Do Query Optimizers Need to be SSD-aware?

Steven Pelley
University of Michigan
2260 Hayward St.
Ann Arbor, Michigan
spelley@umich.edu

Thomas F. Wenisch
University of Michigan
2260 Hayward St.
Ann Arbor, Michigan
twenisch@umich.edu

Kristen LeFevre
University of Michigan
2260 Hayward St.
Ann Arbor, Michigan
klefevre@umich.edu

ABSTRACT

Flash-based solid state disks (SSDs) are beginning to supplant conventional rotating disks for performance-critical data in myriad DBMS applications, including decision support systems. Though SSDs provide the same block-oriented storage abstraction as conventional disks, their performance characteristics differ drastically. Whereas SSDs provide relatively modest improvements in sequential transfer rates (e.g., perhaps 2× improvement), they can provide over 100× improvement for random reads, resulting in similar sustained transfer rates regardless of the access pattern. Conventional query optimizers assume a storage cost model where sequential IOs are far less costly than random IOs, and select access paths and join algorithms based on this assumption. Given the drastic change in SSD performance characteristics, intuition suggests that optimizer cost models must be updated (e.g., to prefer non-clustered index scans more frequently).

Surprisingly, our empirical investigation using a commercial DBMS finds it is not necessary to adjust query optimization when shifting relations from disk to flash—an SSD-oblivious optimizer generally makes effective choices. We make two main observations. First, we demonstrate both empirically and analytically that the range of selectivities for which an unclustered index scan can benefit from SSDs' fast random reads is so narrow that it is inconsequential in practice. Second, our measurements show that the performance variations across alternative join algorithms on SSDs are generally smaller than the corresponding variation on disks and are dwarfed by the 5× to 6× performance boost of shifting data from disk to SSD. We conclude that existing query optimizers largely make correct decisions even when treating all storage devices as conventional disks, and the small and inconsistent performance gains available by making query optimizers SSD-aware are not worth the effort.

1. INTRODUCTION

For decades, database management systems (DBMSs) have used rotating magnetic disks to provide durable storage. Though inexpensive, disks are slow, particularly for non-sequential access patterns due to high seek latencies. With the rapid improvements in storage density and drop in price of Flash-based Solid State Disks (SSDs), DBMS administrators are beginning to supplant conventional rotating disks with SSDs for performance-critical data in myriad DBMS applications. Though SSDs are several factors more expensive than conventional disks (in terms of $ per GB), they provide modest (2×) improvements in sequential IO and drastic (over 100×) improvements for random IO, closing the gap between these access patterns.

Many components of modern DBMSs have been designed to work around the adverse performance characteristics of disks (e.g., page-based buffer pool management, B-tree indexes, advanced join algorithms, query optimization to avoid non-sequential IO, prefetching, and aggressive IO request reordering). As SSDs present substantially different performance trade-offs, over the past few years, researchers have begun to examine how SSDs should be best deployed for a variety of storage applications [3, 4], including DBMSs [7, 14, 8, 1, 13]. A common theme among these studies is to leverage the better random IO performance of SSDs through radical redesigns of index structures [14, 8] and data layouts [1, 13]. However, we note that, even within the confines of conventional storage management and indexing schemes in commercial DBMSs, there may be substantial opportunity to improve query optimization by making it SSD-aware.

In this study, we examine the implications that moving a database from disk to Flash SSDs will have for query optimization in conventional commercial DBMSs. We focus on optimization of read-only queries (e.g., as are common in decision support workloads) as these operations are less sensitive to the SSD adoption barriers identified in prior work, such as poor SSD write/erase performance [4] and write endurance [10]. Conventional query optimizers assume a storage cost model where sequential IOs are far less costly than random IOs, and select access paths and join algorithms based on this assumption. The recent literature [1] suggests that on SSDs, optimizers should instead favor access paths using non-clustered indexes more frequently. Furthermore, as SSDs change the relative costs of computation, sequential, and random IO, the relative performance of alternative join algorithms should be re-examined, and optimizer cost models updated.

Contrary to our initial expectations, our empirical investigation using a commercial DBMS finds *it is not necessary to make any adjustments to the query optimizer when moving data from disk to Flash*—an SSD-oblivious optimizer generally makes effective choices. We demonstrate this result, and explore why it is the case, in two steps.

First, we analyze the performance of scan operators. Classic rules of thumb suggest that non-clustered index scans are preferable at low selectivity (e.g., below 10%), whereas a relation scan is faster at high selectivity, because it can leverage sequential IOs.

Intuition suggests that the optimizer should prefer index scans at much higher selectivities on SSDs. We demonstrate analytically and empirically that this intuition is false—the range of selectivities for which an index scan operation can benefit from SSDs' fast random reads is so narrow that it is inconsequential in practice.

Second, we measure the relative performance of hybrid hash and sort-merge joins on disk and Flash. Our results indicate that the performance variation between the join algorithms is typically smaller (and often negligible) on Flash, and is dwarfed by the 5× to 6× performance boost of shifting data from disk to SSD. We conjecture that because commercial DBMSs have been so heavily optimized to hide the long access latencies of disks (e.g., through sophisticated prefetching and buffering), they are largely insensitive to the latency improvements available on SSDs. Overall, we conclude that the small and inconsistent performance gains available by making query optimizers SSD-aware are not worth the effort.

While our results are specific to existing DBMSs we believe our conclusions will extend to many Flash-optimized database algorithms. As this paper focuses on the performance of commercial DBMSs we omit discussion of algorithms not included in our DBMS and leave it as future work to implement and compare Flash-specific improvements.

The remainder of this paper is organized as follows. In Section 2, we provide brief background on Flash SSD operating principles and performance characteristics relevant to our analysis. We then outline the widely-accepted intuition that leads us to the (incorrect) belief that query optimization should be SSD-aware in Section 3. The remainder of our study refutes this expectation both empirically and (in the case of scans) analytically. We discuss our measurement approach in Section 4 and discuss scans in Section 5 and joins in Section 6. We survey related work in Section 7 and conclude in Section 9.

2. BACKGROUND

Driven by the popularity of mobile devices, Flash memory has quickly improved in both storage density and cost to the point where it has become a viable alternative for durable storage even in enterprise-class systems. Unlike conventional rotating hard disks, which store data using magnetic materials, Flash stores charge on a floating-gate transistor, forming a memory cell. These transistors are arranged in arrays resembling NAND logic gates, after which the "NAND Flash" technology is named. This layout gives NAND Flash a high storage density relative to other memory technologies. Though dense, the layout sacrifices byte addressability and some read latency—an entire array (a.k.a. page, typically 2KB to 4KB) must be read in a single operation—making NAND Flash more appropriate for block-oriented IO than as a direct replacement for RAM.

One of the difficulties of Flash devices is that a cell can be more easily programmed (by adding electrons to the floating gate) than erased (removing these electrons). Erase operations require both greater energy and latency, and typically can be applied only at coarse granularity (e.g., over blocks of 128KB to 512KB). Moreover, repeated erase operations cause the Flash cell to wear out over time, limiting the maximum lifetime of the cell (e.g., to 10^5 to 10^6 writes [10]). Recent Flash devices further increase storage density by using several distinct charge values to represent multiple bits in a single cell at the cost of slower accesses and even shorter lifetimes. In this study, we focus on scan and join operations, which incur few writes (writes arise only due to the use of temporary storage during sort and hash joins). Hence, the performance and endurance challenges of writing to Flash are orthogonal to our study.

	Disk	Flash
Model	WD VelociRaptor 10Krpm	OCZ RevoDrive
Capacity	300gb	120gb
Price	$164	$300
Random Read	10ms	90µs
Seq. Read	120mb/s	190mb/s

Table 1: Disk Characteristics

A Flash-based SSD wraps an array of underlying Flash memory chips with a controller that manages capacity allocation, mapping, and wear leveling across the individual Flash devices. The controller mimics the interface of a conventional (e.g., SATA) hard drive, allowing Flash SSDs to be drop-in replacements for conventional disks.

2.1 Flash Performance.

As previously noted, Flash SSDs provide substantially better performance than disks, particularly for random reads, but at higher cost. Table 1 lists specifications of a typical Flash SSD as compared to a 10,000 RPM hard drive; these particular devices are used in the experiments we describe later. Though neither of these devices are the highest-performing available today, they are representative of the mid-range of their respective markets. The latency for a random read is over 100× better on the SSD than on the disk, while the sequential read bandwidth is 1.6× better. Unlike disks, where each random read incurs mechanical delays (disk head seek and rotational delays), on SSDs, a random read is nearly as fast as a sequential read. Flash memory arrays, disk interfaces, and the operating system all contribute to Flash read latency. Typical SSDs read in about 100 μs.

3. A (FALSE) CASE FOR SSD-AWARENESS

As noted in Section 1, both prior work [1, 13] and our own initial expectations suggested that adding SSD-awareness to the query optimizer of a commercial DBMS should allow it to arrive at better query plans and improve performance. Though we later experimentally refute these arguments, we begin by explaining the chain of reasoning that lead us to investigate SSD-aware query optimization.

3.1 Scans

Whenever a query accesses a table, the query optimizer must choose an access path for that table. Work on access path selection dates back to the late 1970s [11]. There are two classic scan operators implemented by nearly all commercial DBMS systems. When no indexes are available, the only choice is to perform a *relation scan*, where all data pages in the table are read from disk and scanned tuple-by-tuple to select relevant tuples. When a relevant index is available, the DBMS may instead choose to perform an *index scan*, where the execution engine traverses the relevant portion of the index and fetches only pages containing relevant tuples as needed.

For clustered indexes, an index scan is nearly always the preferred access path, regardless of the underlying storage device. For non-clustered indexes, whether the optimizer should choose a relation scan or index scan depends on the selectivity of the query; relation scans have roughly constant cost regardless of selectivity, whereas index scan costs grow approximately linearly with selectivity. When selectivity is low, the index scan provides greater performance because it minimizes the total amount of data that must be

transferred from disk. However, as selectivity increases, the fixed-cost relation scan becomes faster. Though the relation scan reads the entire table, it can do so using sequential rather than random IO, leveraging the better sequential IO performance of rotating hard disks. A classic rule of thumb for access path selection is to choose a relation scan once selectivity exceeds ten percent [9]. (We show later that this rule of thumb is flawed even for disk).

As noted in the literature [1], we expect that the break-even point between index and relation scans will occur at a substantially higher selectivity on SSDs than on disk, because the performance gap between sequential and random IO is much smaller. This observation in turn implies that the query optimizer must be aware of the performance characteristics of the storage device to choose the correct access path, and hence, must be SSD-aware.

Many commercial databases (including the one used in our study) implement a third, hybrid scan operator, which we call the *rowid-sort scan*. In this scan operator, the unclustered index is scanned to identify relevant tuples. However, rather than immediately fetching the underlying data pages, the rowid of each tuple is stored in a temporary table, which is then sorted at the end of the index scan. Then, the pages identified in the temporary table are fetched in order, and relevant tuples are returned from the page.

The rowid-sort scan has the advantage that each data page will be fetched from disk only once, even if multiple relevant tuples are located on the page. (Pages might be fetched and evicted from the buffer pool multiple times in a conventional index scan.) Furthermore, because data pages are read in order, disk seeks are minimized, and the scan is highly amenable to aggressive prefetching. Thus, the rowid-sort scan can achieve performance approaching that of sequential IO on conventional disks. However, the scan incurs some overhead due to the sort operation, and cannot be pipelined. Rowid-sort scan is the optimal access path for intermediate selectivities. On an SSD, we expect the range of selectivities in which rowid-sort scan is optimal to narrow—because random IOs are so much less costly on SSD, avoiding the sort operation may outweigh the savings from fetching a handful of pages more than once.

3.2 Joins

One of the most important decisions a query optimizer must make is to choose an appropriate join algorithm for a given query. The development of join algorithms and optimization strategies dates back over 30 years [11, 12]. Most commercial DBMS systems implement variants of at least three join algorithms: nested-loop join, sort-merge join, and hybrid hash join. At a high level, the nested-loop join iterates over the inner relation for each tuple of the outer relation; the sort-merge join sorts both relations and then performs concurrent scans of the sorted results; and the hybrid hash join forms in-memory hash tables of partitions of the inner relation and then probes these with tuples from the outer relation.

The relative performance of these algorithms depends on a complex interplay of memory capacity, relation sizes, and the relative costs of random and sequential IOs. One example performance model that captures this interplay was proposed by Haas and co-authors [6]. Their model estimates the number disk seeks and the size of each data transfer and weights each by a cost based on assumed characteristics of the IO device. The model further identifies the optimal buffering strategy for the various phases of each join algorithm. As seek and random/sequential transfer times are central parameters of this model, the break-even points (in terms of inner and outer relation sizes) between the various join algorithms shift as a function of disk vs. SSD performance characteristics. Our testing indicated that this particular model was not accurate in absolute terms in predicting performance for the commercial DBMS we study (which is unsurprising, since we have no reason to expect that our DBMS uses the buffering strategy and precise join implementations outlined by Haas). Nonetheless, the intuition underlying the model provides significant evidence that query optimizers should, at the very least, require updated constants to choose join algorithms well for SSDs.

4. METHODOLOGY

The objective of our empirical study is to contrast the performance of alternative scan and join algorithms for the same queries to discover whether the optimal choice of access path or join algorithm differs between SSDs and conventional disks. For either storage device, the optimal access path depends on the selectivity of the selection predicate(s). The optimal join algorithm depends on several factors: the sizes of the inner and outer relations, the selectivity and projectivity of the query, the availability of indexes, and the available memory capacity. Our goal is to determine whether the regions of the parameter space where one algorithm should be preferred over another differ substantially between SSD and disk because of the much better random read performance of the SSD. In other words, we are trying to discover empirically cases where an access path or join algorithm that is an appropriate choice for disk results in substantially sub-optimal performance on an SSD, suggesting that the optimizer must be SSD-aware.

We carry out our empirical investigation using IBM DB2 Enterprise Server Edition version 9.7. Our experiments use the Wisconsin Benchmark schema [2] to provide a simple, well-documented dataset on which to perform scans and joins. Though this benchmark does not represent a particular real-world application, modeling a full application is not our intent. Rather, the Wisconsin Benchmark's uniformly distributed fields allows us to control precisely the selectivity of each query. Whereas real world queries are more complicated than the simple scans and joins we study, these simple microbenchmarks reveal the underlying differences between the storage devices and scan/join algorithms most clearly. We include an aggregate in all queries to avoid materializing output tables as we are primarily interested in isolating other database operations. We run queries on a Pentium Core Duo with 2GB main memory, a 7200 RPM root disk drive, and the conventional and SSD database disks described in Table 1. Both the hard disk and SSD were new at the beginning of our experiments. While other work has shown that SSD performance may degrade over the lifetime of the device we did not observe any change in performance.

Note that we are not concerned with the optimization decisions that DB2 presently makes for either disk or SSD; rather, we are seeking to determine the ground truth of which algorithm a correct optimizer should prefer for each storage device. We use DB2's optimization profiles to explicitly set the query plan for each query.

5. SCAN ANALYSIS

We now turn to our empirical and analytic study of scan operators. We demonstrate that although the expectations outlined in Section 3 are correct in principle, the range of selectivities for which an index scan operation can benefit from SSDs' fast random reads is so narrow that it is inconsequential in practice.

Empirical Results. We compare the measured performance of the different scan operators as a function of selectivity on SSD and disk. Our objective is to find the break-even points where the optimal scan operator shifts from index scan to rowid-sort scan and finally to relation scan on each device. We issue queries for ranges of tuples using a uniformly distributed integer field on a table with

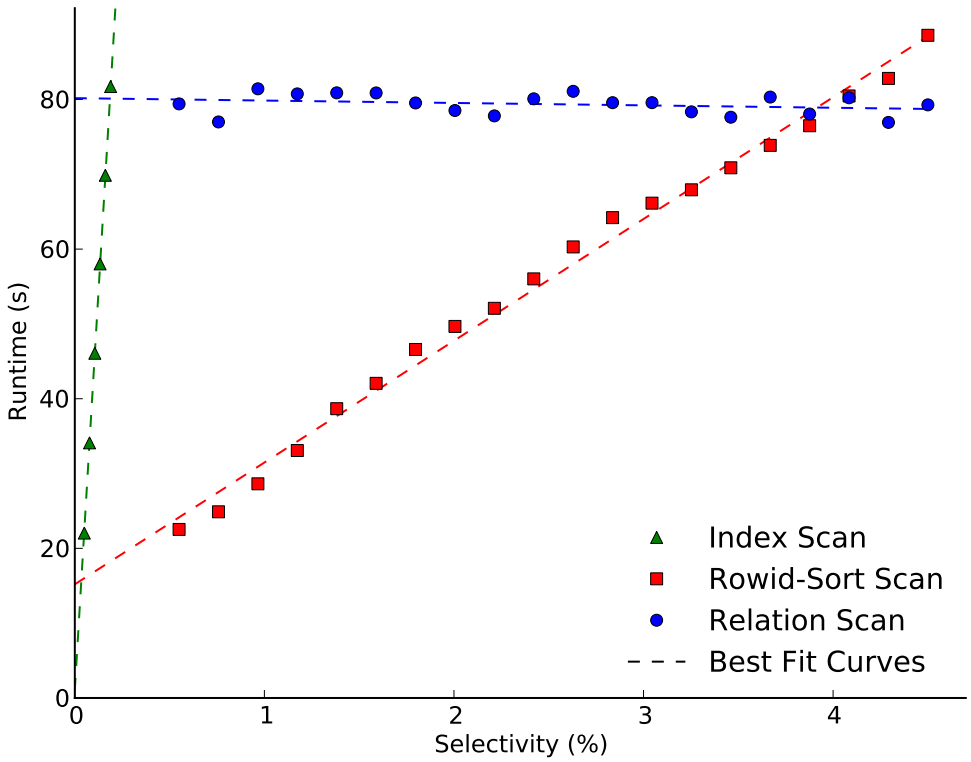

Figure 1: Scan operator performance on Disk. Relation scan outperforms the alternatives at selectivities above 4%, while index scan is optimal only for vanishingly small selectivities (e.g., single-tuple queries). Best fit curves drawn for convenience.

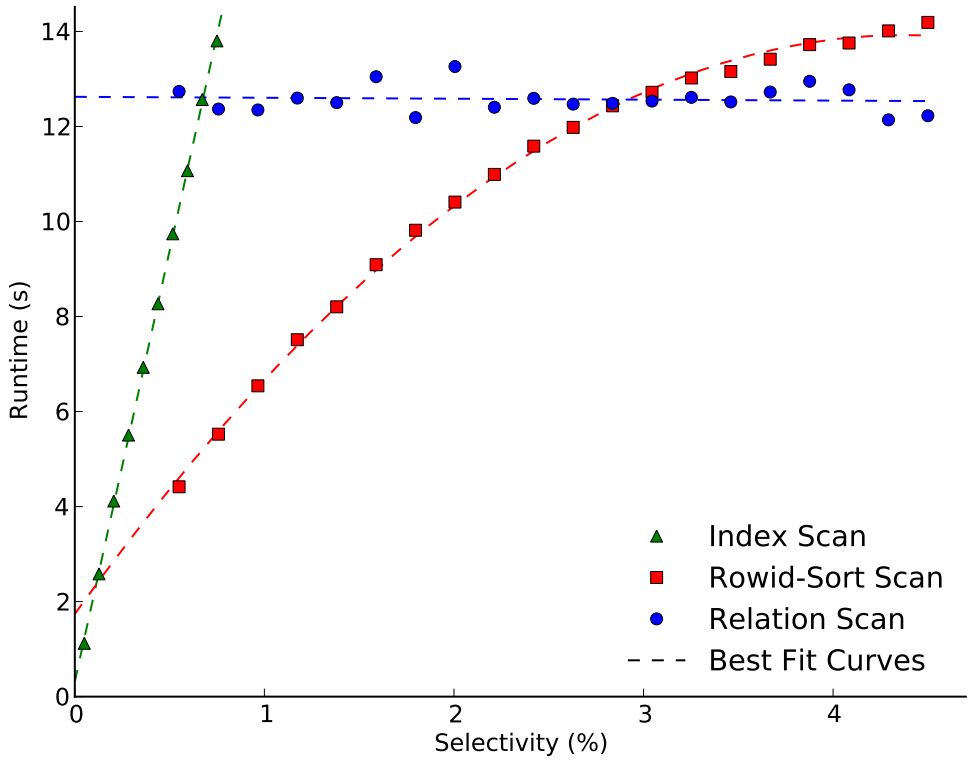

Figure 2: Scan operator performance on Flash SSD. Though both break-even points shift as our intuition suggests, the selectivities where the optimal decision differs between Disk and SSD are so narrow that the difference is inconsequential in practice. Best fit curves drawn for convenience.

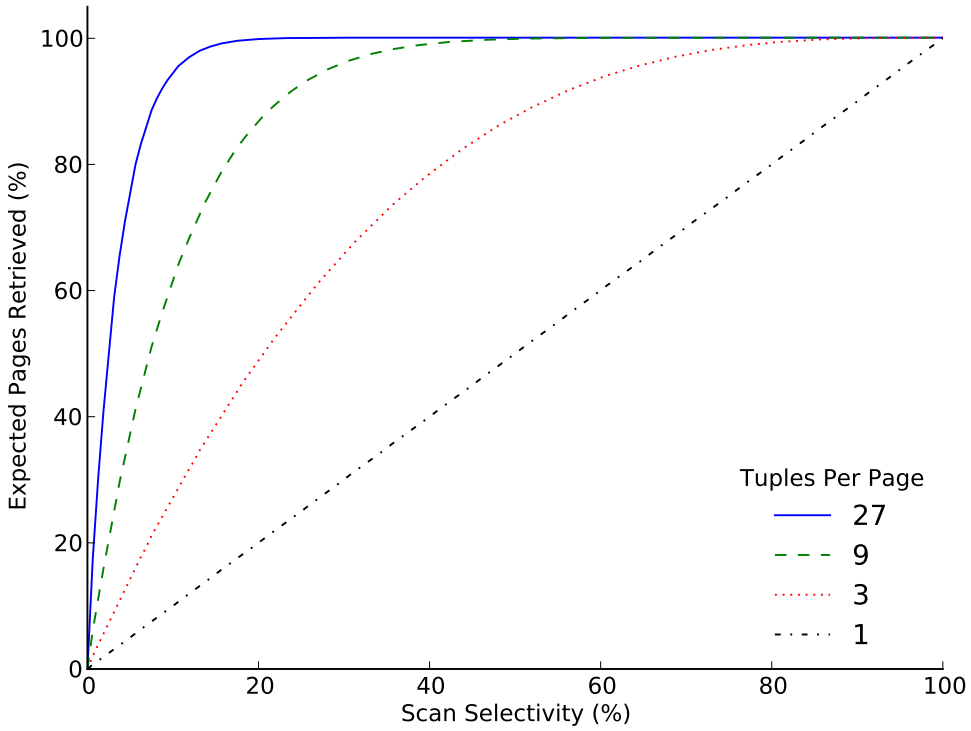

Figure 3: Index scans touch the majority of pages even at low selectivities.

10 million rows, or roughly 2 GB. We use a pipelined aggregation function to ensure that no output table is materialized.

Figures 1 and 2 report scan runtimes on disk and Flash SSD, respectively. The figures show the measured runtime of each scan (in seconds); lower is better. Recall from Section 3 that classic rules of thumb suggest that, on disk, the break-even point between index and relation scan should occur near 10% selectivity, and intuition suggests an even higher break-even point for SSD. Clearly, the conventional wisdom is flawed even for rotating disks; relation scan dominates above selectivities of just 4% (the trends shown in the figure continue to the right). Indeed, our results indicate that even DB2's query optimizer is erroneously choosing to perform index scans at selectivities well below 0.55% where both relation scan and rowid-sort scan still perform substantially better. In the intermediate range from about 0.1% to 4% selectivity the rowid-sort scan performs best.

However, what is more important for our analysis is to compare the locations of the break-even points across SSD and disk. Both crossover points shift in the directions we expect. The slope of the index scan curve is considerably shallower, and the break-even with the relation scan has shifted above 0.5% selectivity. Furthermore, the range in which rowid-sort scan is optimal is narrower. Nevertheless, the key take-away is that the range of selectivities for which the optimal scan *differs* across SSD and Disk is vanishingly small. Hence, it is unnecessary for the optimizer to be SSD-aware to choose the correct scan operator.

Whereas these measurements demonstrate our main result, they do not explain why index scans fail to leverage the random access advantage of Flash. We turn to this question next.

Analytic Results. Our results show that index scan underperforms at selectivities far below what the classic 10% rule of thumb suggests. The flaw in the conventional wisdom is that, when there are many tuples per page, the vast majority of *pages* need to be retrieved even if only a few *tuples* are accessed. (If we construe the 10% rule as applying to page- rather than tuple-selectivity, the guideline is more reasonable). Yue *et al.* provide an analytical formula for the expected number of pages retrieved given the size of the table, tuples per page, and selectivity [15], assuming tuples are randomly distributed among pages. Based on this formula, Figure 3 shows the expected percentage of pages retrieved as a function of query selectivity and tuples per page. When a page contains only a single tuple, clearly, the number of tuples and pages accessed are equal. However, as the number of tuples per page increases, the expectation on the number of pages that must be retrieved quickly approaches 100% even at small selectivities. As a point of reference, given a 4kb page size and neglecting page headers, the Wisconsin Benchmark stores 19 tuples per page while TPC-H's Lineitem and Orders tables store 29 and 30 tuples per page, respectively.

The implication of this result is that, for typical tuple sizes, the vast majority of a relation must be read even if the selectivity is but a few percent. Hence, with the exception of single-tuple lookups, there are few real-world scenarios where scan performance is improved by better random access latency under conventional storage managers that access data in large blocks. To benefit from low access latency, future devices will need to provide random access at tuple (rather than page) granularity. Until we have such devices, relation and rowid-sort scans will dominate and IO bandwidth will be the primary determinant of scan performance.

As a final note, many workloads other than DSS may benefit from the fast random IOs of SSDs. In particular, workloads that perform frequent queries for a few tuples, such as transaction processing workloads, will benefit from faster index lookup and low random access latency.

6. JOIN ANALYSIS

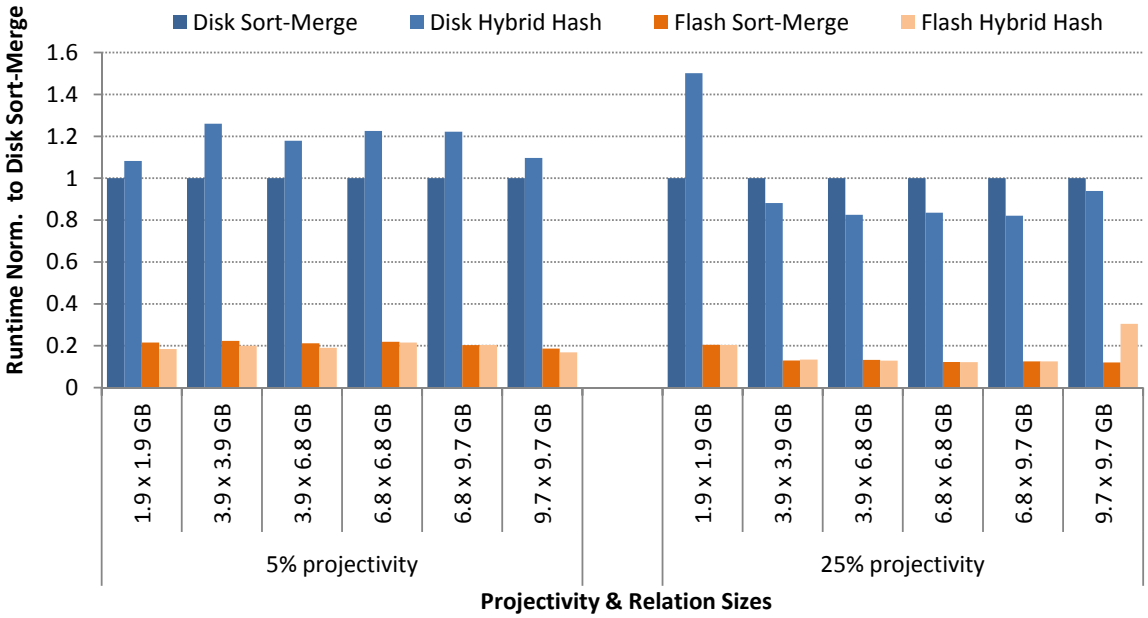

Figure 4: Join runtimes on Flash SSD and Disk, normalized for each join to the runtime of sort-merge on disk. Though there is significant variability in join algorithm performance on disk, performance variability on SSD is dwarfed by the 6× performance advantage of moving data from disk to SSD.

Selectivity	Table Sizes	Disk		Flash SSD	
		Sort-merge	Hybrid hash	Sort-merge	Hybrid hash
4%	1.9 x 1.9 GB	187	202	40	34
	3.9 x 3.9 GB	358	451	80	72
	3.9 x 6.8 GB	487	574	103	93
	6.8 x 6.8 GB	649	795	142	140
	6.8 x 9.7 GB	816	997	166	166
	9.7 x 9.7 GB	1084	1189	202	183
27%	1.9 x 1.9 GB	236	355	48	48
	3.9 x 3.9 GB	751	662	97	101
	3.9 x 6.8 GB	947	781	125	122
	6.8 x 6.8 GB	1415	1182	174	173
	6.8 x 9.7 GB	1581	1298	199	199
	9.7 x 9.7 GB	2081	1955	250	634

Table 2: Join runtimes in seconds. Variability in join runtimes is far lower on Flash SSD than on Disk.

We next study the variability in join performance across disk and Flash SSD. Again, our objective is to identify cases where the optimal join algorithm for disk consistently results in grossly sub-optimal performance on Flash SSD. Such scenarios imply that it is important for the optimizer to be SSD aware.

DB2 implements nested loop, sort-merge, and hybrid hash join operators. However, DB2 does not support a block nested loop join; its nested loop join performs the join tuple-by-tuple instead of prefetching pages or other blocks, relying on indexes to provide high performance. Hence, unless the join can be performed in memory, the nested loop grossly underperforms the other two algorithms for ad-hoc queries regardless of storage device and will not be selected by the query optimizer unless it is the only alternative (e.g., for inequality joins). We therefore restrict our study to a comparison of sort-merge and hybrid hash joins.

When a clustered index exists for a particular scan or join this index should almost always be used, regardless of the nature of the storage device. Hence, we do not include clustered indexes in our analysis. Furthermore, we evaluate only ad hoc joins (joins without indexes or "where" clauses). When indexes are available, the choice of whether or not to use the index is analogous to the choice of which scan operator to use for a simple select query, which is covered by our analysis of scans.

Because of the complexity of the interplay between available memory capacity and relation sizes for join optimization [6], we do not have a specific expectation that one join algorithm will universally outperform another on Flash SSD as opposed to disk. Rather, we perform a cross-product of experiments over a wide spectrum of relation sizes and output projectivities using the Wisconsin Benchmark database. Haas's model demonstrates the importance of the relative sizes of input relations and main memory capacity on join performance; hence we explore a spectrum from joins that are only slightly larger than available memory (joining two 1.9GB tables) to those that are an order of magnitude larger (joining two 9.7GB tables). We vary projectivity because we have discovered empirically that it impacts significantly the optimal join algorithm on disk, as it has a strong influence on partition size in hybrid hash joins. We perform queries with two projectivities: approximately 5% (achieved by selecting all the integer fields in the Wisconsin Benchmark schema), and approximately 25% (selecting an integer field and one of the three strings in the schema). In all experiments, we perform an equijoin on an integer field, and use an aggregation operator to avoid materializing the output.

We report results in graphical form in Figure 4 and absolute runtimes in Table 2. In Figure 4, each group of bars shows the relative performance of sort-merge and hybrid hash joins on disk (darker bars) and Flash SSD (lighter bars), normalized to sort-merge performance on disk. Lower bars indicate higher performance. We provide the same data in tabular form to illustrate the runtime scaling trends with respect to relation size, which are obscured by the normalization in the graph.

Two critical results are immediately apparent from the graph. First, Flash SSDs typically outperform disk by 5× to 6× regardless of join algorithm, a margin that is substantially higher than the gap in sequential IO bandwidth, but far smaller than the gap in random IO bandwidth (see Table 1). Hence, though both join algorithms benefit from the improved random IO performance of SSDs, the benefit is muted compared to the 100× device-level potential. Second, we note that, whereas there is significant performance variability between the join algorithms on disk (typically over 20%), with the exception of a single outlier, the variability is far smaller on Flash SSD (often less than 1%). From these results, we draw our central conclusion: although important on disk, the choice of sort-merge vs. hybrid hash join on SSD leads to inconsequential performance differences relative to the drastic speedup of shifting data from disk to Flash. Hence, we see no compelling reason to make the query optimizer SSD-aware; the choice it makes assuming the performance characteristics of a disk will yield near-optimal performance on SSD.

We highlight two notable outliers in our results. On disk, the best join algorithm is strongly correlated to query projectivity with the exception of the 1.9GB × 1.9GB join at 27% projectivity. Because the required hash table size for this join is close to the main memory capacity, we conjecture that this performance aberration arises due to DB2 selecting poor partition sizes for the join. Second, on Flash, we observe a large performance difference (over 2×) between sort-merge and hybrid hash join for our largest test case, a 9.7GB × 9.7GB join at 27% projectivity. For this query, we observe a long cpu-bound period with negligible IO at the end of the hybrid hash join that does not occur for any of the other hash joins. Hence, we conclude that this performance aberration is unrelated to the type of storage device, and may have arisen due to the methods we must employ to coax the optimizer to choose this join algorithm. In any event, we do not believe either of these outliers outweigh our broader conclusion that there is no particular need for the query optimizer to be SSD aware.

7. RELATED WORK

Previous work studying the applicability of Flash memory in DBMS applications has focused on characterizing Flash, benchmarking specific database operations on Flash, and designing new layouts, data structures, and algorithms for use with Flash.

Both Bouganim *et al.* and Chen *et al.* benchmark the performance of Flash for various IO access patterns [3, 4]. Bouganim introduces the uFLIP micro-benchmarks and tests their performance on several devices. Chen introduces another set of micro-benchmarks, concluding that poor random write performance poses a significant barrier to replacing conventional hard disks with Flash SSDs. While these micro-benchmarks are instructive for understanding database performance, we focus specifically on the performance of existing scan and join operators on SSD and disk. Others have also benchmarked Flash's performance within the context of DBMS systems. Lee *et al.* investigate the performance of specific database operations on Flash, including multiversion concurrency control (MVCC), external sort, and hashes [7]. Similar to our study, Do *et al.* benchmark ad hoc joins, testing the effects of buffer pool size and page size on performance for both disk and Flash [5]. Although related to our study, neither of these works look at the specific performance differences between disk and Flash for scans and joins and how this might impact query optimization.

Whereas the above works (and our own study) focus on measuring the performance of existing databases and devices, others look ahead to redesign DBMS systems in light of the characteristics of Flash. Yin *et al.* and Li *et al.* present new index structures, focusing on maintaining performance while using sequential writes to update the index [14, 8]. Baumann *et al.* investigate Flash's performance alongside a hybrid row-column store referred to as "Grouping" [1]. Similarly, Tsirogiannis *et al.* use a column store motivated by the PAX layout to create faster scans and joins [13].

Interestingly, our findings contradict recommendations of these last two studies. Baumann concludes that SSDs shift optimal query execution towards index-based query plans. The study bases this conclusion on the observation that asynchronous random reads on Flash are nearly as fast as sequential reads. Indeed, the arguments made by Baumann are a key component of the intuition we lay

out in Section 3 that led us to expect a need for SSD-aware query optimization. However, the conclusion neglects the observations discussed in Section 5 demonstrating that queries selecting more than a handful of tuples will likely retrieve the majority of pages in a relation, and thus gain no advantage from fast random IO. Tsirogiannis introduces a join algorithm that retrieves only the join columns, joins these values, and then retrieves projected rows via a temporary index. By our previous argument, scanning for projected data should retrieve the majority of data pages, preferring a relation scan, and thus provide comparable advantage on disk and SSD. We leave a comparative evaluation of their FlashJoin algorithm on SSD and disk to future work.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was partially supported by NSF grant CSR-0834403.

9. CONCLUSION

Flash-based solid state disks provide an exciting new high-performance alternative to disk drives for database applications. Our investigation of SSD-aware query optimization was motivated by a hope that the drastically improved random IO performance on SSDs would result in a large shift in optimal query plans relative to existing optimizations. At a minimum, we expected that constants capturing relative IO costs in the optimizer would require update. In this paper, we have presented evidence that refutes this expectation, instead showing that an SSD-oblivious query optimizer is unlikely to make significant errors in choosing access paths or join algorithms. Specifically, we demonstrate both empirically and analytically that the range of selectivities for which a scan operation can benefit from SSDs' fast random reads is so narrow that it is inconsequential in practice. Moreover, our measurements of alternative join algorithms reveal that their performance variability is far smaller on SSDs and is dwarfed by the 5× to 6× performance boost of shifting data to SSD. Overall, we conclude that the small and inconsistent performance gains available by making query optimizers SSD-aware are not worth the effort. Interestingly, we note that our conclusions may change with the advent of emerging non-volatile memory devices, such as phase change and spin-torque transfer memories, that offer finer (e.g., word-level) access granularity. These memories will allow tuple (rather than page) granularity accesses, and avoid the pitfalls of page oriented data stores demonstrated in Section 5. We intend to investigate the query optimization implications of these emerging devices in future work.

10. REFERENCES

[1] S. Baumann, G. de Nijs, M. Strobel, and K.-U. Sattler. Flashing databases: expectations and limitations. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, 2010.

[2] D. Bitton, D. J. Dewitt, and C. Turbyfill. Benchmarking database systems - a systematic approach. In *Proceedings of the Very Large Database Conference*, 1983.

[3] L. Bouganim, B. r Jnsson, and P. Bonnet. uFLIP: understanding flash IO patterns. In *Fourth Biennial Conference on Innovative Data Systems Research*, 2009.

[4] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, 2009.

[5] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, 2009.

[6] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB Journal*, 6(3), 1997.

[7] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[8] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, 2009.

[9] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 3rd edition, 2002.

[10] D. Roberts, T. Kgil, and T. Mudge. Integrating NAND flash devices onto servers. *Communications of the ACM*, 52, April 2009.

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1979.

[12] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), 1986.

[13] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, 2009.

[14] S. Yin, P. Pucheral, and X. Meng. A sequential indexing scheme for flash-based embedded systems. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009.

[15] P. C. Yue and C. K. Wong. Storage cost considerations in secondary index selection. *International Journal of Parallel Programming*, 4, 1975.