Designing Database Operators for Flash-enabled Memory Hierarchies

Goetz Graefe[‡] Stavros Harizopoulos[‡] Harumi Kuno[‡]
Mehul A. Shah[‡] Dimitris Tsirogiannis[§] Janet L. Wiener

[‡]Hewlett-Packard Laboratories, Palo Alto, CA, USA
{firstname.lastname}@hp.com

[§]Microsoft, Madison, WI, USA
dimitsir@microsoft.com

Abstract

Flash memory affects not only storage options but also query processing. In this paper, we analyze the use of flash memory for database query processing, including algorithms that combine flash memory and traditional disk drives. We first focus on flash-resident databases and present data structures and algorithms that leverage the fast random reads of flash to speed up selection, projection, and join operations. FlashScan and FlashJoin are two such algorithms that leverage a column-based layout to significantly reduce memory and I/O requirements. Experiments with Postgres and an enterprise SSD drive show improved query runtimes by up to 6x for queries ranging from simple relational scans and joins to full TPC-H queries. In the second part of the paper, we use external merge sort as a prototypical query execution algorithm to demonstrate that the most advantageous external sort algorithms combine flash memory and traditional disk, exploiting the fast access latency of flash memory as well as the fast transfer bandwidth and inexpensive capacity of traditional disks. Looking forward, database query processing in a three-level memory hierarchy of RAM, flash memory, and traditional disk can be generalized to any number of levels that future hardware may feature.

1 Introduction

As flash memory improves in price, capacity, reliability, durability, and performance, server installations increasingly include flash-based solid state drives (SSDs). However, the transition to a memory hierarchy that includes SSDs requires us to reexamine database design decisions. SSDs perform random reads more than 100x faster than traditional magnetic hard disks, while offering comparable (often 2-3x higher) sequential read and write bandwidth. Database workloads may not benefit immediately from SSDs, however, because databases have traditionally been designed so as to avoid random I/O. Instead, traditional data structures, algorithms and tuning parameters favor sequential accesses, which are orders of magnitude faster than random acccesses on hard disk drives (HDDs).

The present paper analyzes the use of flash memory for database query processing, including algorithms that combine flash memory and traditional disk drives. We review relevant prior work in Section 2. We consider how to leverage flash for selection, projection, and join operations in Section 3, and propose new techniques for sorting in a multi-level memory hierarchy in Section 4. Finally, we offer conclusions in Section 5.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Prior Work

Work addressing SSDs has thus far focused on quantifying instant benefits from the fast random reads of SSDs [11, 15] and addressing their slow random writes [10, 12, 16, 19]. Our focus is unique in that we investigate query processing techniques that use flash to improve the performance of complex data analysis queries, which are typical in business intelligence (BI) and data warehousing workloads.

Several recent studies have measured the read and write performance of flash SSDs [2, 15, 18]. uFLIP [2] defines a benchmark for measuring sequential and random read and write performance and presents results for 11 different devices. Of particular note, they identify a "startup phase" where random writes may be cheaper on a clean SSD (since no blocks need to be erased) but quickly degrade as the disk fills. Polte et al. perform a similar study using less sophisticated benchmarks, but focus filesystems running on SSDs. They show the degraded mode is 3-7X worse than the "startup" phase but still an order of magnitude faster than current HDDs [18].

Graefe [7] reconsiders the trade-off between keeping data in RAM and retrieving it as needed from non-volative storage in the context of a three-level memory hierarchy in which flash memory is positioned between RAM and disk. The cited paper recommends disk pages of 256KB and flash pages of 4KB to maximize B-tree utility per I/O time. Interestingly, these page sizes derive retention times of about 5 minutes, reinforcing the various forms of the "five-minute rule".

Both Myers [15] and Lee et al. [11] measure the performance of unmodified database algorithms when the underlying storage is a flash SSD. Myers considers B-tree search, hash join, index-nested-loops join, and sort-merge join, while Lee et al. focus on using the flash SSD for logging, sorting, and joins. Both conclude that using flash SSDs provides better performance than using hard disks.

A number of database algorithms have been designed especially for flash characteristics. These generally emphasize random reads and avoid random writes (whereas traditional algorithms try to avoid any random I/O). Lee et al. [10] modify database page layout to make writing and logging more efficient. Clementsen and He [3] improve I/O performance with a fully vertically partitioned storage structure that stores each column separately on either the HDD or SSD. Ross [19] proposes new algorithms for counting, linked lists, and B-trees that minimize writes. Nath and Gibbons [16] define a new data structure, the B-file, for maintaining large samples of tables dynamically. Their algorithm writes only completely new pages to the flash store. They observe that writes of pages to different blocks may be interleaved efficiently on flash SSDs. Li et al. [12] use a B-tree in memory to absorb writes, together with several levels of sorted runs of the data underneath the tree. This structure uses only sequential writes to periodically merge the runs and random reads to traverse the levels during a search. Finally, Koltsidas and Viglas [9] consider hybrid SSD/HDD configurations of databases systems and design a buffer manager that dynamically decides whether to store each page on a flash SSD or a hard disk, based on its read and write access patterns. However, they assume all page accesses are random.

3 Fast Scans and Joins for SSD-only Databases

In the context of workloads that include complex queries, we modify traditional query processing techniques to leverage the fast random reads of flash SSDs. We have three goals: (1) avoid reading unnecessary attributes during scan selections and projections, (2) reduce I/O requirements during join computations by minimizing passes over participating tables, and (3) minimize the I/O needed to fetch attributes for the query result (or any intermediate node in the query plan) by performing the fetch operation as late as possible.

We advocate a column-based layout such as PAX [1] within each database page. We used a PAX-based page layout to implement *FlashScan*, a scan operator that reads from the SSD only those attributes that participate in a query. FlashScan proactively evaluates predicates before fetching additional attributes from a given row, further reducing the amount of data read.

Building on FlashScan's efficient extraction of needed attributes, we introduce *FlashJoin*, a general pipelined

join algorithm that minimizes accesses to relation pages by retrieving only required attributes, as late as possible. FlashJoin consists of a binary join kernel and a separate fetch kernel. Multiway joins are implemented as a series of two-way pipelined joins. Our current implementation uses a hybrid-hash join algorithm [20]; any other join algorithm may be used instead. Subsequently, FlashJoin's fetch kernel retrieves the attributes for later nodes in the query plan as they are needed, using different fetching algorithms depending on the join selectivity and available memory. FlashJoin significantly reduces the memory and I/O needed for each join in the query.

3.1 FlashScan Operator

Figure 1 shows the page organization of both NSM and PAX for a relation with four attributes. In general, for an n-attribute relation, PAX divides each page into n *minipages*, where each minipage stores the values of a column contiguously. When a page is brought into main memory, the CPU can access only the minipages needed by the query [1].

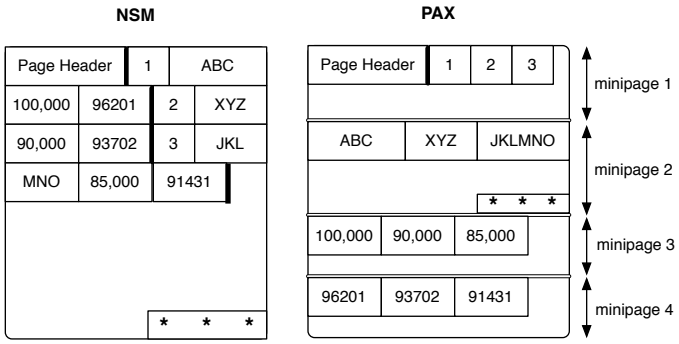

Figure 1: Disk pages in NSM (left) and PAX (right) storage layout of a relation with four attributes.

Data transfer units in main memory can be as small as 32 to 128 bytes (the size of a cacheline), but a database page is typically 8K to 128K. With SSDs, the minimum transfer unit is 512 to 2K bytes, which allows us to selectively read only parts of a regular page. *FlashScan* takes advantage of this small transfer unit to read only the minipages of the attributes that it needs. As discussed in [21], column-store systems can enjoy performance benefits similar to FlashScan on SSDs, but our techniques are easier to integrate in existing row-based database systems than building a column-store from scratch.

3.2 FlashJoin Operator

FlashJoin is a multi-way equi-join algorithm, implemented as a pipeline of stylized binary joins. Like FlashScan, FlashJoin avoids reading unneeded attributes and postpones retrieving attributes until needed. Each binary join in the pipeline is broken into two separate pieces, a *join kernel* and a *fetch kernel*, each of which is implemented as a separate operator, as shown in Figure 2. The join kernel computes the join and outputs a join index. Each join index tuple consists of the join attributes as well as the row-ids (RIDs) of the participating rows from base relations. The fetch kernel retrieves the needed attributes using the RIDs specified in the join index. FlashJoin uses a *late materialization* strategy, in which intermediate fetch kernels only retrieve attributes needed to compute the next join and the final fetch kernel retrieves the remaining attributes for the result.

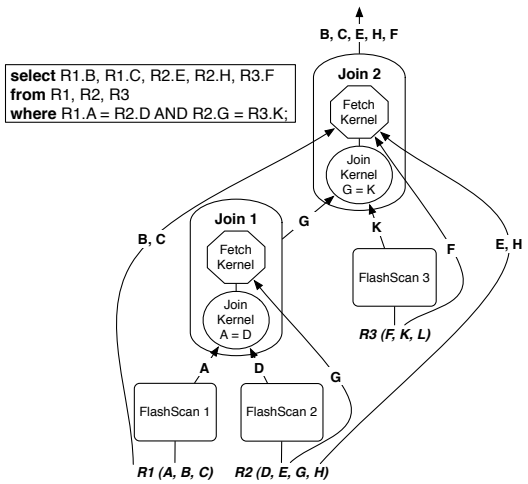

Figure 2: Execution strategy for a three-way join when FlashJoin is employed.

The *join kernel* leverages FlashScan to fetch only the join attributes needed from base relations. The join kernel is implemented as an operator in the iterator model. In general, it can implement any existing join algorithm: block nested loops, index nested loops, sort-merge, hybrid hash, etc. We explore here the characteristics of a join kernel that uses the

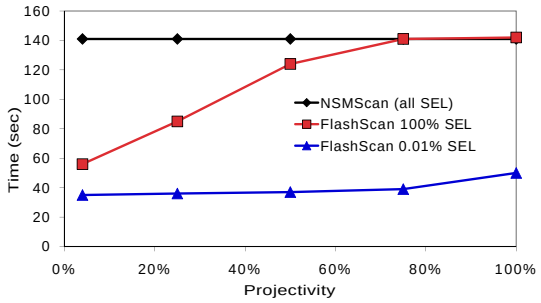

Figure 3: Scans on SSDs: FlashScan is faster than traditional scan when few attributes (small projectivity) or few rows (small selectivity – SEL) are selected.

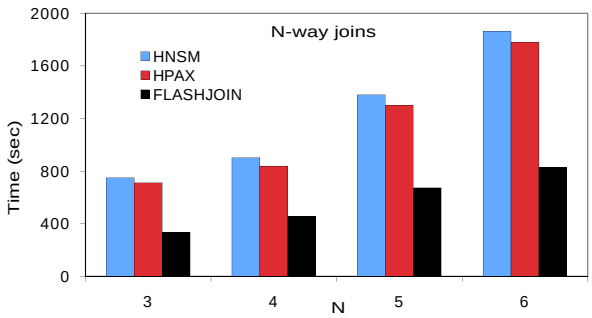

Figure 4: Multi-way joins on SSDs: FlashJoin is at least 2x faster than hybrid hash join over either traditional row-based (HNSM) or PAX layouts.

hybrid-hash algorithm [4]. This kernel builds an index (hash table) on the join attribute and RID from the inner relation and probes the index in the order of the outer. Compared to a traditional hash join, this join kernel is more efficient in two important ways. First, the join kernel needs less memory, thus many practical joins that would otherwise need two passes can complete in one pass. Second, when the join kernel spills to disk, materializing the runs is cheaper. Although we have only explored hybrid-hash, we believe a sort-merge kernel would offer similar results because of the duality of hash and sort [5]. Different strategies for the fetch kernel along with various optimizations can be found in [21]. FlashJoin is inspired by Jive/Slam-joins [13] (those algorithms were limited to two-way non-pipelined joins), and its late materialization strategy bears similarities to radix-cluster/decluster joins [14] which were optimized for main-memory operation.

3.3 Evaluation inside PostgreSQL

We implemented the proposed techniques inside PostgreSQL. In addition to incorporating a new scan and join operator, we modified the buffer manager and the bulk loader to support both the original and PAX layouts. We experimented with an enterprise SSD drive using synthetic datasets and TPC-H queries. Figure 3 compares the performance of FlashScan to Postgres' original scan operator (NSMScan) as we vary the percentage of tuple length projected from 4% to 100% (experimentation details are in [21]). FlashScan (100% selectivity, middle line) is up to 3X faster than the traditional scan for few projected attributes as it exploits fast random accesses to skip non-projected attributes. For low percentages of selectivity (0.01%, bottom line), FlashScan consistently outperforms the traditional scan by 3-4x, as it avoids reading projected attributes that belong to non-qualifying tuples. The original scan reads all pages regardless of selectivity.

Figure 4 shows the performance improvements when running multiway joins in Postgres using our FlashJoin algorithm. FlashJoin uses the drive more efficiently than hybrid-hash join over both traditional row-based (NSM) and column-based (PAX) layouts, by reading the minimum set of attributes needed to compute the join, and then fetching only those attributes that participate in the join result. By accessing only the join attributes needed by each join, FlashJoin also reduces memory requirements, which is beneficial in two ways: it decreases the number of passes needed in multi-pass joins (hence speeding up the join computation), and it frees up memory space to be used by other operators (hence leading in improved overall performance and stability in the system).

4 Sorting in a Memory Hierarchy

Sorting is a fundamental function used by database systems for a large variety of tasks. Sorting algorithms and techniques thus have a significant impact upon database performance. Informed by prior work such as [5, 6, 17, 8], we now outline techniques for sorting efficiently in database servers with flash memory in addition to traditional disk drives. The SSD advantage is access time; their disadvantages are capacity and cost/capacity.

In other words, very large operations require "spilling" from SSD to traditional disk, as do large databases if economy is a factor, i.e., the cost to purchase, house, power, and cool the computer system. On the other hand, data transfer to and from SSD storage is efficient even if the unit of transfer (or page size) is fairly small. Whereas optimal page sizes for traditional disk drives are in the 100s of KB, optimal page sizes for SSD are at most in the 10s of KB. In the case of an external merge sort with a fixed amount of RAM, runs on SSD storage permit a much higher merge fan-in than runs on traditional disks.

4.1 Using Flash Memory Only

If the input volume is too large for a traditional in-memory sort yet sufficiently small to fit on the available flash memory, a traditional external merge sort is the appropriate algorithm, with only minor modifications. Figure 5 illustrates how run generation with flash as external storage is more similar than different from run generation with a traditional disk as external storage.

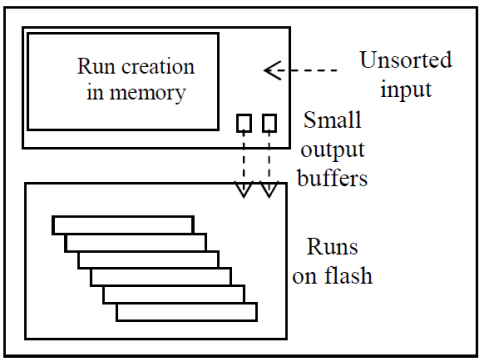

Figure 5: Runs on flash.

Note that due to the fast latency, there is little incentive to employ large transfers or pages. Small pages, e.g., 4 KB, are entirely appropriate. As a second consequence of short access latencies, the importance of double-buffered I/O diminishes. For example, a disk operation of 1 MB takes about 16.7 ms (10 ms + 1 MB ÷ 150 MB/s), equivalent to 33 million cycles in a processor running at 2 GHz. A flash disk operation of 4 KB takes merely 0.041 ms (0.025 ms + 4 KB ÷ 250 MB/s) or 81,000 cycles, a difference of almost three orders of magnitude. This advantage of flash becomes even more apparent as SSD technology improves, offering bandwidth increasingly faster than that of HDD.

4.2 Run Generation with Disk and Flash

If the input size exceeds the available capacity of flash memory, runs on disk must be employed. For best use of both flash and disk, initial runs on flash can be merged to form initial runs on disk. The size of runs on disk will be similar to the available capacity of flash memory.

The page size on flash and disk may differ. In fact, it should differ to exploit the fast access latency of flash devices and the fast bandwidth of disk devices. Thus, the allocation of memory between merge input and merge output will be different from traditional merge operations. For example, half of the available memory might be dedicated to buffering merge output in order to enable large disk transfers. Nonetheless, the small page size on flash may still permit a fairly high merge fan-in.

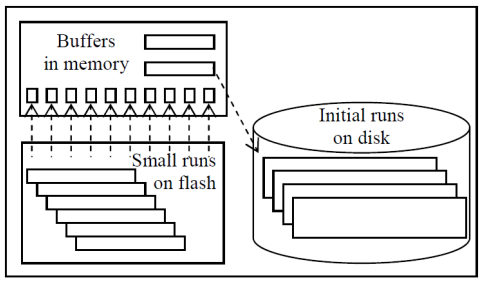

Figure 6: Merging from flash to disk.

Figure 6 illustrates run generation on disk by means of merging memory-sized runs gathered on flash memory. The most important aspect is the difference in page sizes appropriate for the different access latencies on flash memory and disk. Due to small input buffers, the fan-in is quite high even for a fairly small memory and thus the initial runs on disk are much larger than the available memory and the small runs on flash.

The merge bandwidth is limited by the slower of input and output. Two techniques will typically be required for matched bandwidth and thus best overall performance. First, the flash memory must remain far from full such that wear leveling can remain effective without excessive overhead for creation of free space (also a problem in log-structured file systems [OD 89], which are quite similar). In other words, flash capacity must be traded

for performance. In fact, some vendors of flash hardware seem to do this already, i.e., some devices contain actual capacity much larger than their externally visible storage capacity, and some devices make less storage space available than is built in. It might create a point of contention whether the internal or the external capacity should be the stated storage capacity. Second, to match input and output bandwidth, the ratio of the number of flash devices versus the number of disks must be adjusted to reflect their relative read and write bandwidths. Organization of flash devices as the array can be hidden within a hardware device (i.e., a single SATA device contains an array of flash chips) or may be provided by the operating system or the database management system.

4.3 Merging with Disk and Flash

The final sort output is produced merging all runs on disk. Ideally, the availability of flash memory is exploited such that the merge operation benefits from both the fast access latency of flash memory and the fast transfer bandwidth of traditional disks.

If the merge operation from flash to disk may be viewed as "assembling" large disk pages from small pages on flash, merging runs on disk may be viewed as the opposite, i.e., "disassembling" large disk pages. Specifically, large disk pages are moved from disk to flash memory, if necessary via memory (if no disk-to-flash DMA is available). This move exploits the fast disk bandwidth and requires flash arrays and internal organization with matching bandwidth.

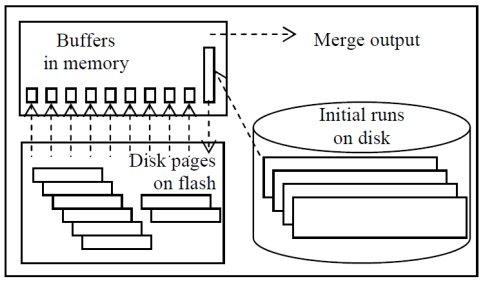

Figure 7: Merging from disk via flash.

These large pages on flash are divided into small pages and those are merged with a high fan-in, thus exploiting the fast access latency of flash memory. In order to make this work, the large pages on disk must indeed be organized as groups of small pages, i.e., each small page must contain an appropriate page header, indirection information for variable- length records, etc.

Figure 7 illustrates sort output formed from runs on disk. A large buffer in memory may be required to move large disk pages to flash memory. The actual merge operation in memory fetches small pages from flash, and replenishes flash memory by fetching additional large pages from disk.

5 Conclusions

SSDs constitute a significant shift in hardware characteristics, comparable to large CPU caches and multi-core processors. In this paper, we explore how for database systems, SSDs change not only power efficiency but also query execution efficiency for business intelligence and other complex queries.

We first demonstrated the suitability of a column-based page layout for accelerating database scan projections and selections on SSDs, through a prototype implementation of *FlashScan*, a scan operator that leverages the columnar layout to improve read efficiency, inside PostgreSQL. We presented *FlashJoin*, a general pipelined join algorithm that minimizes memory requirements and I/Os needed by each join in a query plan. FlashJoin is faster than a variety of existing binary joins, mainly due to its novel combination of three well-known ideas that in the past were only evaluated for hard drives: using a column-based layout when possible, creating a temporary join index, and using late materialization to retrieve the non-join attributes from the fewest possible rows. We incorporated the proposed techniques inside PostgreSQL. Using an enterprise SSD, we were able to speed up queries, ranging from simple scans to multiway joins and full TPC-H queries, by up to 6x.

We next explored how adding flash to the memory hierarchy impacts external merge sort. We chose to investigate sorting because it is prototypical for many query processing algorithms. We consider both the case where the input data fits into flash, as well as how flash can participate in external merge sort. With regard to

the latter, we first explore using flash to store runs, then using flash to merge runs. Balancing the bandwidths of input and output devices during the merge process, challenging even when using just HDD, is complicated by the addition of SSD. Due to the lack of mechanical seeking, accesses are fast and inexpensive. In addition, read and write bandwidths differ greatly in SSDs whereas they are quite similar in HDDs. For these reasons, bandwidth balancing strategies must be reconsidered in the face of SSD.

Our work has focused so far on storage formats and query execution algorithms that can take advantage of the performance characteristics of SSDs. One new consideration for query optimization is that many query execution plans that were not appropriate for HDDs become appropriate for SSDs. The fast access times of SSD shift the break-even point between table scans and index retrieval using both secondary and primary indexes. For example, if index-to-index navigation out-performs a table scan only if less than 1% of 1% of a table is needed, flash devices might shift the break-even point to 1%. In other words, many more queries can use indexes and thus a small concurrency control foot print that in the past would use table scans. Future research and industrial practice will show the extent to which this is true, and whether it might affect the feasibility and efficiency of real-time data warehousing with continuous trickle loading.

References

[1] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3), 2002.

[2] L. Bouganim, B. Jonsson, and P. Bonnet. uflip: Understanding flash IO patterns. In *Proc. CIDR*, 2009.

[3] D. S. Clementsen and Z. He. Vertical partitioning for flash and hdd database systems. *J. Syst. Softw.*, 83(11):2237–2250, 2010.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[5] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[6] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3):10, 2006.

[7] G. Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM*, 52(7):48–59, 2009.

[8] G. Graefe and P.-Å. Larson. B-tree indexes and cpu caches. In *ICDE*, pages 349–358, 2001.

[9] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB*, 1(1):514–525, 2008.

[10] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *Proc. SIGMOD '07*, pages 55–66, New York, NY, USA, 2007.

[11] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *Proc. SIGMOD '08*, pages 1075–1086, New York, NY, USA, 2008.

[12] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proc. ICDE*, 2009.

[13] Z. Li and K. A. Ross. Fast joins using join indices. *VLDB J.*, 8:1–24, 1999.

[14] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-conscious radix-decluster projections. In *Proc. VLDB*, 2004.

[15] D. Myers. On the use of NAND flash memory in high-performance relational databases. *MIT Msc Thesis*, 2008.

[16] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.*, 1(1):970–983, 2008.

[17] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. Alphasort: A cache-sensitive parallel external sort. *VLDB J.*, 4(4):603–627, 1995.

[18] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, 2009.

[19] K. A. Ross. Modeling the performance of algorithms on flash memory devices. In *Proc. DaMoN '08*, pages 11–16, New York, NY, USA, 2008.

[20] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[21] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proc. SIGMOD*, pages 59–72, 2009.