Flash in a DBMS: Where and How?

Manos Athanassoulis[†] Anastasia Ailamaki[†] Shimin Chen[⋆] Phillip B. Gibbons[⋆] Radu Stoica[†]
[†]Ecole Polytechnique Fédérale de Lausanne [⋆]Intel Labs Pittsburgh

Abstract

Over the past decade, new solid state storage technologies, with flash being the most mature one, have become increasingly popular. Such technologies store data durably, and can alleviate many handicaps of hard disk drives (HDDs). Nonetheless, they have very different characteristics compared to HDDs, making it challenging to integrate such technologies into data intensive systems, such as database management systems (DBMS), that rely heavily on underlying storage behaviors. In this paper, we ask the question: Where and how will flash be exploited in a DBMS? We describe techniques for making effective use of flash in three contexts: (i) as a log device for transaction processing on memory-resident data, (ii) as the main data store for transaction processing, and (iii) as an update cache for HDD-resident data warehouses.

1 Introduction

For the past 40 years, hard disk drives (HDDs) have been the building blocks of storage systems. The mechanics of HDDs rotating platters dictate their performance limitations: latencies dominated by mechanical delays (seeks and rotational latencies), throughputs for random accesses much lower than sequential accesses, interference between multiple concurrent workloads further degrading performance [25], etc. Moreover, while CPU performance and DRAM memory bandwidth have increased exponentially for decades, and larger and deeper cache hierarchies have been increasingly successful in hiding main memory latencies, HDD performance falls further and further behind. As illustrated in Table 1, HDDs' random access latency and bandwidth have improved by only 3.5X since 1980, their sequential bandwidth lags far behind their capacity growth, and the ratio of sequential to random access throughput has increased 19 fold.

New storage technologies offer the promise of overcoming the performance limitations of HDDs. Flash, phase change memory (PCM) and memristor are three such technologies [7], with flash being the most mature. Flash memory is becoming the de facto storage medium for increasingly more applications. It started as a storage solution for small consumer devices two decades ago and has evolved into high-end storage for performance sensitive enterprise applications [20, 21]. The absence of mechanical parts implies flash is not limited by any seek or rotational delays, does not suffer mechanical failure, and consumes less power than HDDs. As highlighted in Table 1, flash-based solid state drives (SSDs) fill in the latency and bandwidth gap left by HDDs. SSDs also provide a low ratio between sequential and random access throughput[1]. The time to scan the entire device sequentially is much lower than modern HDDs, close to what it was in older HDDs.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]Random access throughput for SSDs is computed using a mix of reads and writes. If only reads are performed then sequential and random access throughputs are roughly equal.

Table 1: Comparison of Hard Disk Drives (1980, 2010) and Flash Drives (2010)

Device & Year	Capacity (GB)	Cost ($)	Cost/MB ($)	Random Access Latency (ms)	Random Access Bandwidth (MB/s)	Sequential Access Bandwidth (MB/s)	Sequential BW / Random BW	Device Scan (s)
HDD 1980	0.1	20000	200	28	0.28	1.2	4.3	83
HDD 2010	1000	300	0.0003	8	0.98	80	81.6	12500
SSD 2010	100	2000	0.02	0.026	300	700	2.3	143

Sources: Online documents and presentations [13], vendor websites and other sources [18].

Flash devices, however, come with certain limitations of their own. There is an asymmetry between random reads and random writes [3, 6, 23], with the latter incurring a performance hit due to the specifics of the flash technology [2, 6]. Newer SSDs [16, 19] mitigate this performance hit, but random writes still have a negative impact on the future performance of the device. Interestingly, sequential writes not only offer good performance but in many cases repair device performance after extensive random writes [26]. Finally, flash cells wear out after 10K–100K writes to the cell.

The differences between HDDs and SSDs are particularly important in Database Management Systems (DBMS) because their components (query processing, query optimization, query evaluation) have been tuned for decades with the HDD characteristics in mind. Specifically, random accesses are considered slow, sequential accesses are preferred, and capacity is cheap. Thus, achieving successful integration of flash requires revisiting DBMS design. Currently, flash usage in DBMS follows two trends, resulting either in flash-only systems or in hybrid flash-HDD systems. Flash-only systems benefit greatly from flash-friendly join algorithms [29], indexes [1, 8, 23, 24], and data layout (this paper). Hybrid systems use flash judiciously to cache either hot or incoming data or for specific operations [9, 14].

In this paper we describe techniques for using flash to optimize data management in three different settings, covering DRAM-resident, SSD-resident, and HDD-resident data stores. Each technique represents an example of how flash can be used within current systems to address the limitations of HDDs. First, we consider transaction processing on a memory-resident data store, and show how to use flash for transactional logging, dramatically improving transaction throughput (Section 2.1). Second, a straightforward way to use flash is to replace all the HDDs with SSDs without changing the DBMS software. In this scenario flash addresses the performance limitations of HDDs but poses new challenges because excessive random writes degrade performance and wear out the flash storage prematurely. To overcome these challenges, we present a technique that leaves a DBMS unchanged and yet avoids random writes on flash (Section 2.2). Third, we show how flash can be used as a performance booster for data warehouses stored primarily on HDDs. Specifically, we describe techniques for buffering data updates on flash such that (i) for performance, queries process the HDD-resident data without interference from concurrent updates and (ii) for correctness, the query results are adjusted on-the-fly to take into account the flash-resident updates (Section 3). Finally, we conclude by highlighting techniques and open problems for other promising uses of flash in data management (Section 4).

2 Efficient Transaction Processing Using Flash Devices

In this section we describe how flash can be used effectively in the context of online transaction processing (OLTP). We show that the absence of mechanical parts makes flash an efficient logging device and that using SSDs as a drop-in replacement for HDDs greatly benefits from flash-friendly data layout.

2.1 Using Flash for Transactional Logging

Synchronous transactional logging, in which log records are forced to stable media before a transaction commits, is the central mechanism for ensuring data persistency and recoverability in database systems. As DRAM capacity doubles every two years, an OLTP database that was considered "large" ten years ago can now fit into main memory. For example, a database running the TPCC benchmark with 30 million users requires less than 100GB space, which can easily fit into the memory of a server (64–128GB of memory). In contrast,

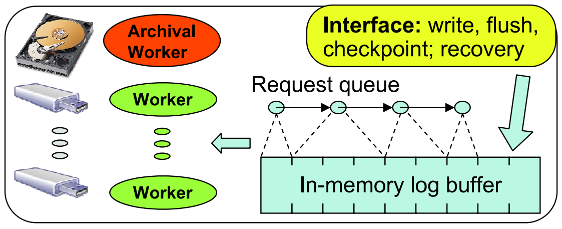


Figure 1: FlashLogging architecture: exploiting an array of flash devices and an archival HDD for faster logging and recovery.

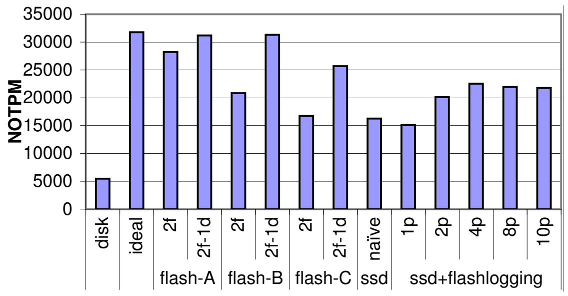


Figure 2: FlashLogging TPCC performance.

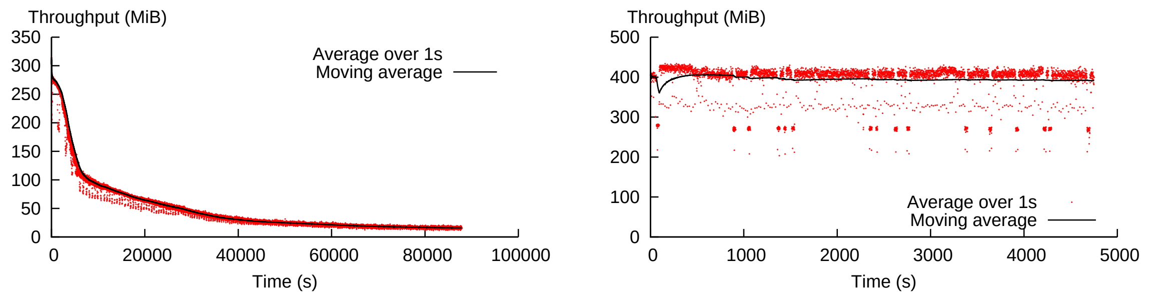
synchronous logging requires writing to stable media, and therefore is becoming increasingly important to OLTP performance. By instrumenting a MySQL-InnoDB running TPCC, we find that synchronous logging generates small sequential I/O writes. Such write patterns are ill-suited for an HDD because its platters continue to rotate between writes and hence each write incurs nearly a full rotational delay to append the next log entry. Because flash supports small sequential writes well, we propose FlashLogging [10] to exploit flash for synchronous logging.

FlashLogging Design. Figure 1 illustrates the FlashLogging design. First, we exploit multiple flash devices for good logging and recovery performance. We find that the conventional striping organization in disk arrays results in sub-optimal behavior (such as request splitting or skipping) for synchronous logging. Because our goal is to optimize sequential writes during normal operations and sequential reads during recovery, we instead propose an *unconventional* array organization that only enforces that the LSNs (log sequence numbers) on an individual device are non-decreasing. This gives the maximal flexibility for log request scheduling. Second, we observe that write latencies suffer from high variance due to management operations (such as erasures) in flash devices. We detect such outlier writes and re-issue them to other ready flash devices, thus hiding the long latencies of outliers. Third, the flash array can be implemented either with multiple low-end USB flash drives, or as multiple partitions on a single high-end SSD. Finally, our solution can exploit an HDD as a near-zero-delay archival disk. During normal processing, flash-resident log data is flushed to the HDD once it reaches a predefined size (e.g., 32KB). Logging performance is improved because the HDD can also serve write requests when all the flash drives are busy.

Performance Evaluation. We replace the logging subsystem in MySQL/InnoDB with FlashLogging. Figure 2 reports TPCC throughput in new order transactions per minute (NOTPM), comparing 14 configurations. “Disk” represents logging on a 10k rpm HDD, while “ideal” enables the write cache in “disk” (violating correctness) so that small synchronous writes achieve almost ideal latency. We evaluate three low-end USB flash drives (A, B, and C) from different vendors; “2f” uses two identical flash drives, and “2f-1d” is “2f” plus an archival HDD. We also employ a single high-end SSD either directly with the original logging system (“naive”), or with FlashLogging while using multiple SSD partitions (1–10 partitions) as multiple virtual flash drives. From Figure 2, we see that (i) FlashLogging achieves up to 5.7X improvements over traditional (HDD-based) logging, and obtains up to 98.6% of the ideal performance; (ii) the optional archival HDD brings significant benefits; (iii) while replacing the HDD with an SSD immediately improves TPCC throughput by 3X, FlashLogging further exploits the SSD’s inner parallelism to achieve an additional 1.4X improvement; and (iv) when compared to the high-end SSD, multiple low-end USB flash drives achieve comparable or better performance at much lower price.

2.2 Transparent Flash-friendly Data Layout

Using flash as persistent storage medium for a DBMS often leads to excessive random writes, which impact flash performance and its predictability. In Figure 3(a) we quantify the impact of continuous 4K random writes



(a) Random writes (b) Random read/write mix (50%-50%) using A/P

Figure 3: Experiments with a FusionIO SSD as (a) drop-in replacement of HDDs and (b) using Append/Pack.

on FusionIO [16], a high-end SSD. We find that the sustained throughout is about 16% of the advertised random throughput. At the same time, sequential writes can fix random read performance [26].

Transforming Temporal Locality To Spatial Locality. Viewing SSDs as append logs helps us exploit the good sequential write bandwidth when writing. Consequently, temporal locality is transformed to spatial locality and thus sequential reads are transformed to random reads. This transformation helps overall performance, because unlike HDDs, flash offers virtually the same random and sequential read performance.

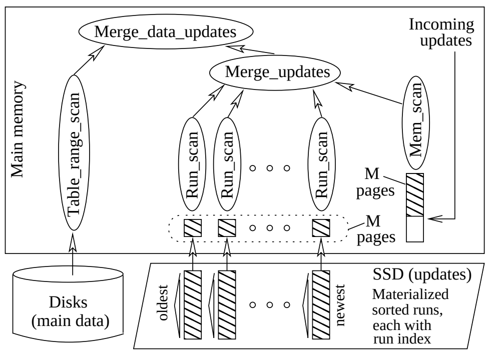
Append/Pack (A/P) Design. The A/P data layout [26] treats the entire flash device as a log. The dirty pages that come out of the buffer pool to be stored persistently are appended in the flash device and an index is kept in memory maintaining the mapping between the database page id and the location in the log. The A/P layer exports to the overlying device a transparent block device abstraction. We implement a log-structure data layout algorithm that buffers the dirty pages and writes them in blocks equal to the erase block size in order to optimize write performance. A/P organizes, as well, a second log-structured region of the device that is used for the packing part of the algorithm. During the operation of the system, some logical pages are appended more than once (i.e., the pages are updated, having multiple versions in the log). In order to keep track of the most recent version of the data, we invalidate the old entry of the page in the in-memory data structure. When the log is close to full with appended pages, we begin *packing*, that is, we move valid pages from the beginning of the log to the second log-structure assuming these pages are not updated for a long period (*cold pages*). The premise is that any (write-)hot page should be invalidated and all remaining pages are assumed to be cold. The cold pages are appended in a second log-structure to ensure good sequential performance and, thus, fast packing of the device.

Experimentation and Evaluation. We implement A/P as a standalone dynamic linked library which can be used by any system. The A/P library offers the typical *pwrite* and *pread* functions but manipulates writes as described above. We experimented with a PCIe Fusion ioDrive card [16]. The flash card offers 160GB capacity, and can sustain up to 700MB/s read bandwidth and up to 350MB/s sequential write bandwidth. Serving a TPCC-like workload using the A/P design, we maintain stable performance (in a read/write mix) achieving the max that the device could offer (400MB/s throughput as a result of combining reads and writes), as shown in Figure 3(b). When compared to the performance of a TPCC system on the same device but without A/P, we achieved speedups up to 9X [26].

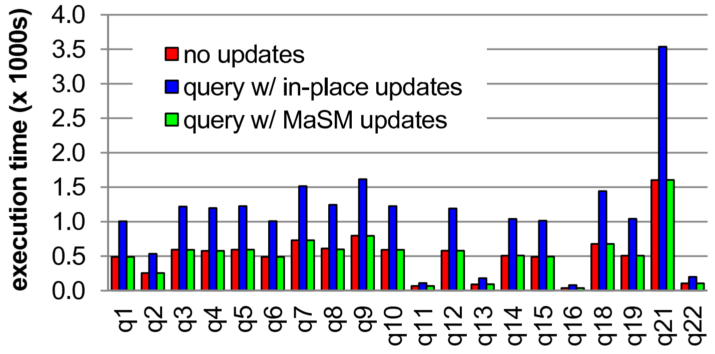
3 Flash-enabled Online Updates in Data Warehouses

In this section, we investigate the use of flash storage as a performance booster for data warehouses (DW) stored primarily on HDDs, because for the foreseeable future, HDDs will remain much cheaper but slower than SSDs. We focus on how to minimize the interference between updates and queries in DWs.

While traditional DWs allow only offline updates at night, the need for 24x7 operations in global markets and the data freshness requirement of online and other quickly-reacting businesses make concurrent online updates



(a) MaSM algorithm using 2M memory



(b) TPCH performance with online updates

Figure 4: MaSM design and performance. ($M = \sqrt{\|SSD\|}$)

increasingly desirable. Unfortunately, the conventional approach of performing updates in place can greatly disturb the disk-friendly access patterns (mainly, sequential scans) of large analysis queries, thereby slowing down TPCH queries by 1.5–4.0X in a row-store DW and by 1.2–4.0X in a column-store DW [4]. Recent work studied *differential updates* for addressing this challenge in column store DWs [17, 27]. The basic idea is (i) to cache incoming updates in an in-memory buffer, (ii) to take the cached updates into account on-the-fly during query processing, so that queries see fresh data, and (iii) to migrate the cached updates to the main data when the buffer is full. While significantly improving performance, these proposals require large in-memory buffers in order to avoid frequent, expensive update migrations. But dedicating a large amount of memory solely to buffering updates significantly degrades query operator performance, because less memory is available for caching frequently accessed structures (e.g., indices) and intermediate data (e.g., in sorting, hash-joins).

Exploiting Flash to Cache Updates. Our proposal follows the differential updates idea discussed above, but instead of using a large in-memory buffer, exploits SSDs (and a small amount of memory) to cache incoming updates. The goal is to minimize online updates' performance impact on table range scan operations, which are the major query pattern in large analytical DWs [5]. Unfortunately, naively applying the prior differential update techniques [17, 27] in the flash-augmented setting slows down table range scans by up to 3.8X [4]. Instead, we observe that the merging operation is similar to an outer join between the main data residing on HDDs and the updates cached on SSDs. To facilitate the merging, we sort the cached updates in the same order as the main data. The memory footprint is reduced by using two-pass external sorting: generating sorted runs then merging them. This requires $M = \sqrt{\|SSD\|}$ pages of memory to sort $\|SSD\|$ pages of cached updates. Moreover, because a later query should see all the updates that an earlier query has seen, we can materialize sorted runs and reuse them across multiple queries. We call the resulting algorithm *materialized sort-merge (MaSM)*.

Figure 4(a) illustrates a MaSM algorithm using $2M$ memory. It uses an M-page in-memory buffer to cache incoming updates. When the buffer fills, updates are sorted to generate a materialized sorted run of size M on the SSD. A table range scan will merge up to M materialized sorted runs and the in-memory updates. When the SSD is close to full, it is likely that there exist cached updates for every data page. Therefore, MaSM migrates updates to the main HDD-resident data store by sequentially writing back the merging outcome of a full table scan. Moreover, by attaching the commit timestamp to every update record, MaSM can correctly support concurrent queries, migration, and incoming updates, as well as transactions based on snapshot isolation or two-phase locking [4]. Furthermore, we also designed a more sophisticated MaSM algorithm that reduces the memory footprint to M pages by incurring extra SSD I/Os [4].

MaSM Evaluation. We implement MaSM in a prototype row-store DW. We compare MaSM and prior approaches using both synthetic data and TPCH [4]. We report the latter here. Figure 4(b) shows the TPCH performance with online updates. We record the disk traces when running TPCH queries on a 30GB database in a commercial row store. (Query 17 and 20 did not finish in 24 hours.) Then we replay the disk accesses as the query workload on our prototype. For MaSM, we use 1GB flash space, 8MB memory, and 64KB sized SSD I/Os. In Figure 4(b), there are three bars for every query: query time without updates (left), with in-place updates (middle), and with updates using MaSM (right). We see that in-place updates incur 1.6–2.2X slowdowns. In contrast, the MaSM algorithm achieves almost the same performance (within 1%) as the queries without updates, providing fresh data with negligible overhead.

4 Conclusions and Open Problems

Increasingly popular in mainstream computing, flash devices present new performance vs. price tradeoffs for data management systems. In this paper, we examined three aspects of traditional relational database management systems (RDBMS) for answering the question: Where and how can flash be exploited in a DBMS? In particular, we studied the use of flash in transactional logging in memory-resident OLTP systems, flash-friendly data layout in flash-resident OLTP systems, and flash as an update cache in HDD-resident data warehousing systems. Experimental results showed that our proposed techniques make effective uses of flash in a DBMS.

There are a number of other opportunities for exploiting flash in data management, including:

- *Flash-Only Data Warehouses.* Given its high performance, low energy consumption, and decreasing price, flash-based SSDs have been considered as the main storage for DWs [28]. In such settings, the fast random accesses of flash may significantly benefit database data structures and query processing algorithms (e.g., joins [29]). On the other hand, because of flash's poor random write performance, our solution for online updates may still be desirable for converting random in-place updates into sequential writes.
- *Exploiting Flash Beyond Traditional RDBMS.* More generally, it is interesting to study the use of flash for improving data management solutions beyond traditional RDBMS, such as data stream management, data management in the cloud, key-value stores [14], sensor networks [23], approximate query processing, and so on. For example, Chen *et al.* [11] proposed a non-blocking join algorithm, PR-Join, that achieves nearly optimal performance by exploiting SSDs as temporary storage for spilling intermediate data. PR-Join is shown to support efficiently both online aggregation and stream processing.
- *Alternative Memory Hierarchies with Flash.* Given the very different characteristics of flash compared to both DRAM and HDDs, it can be beneficial to use flash in alternative memory/storage hierarchy organizations. For example, Mesnier *et al.* [22] proposed differentiated storage services that intelligently cache high-priority I/O blocks on SSDs for better QoS. Canim *et al.* [9] showed the benefits to traditional RDBMS of including flash as a caching layer between main memory and HDDs.
- *Emerging Byte-Addressable Non-Volatile Memory Technologies.* Several emerging non-volatile memory technologies are byte-addressable with access latencies comparable to DRAM, and endurance much better than flash [7]. Among them, phase change memory (PCM) is the most promising to be ready for commercial use in the near future. Because the performance of PCM falls inbetween flash and DRAM, there have been proposals to use PCM both in SSDs and as main memory (replacing DRAM) [15]. PCM-based SSDs provide a similar block-level interface and better read/write performance than flash-based SSDs. The main characteristic difference is that PCM does not require erases before writing and therefore its random write performance is close to its sequential write performance. PCM-based main memory promises to bring more profound changes to computer systems because of the fine-grained non-volatility and the unique read/write characteristics of PCM (e.g., writes are much slower and power-hungry than reads). Thus, it is important to investigate the impact of such changes on data management systems [12].

These opportunities make for interesting future work.

References

[1] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *VLDB*, 2009.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. *USENIX*, 2008.

[3] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design. *WEA*, 2008.

[4] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Towards Efficient Concurrent Updates and Queries in Data Warehousing. *Technical Report, EPFL-REPORT-152941*, 2010.

[5] J. Becla and K.-T. Lim. Report from the First Workshop on Extremely Large Databases (XLDB 2007). In *Data Science Journal*, 2008.

[6] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. *CIDR*, 2009.

[7] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of Candidate Device Technologies for Storage-Class Memory. *IBM J. of R&D*, 2008.

[8] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered Bloom Filters on Solid State Storage. *ADMS*, 2010.

[9] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 2010.

[10] S. Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. *SIGMOD*, 2009.

[11] S. Chen, P. B. Gibbons, and S. Nath. PR-Join: A Non-blocking Join Achieving Higher Early Result Rate with Statistical Guarantees. *SIGMOD*, 2010.

[12] S. Chen, P. B. Gibbons, and S. Nath. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*, 2011.

[13] M. Dahlin. Technology Trends. http://www.cs.utexas.edu/~dahlin/techTrends.

[14] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. *VLDB*, 2010.

[15] E. Doller. Phase Change Memory and its Impacts on Memory Hierarchy. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[16] FushionIO. The FusionIO Drive Data Sheet. http://community.fusionio.com/media/p/459.aspx.

[17] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. A. Boncz. Positional Update Handling in Column Stores. *SIGMOD*, 2010.

[18] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2007.

[19] Intel. X-25E Technical Documents. http://www.intel.com/design/flash/nand/extreme/technicaldocuments.htm.

[20] S.-W. Lee, B. Moon, and C. Park. Advances in Flash Memory SSD Technology for Enterprise Database Applications. *SIGMOD*, 2009.

[21] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. *SIGMOD*, 2008.

[22] M. Mesnier, S. Hahn, and B. McKean. Making the Most of Your SSD: A Case for Differentiated Storage Services. *FAST WIPs*, 2009.

[23] S. Nath and P. B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. *VLDB Journal*, 2010.

[24] S. T. On, H. Hu, Y. Li, and J. Xu. Lazy-Update B+-Tree for Flash Devices. *Mobile Data Management*, 2009.

[25] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: Robust Database Storage Management Based on Device-specific Performance Characteristics. *VLDB*, 2003.

[26] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. *DaMoN*, 2009.

[27] M. Stonebraker et al. C-Store: A Column-Oriented DBMS. *VLDB*, 2005.

[28] Teradata. Teradata Extreme Performance Appliance: The World's First Solid State Data Warehouse Appliance for Hyper-analytics. http://www.teradata.com/t/newsrelease.aspx?id=12282, October 2009.

[29] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. *SIGMOD*, 2009.