persistent keyvalue store. *Proc. VLDB Endow.*, 3(2), 2010.

[13] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In *DAMON 2009*.

[14] T. Emrich *et al.* On the impact of flash SSDs on spatial indexing. In *DAMON*, 2010.

[15] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2), 2005.

[16] A. Holloway. Adapting database storage for new hardware. PhD Thesis. University of Wisconsin - Madison., 2009.

[17] HP Labs. Getting smart about data center cooling. `http://www.hpl.hp.com/news/2006/power.html`, 2006.

[18] X.-Y. Hu *et al.* Write amplification analysis in flash-based solid state drives. In *SYSTOR 2009*.

[19] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.

[20] J. Kim *et al.* A space-efficient flash translation layer for CompactFlash systems. *Trans. on Consumer Electronics.*, 2002.

[21] I. Koltsidas. Flashing up the storage hierarchy. PhD Thesis. The University of Edinburgh., 2010.

[22] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1), 2008.

[23] S.-W. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In *SIGMOD*, 2007.

[24] S.-W. Lee *et al.* Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD 2009*.

[25] S.-W. Lee *et al.* A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Comp. Sys.*, 2007.

[26] S.-W. Lee *et al.* A case for flash memory SSD in enterprise database applications. In *SIGMOD*, 2008.

[27] A. Leventhal. Flash storage memory. *Commun. ACM*, 51(7):47–51, 2008.

[28] X. Li *et al.* A new dynamic hash index for flash-based storage. In *WAIM*, 2008.

[29] Y. Li *et al.* Tree indexing on flash disks. In *ICDE*, 2009.

[30] D. Ma *et al.* LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *SIGMOD*, 2011.

[31] D. Myers. On the use of NAND flash memory in high-performance relational databases. MSc Thesis, MIT, 2007.

[32] D. Narayanan *et al.* Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys*, 2009.

[33] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.*, 1(1), 2008.

[34] S. Nath and A. Kansal. FlashDB: Dynamic Self-Tuning Database for NAND Flash. In *IPSN*, 2007.

[35] Y. Ou *et al.* CFDC: a flash-aware replacement policy for database buffer management. In *DAMON*, 2009.

[36] S.-Y. Park *et al.* CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.

[37] K. A. Ross. Modeling the performance of algorithms on flash memory devices. In *DAMON*, 2008.

[38] G. Soundararajan *et al.* Extending SSD lifetimes with disk-based write caches. In *FAST*, 2010.

[39] R. Stoica *et al.* Evaluating and repairing write performance on flash devices. In *DAMON*, 2009.

[40] Sun Microsystems. The Solaris ZFS filesystem, 2008.

[41] D. Tsirogiannis *et al.* Query processing techniques for solid state drives. In *SIGMOD*, 2009.

[42] X. Wu and A. N. Reddy. Exploiting concurrency to improve latency and throughput in a hybrid storage system. *MASCOTS 2010*.

[43] C.-H. Wu *et al.* An efficient R-tree implementation over flash-memory storage systems. In *GIS*, 2003.

[44] C.-H. Wu *et al.* An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems. *Trans. on Embedded Computing Sys.*, 6(3), 2007.

[45] D. Zeinalipour-Yazti *et al.* Microhash: an efficient index structure for flash-based sensor devices. In *FAST*, 2005.