Computer Sciences Department

Revisiting Database Storage Optimizations on Flash

Mohit Saxena
Michael M. Swift

Technical Report #1671

March 2010

Revisiting Database Storage Optimizations on Flash

Mohit Saxena and Michael M. Swift
Department of Computer Sciences
University of Wisconsin-Madison
{msaxena,swift}@cs.wisc.edu

ABSTRACT

The database storage hierarchy has been heavily optimized for the performance characteristics of disks. Storage managers typically employ row- or column-oriented storage layouts, or a combination, to improve the I/O performance of different query workloads with disks. The recent rise of flash memory-based solid-state drives (SSDs) significantly change the performance characteristics of storage: these drives provide an order of magnitude lower read/access latencies, significantly higher read bandwidths, and most importantly, negligible seek overheads.

In light of these differences, we analyze major storage optimizations for read-optimized databases. We examine the benefits of row and column-oriented storage layouts on flash SSDs. Our measurements span through different workload variations, including selectivity, projectivity and concurrency that affect query processing on flash. Further, we also investigate the cost and benefits of a set of database optimizations, including data compression, prefetching, and indexes on flash SSDs. We back our experimental evaluation with analytical models of the performance tradeoffs of these optimizations.

Three of our key findings are: (1) SSDs scale up linearly with concurrent execution of database queries and outperform disks by up to a factor of two, (2) the low seek cost on SSDs makes column-storage a better choice for laying out data on a variety of flash devices, (3) and that while data compression is useful to further leverage the bandwidth of flash, database prefetching has less benefit for flash storage. Finally, we present a list of design implications of our findings on future database and operating systems for effectively embracing flash storage.

1. INTRODUCTION

> ***Tape is Dead, Disk is Tape, Flash is Disk, RAM locality is King.***
>
> – Jim Gray [15]

For decades, databases have been optimized for the performance characteristics of magnetic disks, such as their long access and seek latencies, and high sequential bandwidth [14, 26]. For example, databases often prefetch large buffers to amortize the cost of I/O over more data. However, solid-state disks (SSDs), built on flash memory, have recently achieved large capacity and high performance, making them a promising replacement for disks in many workloads.

SSDs represent a major advancement for storage management in database systems. To date, most uses of flash technology have focused on their high random-read throughput: a single mid-market device may provide 35,000 random I/O reads per second, while the fastest disks achieve barely 300. Thus, SSD usage in data management has been limited to the domain of transaction processing, where small random accesses are common.

However, there has been little investigation of the use of SSDs in decision support systems for analytical data processing. These workloads benefit from higher sequential bandwidths of SSDs, their small form factors and their low-power operation. A farm of slow, expensive and power-hungry disk arrays can be replaced with large SSDs optimized for selection, projection and scan queries used for business-intelligence applications and data warehouses. These applications deploy read-optimized databases for these workloads. In particular, such systems are tailored for read-only queries and are updated by bulk-loading with large database relations periodically [29, 18]. SSDs, with an order of magnitude faster access latencies and high bandwidths, are well suited for these applications when combined with a separate write-optimized staging area for periodic updates [27].

Most read-optimized databases employ several techniques that improve the performance with magnetic disks [18, 20]. In particular, database storage managers use:

- column-oriented storage layouts [13, 29] or a row/column combination [9, 16, 17] to reduce the cost of I/O to disks;
- compression to improve the effective bandwidth of disks at the cost of increased CPU overheads [6, 14, 32];
- database and file-system prefetching to amortize seek costs by reading ahead additional contiguous pages from disk; [28].
- reordering, scheduling and delaying I/O requests to minimize seeking between different datasets on disk [22].

In the light of widely different performance characteristics of SSDs, the cost and benefits of these optimizations may change as compared to disks.

In this paper, we revisit these storage optimizations on flash storage. We experiment with a high-performance database storage manager [1] and workloads based on the TPC-H specification [5] to isolate the performance impact of different database storage layouts for SSDs. Our experiments use densely packed pages that closely resemble the characteristics of commodity read-optimized databases [7, 10]. To provide generality, our measurements span a range of devices, from disks to low-end SSDs to high-performance SSDs, and workload variations.

With this study, we hope to inspire the database and OS research community to reconsider these optimizations originally designed for disks as many applications migrate to flash storage. Specifically, we address the following questions to unravel the different performance tradeoffs for data processing on SSDs through our experiments and analysis:

- What is the impact of different storage layouts on database query processing on flash storage? How do the performance tradeoffs for row and column stores differ for SSDs when compared with near-line and enterprise disk configurations?
- What is the impact of different query processing workloads, such as changed relation size, selectivity, or concurrency of different queries, on SSDs? Does workload affect performance differently on flash than on disk?
- What are the costs and benefits of optimizations such as data compression, storage indexing and database prefetching when used with SSDs?

The rest of the paper is structured as follows. We review the basics of flash memory and solid-state disks in Section 2, followed by a description of modern database storage hierarchy in Section 3. Section 4 describes our experimental methodology, discussing our query workloads, storage manager, measurement framework and storage devices used for experiments. Section 5 describes our findings and analytical models for performance tradeoffs of different storage optimizations. Section 6 presents a list of design implications on future database and operating systems for effectively embracing flash storage. Finally, we present related work in Section 7 and conclude in Section 8.

2. FLASH STORAGE BACKGROUND

As prices drop and write performance improves, non-volatile NAND flash memory has become a viable storage replacement for hard disks. Solid-state disks, built of multiple flash memory chips, commonly provide a drop-in replacement for hard disks to avoid the need for new device drivers. With additional mechanisms incorporated in the device firmware called the Flash Translation Layer (FTL), SSDs mask the differences between flash and disk storage technologies. However, SSDs differ from hard disks in three major ways relevant to data-analytics workloads: I/O performance, cost, and power consumption.

I/O Performance. SSD performance differs from disks both in transfer rate and seek time. Most importantly, flash media provides significantly lower random read latencies (0.1ms vs. 4-8ms for disks). In addition, a single SSD may internally contain many flash chips, allowing a RAID-like increase in I/O bandwidth within a single device. Thus, sequential read performance can be much higher than disks (250 MB/s for mid-range SSDs vs. 120 MB/s for the fastest disks). However, write performance for flash may be slower than disk, because blocks must first be erased. Better flash devices maintain a pool of clean blocks to absorb writes thus reducing the need to wait while erasing a block [8].

Device	Sequential (MB/s)		Random 4K-IO/s	
	Read	Write	Read	Write
HDD	80	70	120-300/s	
USB flash	11.7	4.3	150/s	20/s
SSD	250	170	35K/s	3.3K/s
PCI-e flash	700	600	102K/s	101K/s

Table 1: Disk and NAND flash memory performance: Hard disks exhibit a small variance in performance due to their mechanical nature. In contrast, flash memory devices present a wide range in performance due to different host interfaces and significant internal parallelism.

In contrast to disks, which present a small variance in performance due to their mechanical nature (seek times and rotation speed have a narrow range), flash storage devices exhibit a wide range of performance. Table 1 shows the performance of a variety of devices. Inexpensive and low-end devices such as USB flash sticks or camera memories offer moderate read bandwidth but have poor random-write performance. Solid-state disks (SSDs), with a standard SATA interface provide much better bandwidths, up to three times the fastest hard disks. This performance is mainly attributable to the device firmware, which implements intelligent block mapping schemes, parallel I/O accesses to multiple flash chips and write buffering [23]. High-end flash drives connected with the PCI-e interconnect interface and dedicated device drivers (rather than using the existing SATA drivers) are even faster [2]. Therefore, the variance in flash performance arises from two sources. First, an SSD can incorporate additional banks of flash chips, allowing more throughput through parallelism. Second, an SSD can incorporate smarter FTLs that are better able to conceal the costs of erasing flash before writing.

Cost. Until recently, flash memory was far more expensive than either disk or DRAM. The density of flash memory chips has doubled 14 times in the last 19 years, which is faster than the Moore's law for processors. This trend is expected to continue at least until a density of 32 GB/chip is achieved in the next few years [3]. Mid-range SSDs currently cost approximately $2.8/GB (quote of Intel X-25M SATA SSD, as of October 2009), which is 2–10 times expensive per byte than enterprise and near-line disks. This high cost arises from the manufacturing process of SSDs, which requires expensive wafer fabs. For workloads demanding high random I/O operations per second (IOPS), though, flash SSDs are about 50 times cheaper than a configuration of disks supporting the same number of IOPS. In addition, price-per-MBPS for sequential throughput is comparable to disks, as a single SSD can deliver nearly triple the bandwidth of a single disk.

Power. Unlike disks, flash does not have any mechanical or moving parts. Hence, flash devices consume significantly lower power while operating and almost zero power when idle. The typical power consumption for SSDs range between 0.15–2 W when active and as low as 0.06 W when idle [21]. In contrast, power consump-

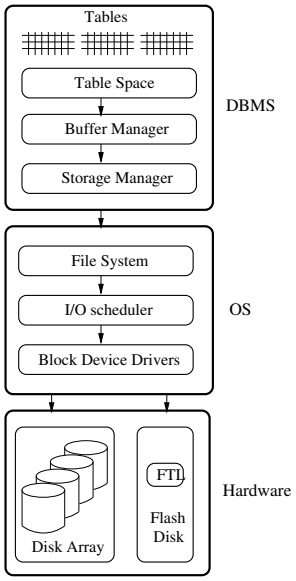


Figure 1: Database Storage Hierarchy: Buffer and storage managers employ different mechanisms to optimize for the performance of storage device and query workload. In addition, file systems and the operating system prefetch data and schedule block I/O requests submitted to the underlying device for amortizing disk seeks.

tion for SATA disks is between 13–18 W, or six to ten times greater than an SSD. At 10 cents per kilowatt-hour, the cost of a single disk for continuous three-year activity would be about $47, and almost $100 when including the cost of cooling and power distribution. In contrast, an SSD can be powered for just $10. The power savings increase further for large disk arrays with expensive controllers that provide comparable random performance. Thus, SSDs tend to be price-competitive with disks when considering the complete cost of both device and power.

3. DATABASE STORAGE MANAGEMENT

Disk access can be a dominant cost for databases workloads, so database management systems carefully manage all I/O. DBMS storage managers also account for the storage device performance characteristics, such as seek time, access latency, and sequential bandwidth. Thus, databases lay out data and optimize their access patterns to minimize the cost of I/O.

Figure 1 shows a typical database storage hierarchy on Linux (sophisticated and special-purpose database systems may differ). Multiple database relations and storage indexes are clustered together in logical table spaces that are laid out as files on disks. The buffer manager maintains a pool of memory buffers to cache data in memory. Lower down the stack, the storage manager is responsible for most of the I/O to the underlying storage device, deciding which blocks to retrieve and when. The storage manager may directly access the device or may use the file system to perform I/O on its behalf.

Data Layout. One of the major focuses of this paper is studying the impact of the storage manager's data layout. The two major layout organizations are row-oriented, in which all rows of a database relation are stored contiguously in a single file, or column-oriented, in which attribute values of each column in the relation are stored contiguously in a separate column file, also called a *chunk*. Row stores are more effective if the entire row is read, while column stores improve performance if only a small number of attributes are projected from each row. Unlike row stores, column stores tend to read fewer bytes by (i) seeking between files corresponding to the columns projected in the query, and (ii) seeking between attribute values in a column file for which the corresponding row has been selected by the query predicate. Other hybrid storage layouts, such as PAX [9], DMG [17] and column abstractions [16], mix row- and column-oriented storage.

Compression. Compression can improve query performance by trading CPU processing for more effective use of disk and memory bandwidth. Column stores enable compression by storing all values of an attribute together, for example by replacing data values with indexes into a dictionary. Therefore, database administrators frequently use different compression schemes during the physical design phase of database schemas to optimize for both performance and storage space [6, 14, 32].

Prefetching. Database storage managers prefetch data that is not needed immediately. Prefetching for disks provides two benefits. First, reading more data at a time amortizes the high random seek latencies for disks over larger sequential requests. Second, prefetching overlaps I/O with computation, so that data is already available in memory when it is finally requested [28]. Storage managers in modern database systems, for example SQL Server Enterprise, prefetch up to 1,024 8 KB pages.

OS and Device Optimizations. Within the OS, the I/O scheduler merges, reorders and delays requests to optimize the performance on the underlying storage device (presumably disks or disk arrays). The device provides the final layer of I/O scheduling. For disks, the controller may again reorder or buffer requests to improve performance based on the current location of the disk head. For SSDs, scheduling occurs in the FTL, which improves performance by remapping logical block addresses to physical flash addresses.

In summary, the database storage hierarchy embeds different disk-oriented optimizations at various levels. Both storage managers and file systems employ data prefetching to reduce access latencies, and optimize data layout to reduce the number of seeks. Finally, the disk scheduler and device drivers both reorder operations to minimize seeks as well. This paper revisits the cost and benefits of these disk-oriented optimizations for flash storage.

4. EXPERIMENTAL SETUP

The I/O performance of query processing in database systems is affected by the query workload, storage management and the characteristics of the storage device. This section presents our experimental methodology to investigate the impact of each of them for flash storage.

We focus our study on queries commonly used for large-scale data analysis. To model this workload, we use different select, project and scan queries based on the TPC-H workload specifications [5]. To isolate the impact of various storage manager optimizations, we use a high-performance query engine [18] that implements both row and column-oriented storage layouts. We measure the performance of query processing on a variety of storage devices with different performance characteristics. All our experiments are re-

peated multiple times, and we report the average over ten executions.

4.1 Query Workload

We focus our study on data analysis queries used for mining large data repositories in data warehouses and business intelligence applications. This workload mainly consists of selection, projection and scans over large relations, but few updates. Thus, these workloads are generally run on read-optimized databases that minimize the number of bytes read from disk for processing a given query. Database relations are periodically updated in bulk from a separate write-optimized staging area, where new data is aggregated. This workload forms the basis of TPC-H [5].

Flash storage provides ample opportunities to optimize the performance of such queries because of its high read bandwidth and low random access latency. Hence, we study the performance of different variants of the following select, project and scan queries:

select T.a_1, T.a_2, T.a_3 ... **from** T
where Predicate P(T.a_1)

In this query, T represents the database relation; a_1, a_2, a_3 are different attributes and P is a sargable predicate on the first attribute. To isolate the effects of different storage layouts, we do not use storage indexes to accelerate queries unless otherwise noted. We change the projectivity of the query by varying the number of attributes projected in the select phrase of the query. Similarly, we change the selectivity factor from 0.1% to 100% (low selectivity implies less qualified tuples) by modifying the predicate P. The number of columns projected and the number of rows selected in a query have a direct influence on its execution time.

Our experiments use two tables LINEITEM and ORDERS, that are based on TPC-H benchmark specification. We choose these tables to isolate the effects of tuple sizes and to ensure direct comparison of our results with earlier studies [18]. We use the official TPC-H toolkit [5] to populate these tables with data values for different attributes. For our experiments, LINEITEM represents a wide relation of 16 attributes of 150 bytes per tuple. ORDERS has a tuple width of 32 bytes and contains 7 attributes per tuple. To ensure a fair comparison between the two tables, we scale them to have the same number of rows: scaling LINEITEM by 10x and ORDERS by 40x ensures that both relations have 60 million tuples. LINEITEM takes over 9GB of disk space and ORDERS takes over 2GB. For most of our experiments, the size of these relations is sufficient enough to analyze the steady state I/O performance of different storage devices.

4.2 Data Manager

Query Engine. We focus on comparing the performance tradeoffs of row and column stores for flash devices. In order to isolate the impact of storage layout, we use the query engine implemented by Harizopoulos et al. [18], which is available online [1]. While some commodity and research database systems implement column stores, such as C-Store [29] and MonetDB [10], they provide extensive performance optimizations for query processing such as in-memory database kernels built on virtual memory primitives, multi-threaded parallelism and vector storage for columns. These optimizations tend to blur the fundamental impact of row and column stores for flash devices, which is the focus of our experiments. The query engine used can operate on both row and column-oriented data. It has been used in previous published work [18, 20], and thus ensures a direct comparison of row and columns stores for flash devices. Furthermore, the query engine uses zero-copy direct I/O, and transfers data directly from the storage device to user-space buffers without an explicit buffer pool.

Scanners. The query engine pre-compiles the queries and pipelines their execution for operating on the output blocks. Scanners reconstruct the tuples, apply predicates, extract the projected attributes and combine them for materialization later. Both the row and column stores use densely packed pages on disk. The scanners for row stores read data pages from disks into an I/O scan buffer and then decode the columns from each page. Column store scanners, in contrast, read multiple files (chunks) from disk corresponding to the columns projected until the output tuple buffer is full. Each projected column is examined only at positions where the predicate was satisfied by the scan of the preceding column. This reduces disk I/O at the cost of additional seeking within a column.

Application Parameters. We tune the configuration parameters of the query engine for high performance. The major parameters we tune are: I/O depth (prefetching distance), I/O unit (scan buffer) size, page size, and block (materialized tuple buffer) size. We find that the most significant parameters are I/O depth and I/O unit size. We use an I/O unit of 128 KB and an I/O depth (prefetch read-ahead distance) of 6 MB (48 I/O units) unless otherwise specified. In addition to these application-level parameters, our experiments require careful configuration of operating system and storage device parameters, which we discuss in Section 4.3 and Section 4.4 respectively.

Data Compression. Compression can improve scan performance by trading CPU processing for more effective use of disk bandwidth. Flash devices have higher bandwidths and thus may benefit from compression in a different manner than disks. We use three different compression schemes for our experiments - bit-packing, dictionary and FOR-delta. Bit-packing stores each attribute using only the minimum number of bits in the maximum value of its domain. Dictionary-based compression uses an array with all distinct values of the attribute and stores each attribute as an index number to that array (similar to a hash lookup). FOR-delta (Frame-Of-Reference) uses a base value per page and stores deltas for attributes with it (see [6, 32, 14] for more details on these compression schemes). The performance differences of the compression schemes have been studied earlier [6], so we only present results for the best mechanism.

Database Indexes. Storage indexes improve the execution time for processing a query by directly seeking to the selected row. As our optimized storage manager does not support indexes, we instead use PostgreSQL 8.3 [4] with the same tables (LINEITEM and ORDERS) for these experiments. We use bitmap indexes to investigate the impact of SSDs for storing database indexes. We vary the number of attributes projected in each query from one to all. The PostgreSQL query planner selects the primary and secondary indexes for the columns used in query predicates. We also investigate the impact of multiple indexes by using additional AND predicates. Such queries have the following format:

select T.a_1, T.a_2, T.a_3 ... **from** T
where Predicate P_1(T.a_1) AND P_2(T.a_2)

Device	Sequential (MB/s)		Random 4K-IO/s		Latency
	Read	Write	Read	Write	ms
Disk	80	68	120-300/s		4-5
SSD-Fast	250	70	35K/s	3.3K/s	0.1
SSD-Medium	69	20	7K/s	66/s	0.2
SSD-Slow	25	20	6K/s	136/s	0.6

Table 2: Performance characteristics of storage devices used: SSD-Fast, SSD-Medium and SSD-Slow represent different price points and performance. SSDs substantially outperform disks for random read IOPS.

In this query, P_1 and P_2 are sargable predicates casted over the indexed attributes a_1 and a_2. Finally, we study the effect of index utilization by varying the selectivity of the query.

4.3 Measurement Platform

Platform and Measurement Tools: We perform all measurements on a 2.5GHz Intel Core 2 Quad system configured with 4GB DRAM and 3MB L2 cache per core, running Ubuntu 8.0.4 (Linux kernel 2.6.24). We verify our results for the elapsed time for query execution using both performance counters and the Posix *time* utility. Furthermore, we instrument the Linux kernel with the Linux *blktrace* mechanism to intercept and trace the I/O requests at the block layer in the operating system. These traces, along with the Linux *iostat* utility, enable us to monitor disk activity, such as the number of seeks performed during a query. We use the ext2 file system for both disk and flash devices. While the journaling ext3 file system is more commonly used in practice, its read performance is identical to ext2 but the journal requires extra updates to ensure consistency after a crash.

4.4 Storage Device Characteristics

There is a high variance in the performance of flash SSDs, with performance roughly corresponding to price. We therefore use three flash devices (solid-state disks) fabricated by three major SSD manufacturers at different price points. Among the three, two use a SATA 2.0 interface and the third uses a SATA 1.0 interface. Since our intention is not to compare the performance of these competing SSDs, we refer to them as SSD-Fast, SSD-Medium and SSD-Slow from faster to slower devices. SSD-Fast is a relatively high-end device with SSD-Medium and SSD-Slow are intermediate and low-end devices. Table 2 presents the measured performance for these devices, which differs from advertised data-sheet values. Sequential read bandwidth, random read I/O operations per second and seek latency of the three devices are most relevant for our experiments. We use a Seagate Barracuda 7200 RPM disk, which uses a SATA 2.0 interface and simply refer to it as disk. Scan workloads, with largely sequential access, are dependent on the I/O bandwidth of the system, which RAID striping can improve. Therefore, we also investigate the impact of using a software RAID-0 disk array for our measurements.

Flash devices often require tuning to attain optimal performance. We enable Native Command Queuing (NCQ) for SSD-Fast and configure the system BIOS to treat SATA devices in native, rather than compatible mode, to boost its performance. The device I/O queue depth is configured to 32 for both SSD-Fast and disk to ensure a fair comparison. Finally, we enable on-disk prefetching for all devices unless otherwise mentioned.

5. PERFORMANCE STUDY

Database management systems use different storage layouts and other mechanisms such as data compression, prefetching and storage indexing to optimize the performance of different query workloads. We focus our measurement study on three questions surrounding these components:

- What is the impact of different *storage technologies* and *layouts* on the performance of database query processing?
- How does the performance of flash database storage vary across different *query workloads*?
- What are the costs and benefits of different disk-oriented *storage optimizations* for flash storage?

We experiment with different disk, SSD and device configurations and answer the first question in Section 5.1. Next, we investigate a variety of query workloads to measure the performance of flash database storage in Section 5.2. Finally, we evaluate the cost and benefits of different storage optimizations such as database prefetching, data compression and storage indexing in Section 5.3.

5.1 Database Storage Layouts

In this section, we experimentally evaluate and analytically model the performance of different database storage layouts across a range of devices.

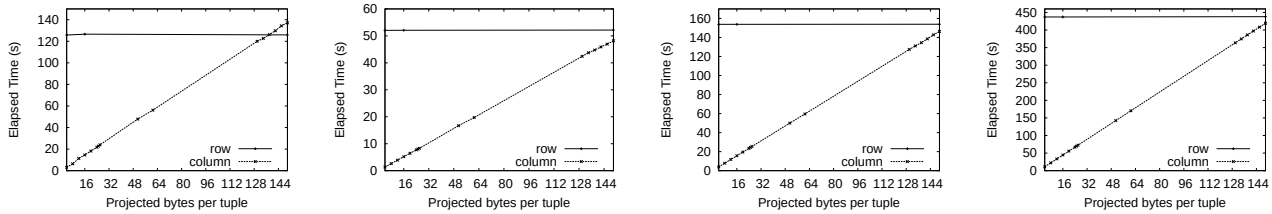
5.1.1 How do the performance tradeoffs for row and column stores differ for flash as compared to disks?

To answer this question, we compare the performance of row stores with column stores on the high-end flash SSD-Fast with disk. We measure scan performance with select queries for a selectivity factor of 10%. Column-store effectiveness increases when only a few attributes are retrieved, so fewer bytes are read from the device. Thus, we vary the number of attributes projected per tuple for our experiments. We use the LINEITEM table with a tuple width of 150 bytes and 16 attributes.

Figure 2(a) and 2(b) show the time taken to complete the select query using row and column store layouts on disk and flash SSD-Fast with different projectivity factors. Both row and column store layouts for SSD-Fast outperform disk. Row stores for SSD-Fast are twice as fast as for the disk, which reflects the difference in sequential read bandwidths of the two devices; row stores tend to saturate the I/O capacity of the device.

For disk, column store performance degrades quickly as the number of projected columns increases from 1 to 16 because of an increase in both I/O and CPU overheads for processing more columns. The I/O wait time increases by a factor of fifty when increasing the number of bytes projected per tuple from 4 to 150, and total CPU time increases by a factor of five. Thus, for disks, we observe a crossover point when 90% of the tuple is projected, at which point row stores become more efficient than column stores.

For a deeper understanding of why column store performance degrades on disk with projectivity, we instrument the Linux kernel with *blktrace* and trace each I/O request submitted to the device driver. We identify as *seeks* all requests that are at least 63 disk sectors apart from the previous request. At 100% projectivity, 867 seeks occur for column stores – almost 10 times greater than for row stores. At 10% projectivity, column stores issue only 83 seeks



(a) Disk (b) SSD-Fast (c) SSD-Medium (d) SSD-Slow

Figure 2: Performance of row and column stores on flash devices and disk. Flash devices at different price points exhibit different performance. However, in contrast to disks, column stores always outperform row stores for all flash devices at any projectivity and fixed selectivity of 10%. The y-axis scale is different for all devices.

– roughly the same number for row stores. This suggests that as projectivity increases, column stores spend more time seeking, both across columns and within columns to skip the attributes that do not satisfy the predicate. Thus, it results in a crossover point where column stores perform worse with disks.

The shape of the performance curves for column stores on flash is similar to that for disk. However, column stores *always* perform better than row stores for SSD-Fast. SSD-Fast provides much lower seek latency (0.1ms vs. 4ms) than disks, which prevents a crossover between the column- and row-store curves even at high projectivities. For disks, the penalty for seeks at 100% projectivity takes at least 3.5 seconds, while they take less than 0.1 seconds for SSD-Fast.

We now present a simple analytical model of the performance of the two storage layouts that explains the crossover point. We assume that select query workloads are I/O bound with negligible CPU overhead because there is a significant overlap between the CPU and I/O times. Let *R* be the size of the relation in megabytes and *B* the bandwidth of the storage device in megabytes/s. For a row store layout, the query completion time is given as the ratio of the two quantities:

$$t_r = R/B \tag{1}$$

For a column store layout, query execution time is also affected due to seeking between different columns. Let *k* be the number of attributes projected, *C* be the average size of each chunk (column file), and *l* be the seek latency of the device.

$$t_c = \alpha \cdot k \cdot (C/B) + \beta \cdot k \cdot l \tag{2}$$

In Equation 2, the elapsed time for column stores has two components: the time to read columns at full sequential bandwidth and the time to seek between columns. α and β adjust for imperfect I/O behavior. α reflects that not all columns are of equal width and that not all columns are read at the full sequential bandwidth. This is because ext2 and most other file systems do not lay out data in a perfectly sequential manner. Hence, α is the reduction in the sustained transfer bandwidth. Since column stores result in seeking both across and within columns to skip attribute values that do not satisfy the predicate, the term β adjusts the number of seeks; in most cases β is greater than one.

The crossover point occurs when the performance of row stores equals that of column stores as modeled by Equation 1 and 2 respectively. We derive the following formulation for *k*, the number of columns projected at crossover:

$$k \approx \frac{R}{\alpha \cdot C + \beta \cdot B \cdot l} \tag{3}$$

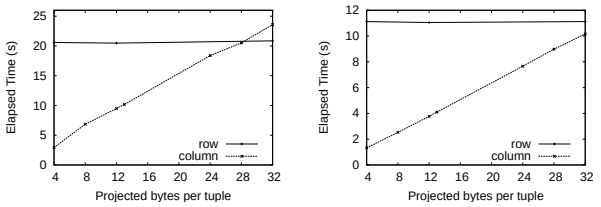
For a device with no seek cost, the crossover point never occurs and *k* approximates the total number of attributes in the relation, which equals $\frac{R}{C}$. For devices with higher seek cost *l*, the crossover occurs when the bandwidth-latency product, $\beta \cdot B \cdot l$ becomes a substantial fraction of the column size *C* in the denominator.

For disk, *l* can be as high as 4–5 milliseconds and thus we find that a crossover occurs. A single seek for column stores on a disk with sequential read bandwidth of 80 MB/s results in 0.4 MB less data read than for a purely sequential workload. The bandwidth-latency product in the denominator of Equation 3 accounts for this loss. For disks, the 867 seeks incurred at 100% projectivity reduces the effective bandwidth of the disk by 347 MB over the workload. In comparison, the average column size for LINEITEM relation of 9 GB size with 16 attributes is about 562 MB. In contrast, for flash SSD-Fast with negligible seek overheads, the contribution of seek latency is 22 MB, or less than 4 percent of a column. This results in a crossover point *k* equal to the total number of attributes in the tuple, and explaining why columns stores always outperform row stores.

5.1.2 Do the performance tradeoffs differ across device models and disk configurations?

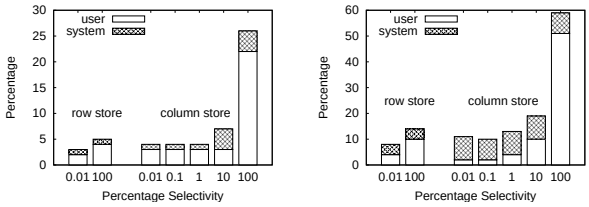
Flash devices at different price points provide widely varying performance due to different internal levels of parallelism (as in RAID for disks) and sophistication of write-buffering algorithms [8]. Thus, the performance on a high-end SSD may not carry through to cheaper devices. We repeat our experiments on an intermediate flash SSD-Medium and a low-end flash SSD-Slow. Figure 2(c) and 2(d) shows row and column store performance for these two devices respectively.

We observe two important features. First, regardless of the device performance characteristics, column stores always perform better than row stores for all flash devices; there is no crossover between the two storage layouts. This again conforms with our formulation for predicting crossover for flash devices since the bandwidth-latency product for both SSD-Medium and SSD-Slow is half that



(a) Disk (b) SSD-Fast

Figure 3: Performance of row and column stores on disk and SSD-Fast for narrow tuples: Reduction in tuple size further shifts the crossover point to left for disk. In contrast, there is no crossover for SSD-Fast for narrow tuples. The y-axis scale is different for both devices.



(a) Disk (b) SSD-Fast

Figure 4: Impact of selectivity on breakdown of CPU time for query execution on disk and SSD-Fast: Selectivity does not affect the performance of row stores. An increase in selectivity also increases the CPU overhead, thus reducing its overlap with I/O wait time for both disk and SSD-Fast. However, this CPU overhead is a larger fraction of the total execution time for column stores on flash SSD-Fast than disk. The y-axis scale is different for both devices.

of SSD-Fast and much smaller than a column.

Second, the performance of row and column stores for SSD-Medium is comparable to that of disk. This is because SSD-Medium provides lower read bandwidth than disk (69MB/s vs. 80MB/s), which is compensated by its order of magnitude better seek latency (0.2ms vs. 4ms) to some extent. On the other hand, SSD-Slow always performs worse than disk for both row and column stores because of its considerably lower sustained read bandwidth (25MB/s) and relatively higher seek latency (0.6ms).

While flash devices exhibit internal parallelism, disks can be configured in RAID arrays to provide device level parallelism and improved performance. We construct a RAID-0 software disk array with two SATA disks, capable of delivering up to 160 MB/s bandwidth (80 MB/s per disk). However, we found that while bandwidth improved, seek latencies did not. As a result, the performance of column store layouts is similar to that of a single disk. While the RAID-0 configuration provides throughput similar to SSD-Fast, its seek time is still much higher and we again observe a crossover for the two layouts when 90% of the tuple is projected.

5.2 Query Workloads

Database workloads, in terms of the query selectivity, width of data, and concurrency of access, can also accentuate differences between flash and disk storage.

5.2.1 Does the width of tuples affect this tradeoff?

For disks, prior work has shown that row stores perform better with narrower tables because narrow tuples can be packed tightly in a read-optimized page and thus can be scanned much faster [18, 20]. The width of tuples also changes the number of bytes retrieved for each tuple. Furthermore, tuple size also affects the seek components in the denominator of Equation 2 due to a reduction in the size of columns.

To investigate the impact of tuple width, we repeat our experiments with the ORDERS table, which has only 7 attributes per tuple with a total size of 32 bytes (in comparison, LINEITEM has 16 attributes in 150 bytes). Figure 3 plots the performance of row and column stores against projectivity on SSD-Fast and disk. We do not show the results for SSD-Medium and SSD-Slow for brevity since they were similar to SSD-Fast but scaled to their lower sequential bandwidths (as access latencies are similar).

For flash device SSD-Fast, the row and column store performance is similar to the LINEITEM table, and column stores still outperform row stores. Thus, column stores provide high performance on flash regardless of tuple width.

For disk, however, we observe that the crossover point where row stores perform better occurs earlier for narrow tuples. Column stores perform worse than row stores when projecting more than 75% of the tuple. As compared to LINEITEM in Figure 2(a) with crossover point at 90% projectivity, it shifts left for ORDERS because row stores perform less I/O per tuple, so the opportunity to improve performance with column stores is lower. As each column is smaller, the seeks between columns have proportionally more impact on performance: column stores seek about 182 times and row stores only 19 times at 100% projectivity for ORDERS.

5.2.2 Does selectivity of the query affects this tradeoff?

The selectivity of a query decides the number of tuples read while scanning that are discarded because they do not match the predicate. We investigate its impact by varying the selectivity of our queries from 0.1% to 100% on a log scale for both relations. To highlight the difference between row and column stores, Figure 4 shows the fraction of user and system times for queries that yield variable selectivities at 100% projectivity. These queries are executed on LINEITEM table stored on disk and flash SSD-Fast. Row stores iterate through all the tuples in the relation regardless of selectivity. So there is little change in the CPU overhead for row stores on both disk and SSD-Fast.

However, we observe that with increased selectivity there is an increase in CPU overhead for column stores on both disk and SSD-Fast. As we vary selectivity for column stores, the total CPU utilization increases up to 59% at 100% selectivity for SSD-Fast. It remains constant between 5–8% for row stores regardless of selectivity. The increase in CPU time for column stores is mainly attributed to the extra work done by each scan node of the query engine for processing extra tuples. In contrast, the percentage of CPU overhead for column stores only increases up to 26% on disk, which is less than half that on SSD-Fast. Therefore, column stores are still I/O bound on disk, but become more CPU bound on SSD-Fast with an increase in selectivity. Hence, there is less overlap between CPU and I/O with flash SSD-Fast than with disk.

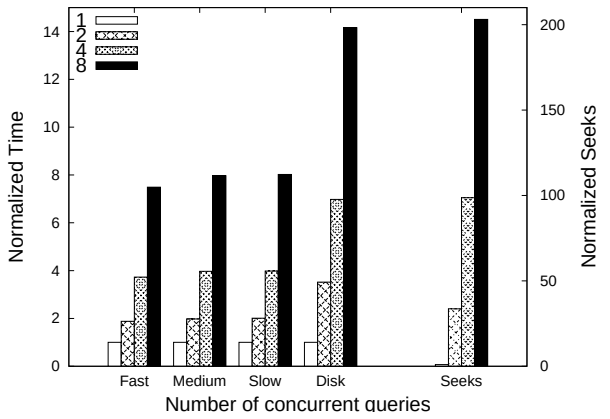


Figure 5: Execution time and number of seeks for concurrent queries relative to a single query: Flash devices scale linearly with an increase in the number of concurrent queries. In contrast, for eight concurrent queries, disk performance is up to fourteen times worse than a single query. Furthermore, the number of seeks also increase non-linearly, thus degenerating the workload to a large extent. The left y-axis shows the query execution time and the right y-axis shows the number of seeks, both relative to single query.

Nevertheless, we do not find any crossover for the total query execution time on SSD-Fast even at 100% selectivity because the increased CPU time still overlaps with I/O wait time. Similar results are obtained for the two other slower flash devices. However, with faster devices, such as the Fusion-IO ioXtreme PCI-e SSD that provides up to 600 MB/s sequential read bandwidth [2], there will be less overlap between the I/O and CPU times, and the increase in CPU utilization for column stores may tip performance to favor row stores at high selectivity.

Our results reiterate recent work by Tsirogiannis et al. [31]. Their work shows that using PAX architecture [9], a hybrid of row and column stores, does not improve query execution time for variable selectivity factors unless additional optimizations are used. For example, reading only mini-pages that correspond to attributes used in the selection predicate and using fully or partially sorted attributes are necessary to leverage selectivity for performance. However, such optimizations only improve the performance of PAX layout for low selectivity queries. We discuss this more in Section 7.

5.2.3 How do concurrent queries scale with flash and disk?

Database systems may perform poorly with concurrent queries that cause competing disk traffic [24]. Such competing traffic can turn multiple sequential workloads into a collectively seek-bound workload that performs poorly on disks. The database, file system, and I/O scheduler of the operating system may try to minimize seeks by clustering nearby I/O requests. In some cases, concurrent scan queries to the same relation can be optimized by sharing the same scanner [19], so we analyze scans of different relations. We measure the performance with a single row or column store select query on an instance of ORDERS table, while competing against a variable number of concurrent row store select queries on an instance of LINEITEM table.

Figure 5 plots the query execution time of concurrent row store select queries normalized to one individual query for all three flash devices and disk, as we increase the degree of concurrency. Against the right y-axis, we show the number of seeks. Column store performance is similar, so we do not include its results.

With a single query, performance is equal to that of row stores in Figure 3 and normalized to one in Figure 5 for the different devices. Furtermore, there are few seeks for single query. However, when the number of concurrent scans increases to two, the number of seeks shoot up quickly and disk performs worse than both SSD-Fast and SSD-Medium, despite the 6 megabytes read-ahead that tries to amortize the cost of seeks. With an SSD, each individual query takes twice as long to complete, while with disk, each query takes 4 times longer. As the degree of concurrency increases, execution time increases linearly with the number of concurrent queries for all SSDs. This demonstrates that bandwidth is the most significant factor for concurrent queries with SSDs, as the seeks incurred have little impact on performance.

For disk, though, execution time increases twice as fast, and 8 concurrent queries perform 14 times slower than for a single query. Unlike SSDs, seek times dominate performance for disk at high concurrency. Thus, performance for disk would be much better if the two queries were run sequentially rather than concurrently as the system is unable to effectively schedule the two competing I/O streams to achieve maximum performance.

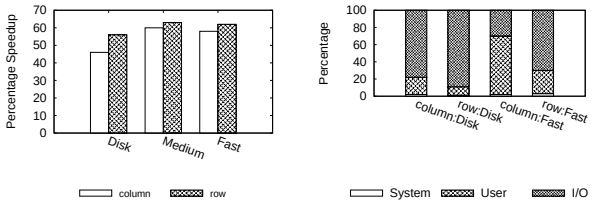
We also measure the impact of the operating system with two different I/O schedulers: CFQ and NOOP. CFQ batches up all the asynchronous requests from different processes in a number of queues with different I/O priorities. As CFQ seeks fairness between queues, the scheduler trades off seek time between successive requests submitted to the device driver for more equal performance. In contrast, NOOP inserts all requests in a single FIFO queue and submits them as soon as possible.

We observe marginal differences in the performance of flash devices with the two schedulers. The performance difference between NOOP and CFQ is less than 3 percent, indicating that scheduling is less necessary for SSDs. For disks, NOOP performs 13% better than CFQ. This suggests that the workload submitted to the device has degenerated to a large extent and ensuring fairness between different query executions further degrades the performance. This is also visible in Figure 5 as the steep rise in the number of seeks. They rise by a 33 times for two competing queries, and by over 200 for eight queries. For the 2 GB scan, these seeks lead to an average request length of 2.2 MB, which may be larger than the window of requests the CFQ scheduler considers for reordering. Thus, it breaks large sequential reads to achieve better fairness, which increases the seek overhead.

These results suggest that achieving fairness for the different concurrent queries at the block layer may prove difficult with disks, and demands careful planning within the database or application. In contrast, flash devices require little scheduling to achieve high performance, and thus naturally perform well with concurrent queries.

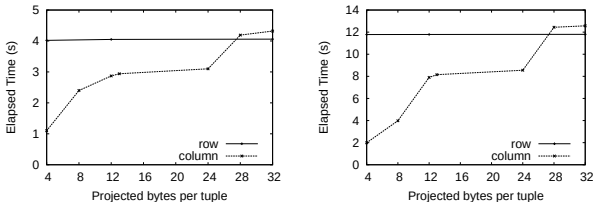
5.3 Database Storage Optimizations

In addition to the customized storage layouts, database storage managers implement numerous optimizations to improve performance by hiding or reducing the cost of disk accesses. We next investigate the impact of three such optimizations on flash storage: compres-



(a) Compression Speedup (b) Compression Cost

Figure 6: Performance of row and column stores on disk, SSD-Fast and SSD-Medium with data compression: Compression benefits SSDs to a larger extent than disk. However, compression costs extra CPU time for column stores and makes them CPU bound on flash devices. The y-axis scale is different for both devices.



(a) SSD-Fast (b) SSD-Medium

Figure 7: Impact of compression on performance of column stores on flash devices: Compression increases the CPU overhead for column stores at high projectivity resulting in low overlap with I/O wait time. Thus, it causes a crossover between row and column store performance for both flash devices.

sion, prefetching, and indexing.

5.3.1 *How does database compression perform on flash storage?*

As we have shown, performance of select and project queries is constrained by I/O bandwidth. Thus, compression offers an opportunity to improve I/O performance at the cost of additional CPU usage to compress and uncompress the relations. We use FOR compression on the ORDERS table, which compresses its 32 byte tuples down to 12 bytes.

Figure 6(a) shows the speedup in performance of row and column stores for different storage devices at 100% projectivity. We make two major observations about the effects of compression. First, compression greatly benefits both disks and SSDs but by different amounts. Compression reduces the query time for disk by 56% with a row store and 46% for a column store. However, compression improves the performance of both SSD-Slow and SSD-Medium by 63% for row stores and 59% for column stores. The benefit of compression comes from increasing the effective I/O bandwidth while leaving seek latencies unchanged. Thus, for SSDs, where seek latency is negligible, compression offers greater benefits for reducing total I/O time.

Second, row stores benefit from compression more than column stores at 100% projectivity. This becomes more clear when we observe the performance of the two layouts as we vary the number of attributes projected and observe the execution time breakdown of the workload. We plot the query execution time for flash SSD-Fast and SSD-Medium in Figure 7. Disk is still I/O bound, thus we omit its results. As we increase projectivity, the CPU cost for re-assembling tuples from columns grows even higher with compression for column stores. Therefore, the benefit of increased effective bandwidth is reduced by the extra CPU time spent generating output data. Furthermore, with compression the system cannot overlap CPU utilization with I/O as effectively for column stores. Figure 6(b) shows the breakdown of the elapsed time at 100% projectivity for compressed row and column stores on disk and SSD-Fast. In this figure, I/O time represents the fraction of the elapsed time that is neither user nor system time. Similar results are observed on flash SSD-Medium and SSD-Slow. The salient feature of this figure is the time spent in usermode, which represents the time to uncompress data. Row store layouts require less processing, and hence are better able to overlap CPU usage with I/O. The column store, in contrast, spends more of its time reconstructing tuples and hence does not keep the device busy, leading to longer execution.

We now analyze the performance of row and column stores with compression by extending our original model described in Equation 3. For simplicity, we redefine R and C as the average size of the relation and a chunk (column file) after compression. As shown in Figure 7, row store performance is independent of the number of projected columns and is only dominated by I/O bandwidth, thus we reuse Equation 1 for its query execution time.

However, columns stores are less able to overlap the extra CPU time to uncompress data with I/O because of their less-regular I/O patterns. Therefore, the CPU cost of compression is proportional to $k \cdot C$, the product of the number of columns projected and the average size of a chunk that is uncompressed. We multiply this product by γ to adjust for the differences in compression speeds and ratios for different schemes and attribute values. For flash devices with negligible seek latencies, we rewrite Equation 2 with this additional CPU overhead to compute the query execution time for column stores as follows:

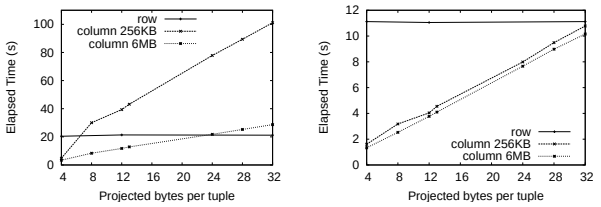
$$t_c = max\{\alpha \cdot k \cdot (C/B), \gamma \cdot k \cdot C\} \quad (4)$$

Thus, when bandwidth is low, the first term dominates and the workload is I/O bound. However, when bandwidth is higher, the workload can become CPU bound with the second term. We calculate that for our workloads, the query saturates the CPU with a device bandwidth of 380 MB/s. Thus, for higher-bandwidth devices, row stores will outperform column stores with compression.

In summary, compression benefits flash devices and disks by reducing the amount of data read. However, flash devices benefit more, because bandwidth is a greater part of the I/O cost. In addition, the additional processing to reconstruct tuples can push the performance of column stores below row stores at high projectivities. This shows that although compression increases effective bandwidth for flash devices, it can still tradeoff for column store performance because of its high CPU costs.

5.3.2 *Does prefetching benefit flash like disks?*

Database storage managers prefetch data that is not needed immediately. Prefetching for disks provides two benefits. First, reading more data at a time amortizes the high random seek latencies over larger sequential requests. Second, prefetching overlaps I/O with



(a) Disk (b) SSD-Fast

Figure 8: Database Prefetching: For disks, a high read-ahead is beneficial for amortizing seeks and improving the query execution time. In contrast, prefetching has almost no impact on the performance of column stores on flash devices. The y-axis scale is different for both devices.

computation, so that data is already available in memory when it is finally requested [28].

We measure the benefits of prefetching for the performance of row and column store layout by scanning ORDERS on the four devices. Figure 8 shows the query execution time with read-aheads of 256 KB (2 I/O units) and 6 MB (48 I/O units). We also measure the number of seeks for each data point. We observe that prefetching does not improve row store performance, because there are few seeks to be amortized. In addition, the number of seeks for row stores is fairly constant regardless of projectivity. Hence, we only show a single row-store curve for each of the two devices.

However, as shown earlier in Section 5.1.1, column stores incur more seeks as projectivity increases. Therefore, column stores benefit differently for the two devices with prefetching. For disk, as we increase the prefetch read-ahead from 256 KB to 6MB, there is a dramatic decrease in the query execution time for column stores. This is because the number of times column stores seek decreases with an increase in the read-ahead: at 100% projectivity, the 6 MB read-ahead causes 182 seeks, while for a 256 KB read-ahead, it shoots up to 2941. Thus, prefetching compensates for disk performance by reducing the number of seeks.

However, for flash devices the cost for random access is a small component of the access time. This is reflected in the curves for column store performance for flash SSD-Fast. There is little change in the performance of SSD-Fast as the prefetch read-ahead is reduced. Similar results occur with SSD-Medium and SSD-Slow.

In summary, storage optimizations that compensate for slow seeks, such as prefetching, are no longer required for flash devices, while those that improve effective bandwidth, such as compression, are still useful.

5.3.3 How do storage indexes perform on flash?

Storage indexes accelerate database query processing by directly seeking to the selected rows instead of scanning all rows. As our optimized storage manager does not support indexes, we instead use PostgreSQL for these tests, which uses a row-store layout, and configure it to create bitmap indexes over different attributes.

Use of an index can change performance for two reasons. First, the database must perform I/O to read in the index. Second, with an index the database can seek between the selected tuples in the data tables rather than scanning.

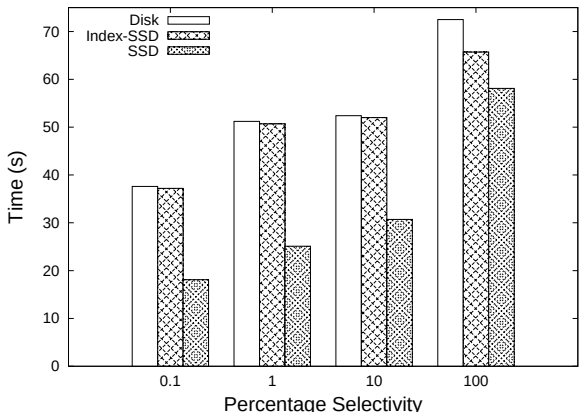


Figure 9: Impact of indexing on performance of query processing for variable selectivities on PostgreSQL with different devices used for storing index and relation data.

We investigate the impact of storing indexes on disk and flash devices in three configurations. We compare storing both index and relation on (i) a single disk and (ii) a single SSD. Compared to the actual data, indexes are accessed more frequently and are much smaller. Therefore, we also evaluate (iii) storing the index on an SSD and relation on disk, which allows a much larger dataset than if it is all on an SSD. We use the ORDERS relation and flash SSD-Fast for these experiments. We create a bitmap index on the first column with a total size of 375 megabytes and vary the selectivity, which affects the utilization of the index. Lower selectivity means the index is more useful, while at higher selectivity there is less opportunity to skip over unnecessary rows.

Figure 9 plots the query execution time for these three configurations. We make two observations on the impact of indexing. First, we find no difference in the benefit of indexes between disk and SSD: in both cases, the change in selectivity leads to a similar change in performance, indicating that index behavior affects performance on both devices similarly. Thus, the change in performance is not dependent on the device.

Second, at high selectivities we observe that storing the index on an SSD improves performance by 10% compared to the disk-only case. In this case, the time to access the index is a noticeable fraction of total query execution time, so the faster SSD improves performance. However, we observe that at low selectivities, storing the index on an SSD and the data on a disk has little benefit. The performance is similar because PostgreSQL aggressively prefetches index data to hide the cost of access, and thus reading the index has little impact on performance.

6. FINDINGS AND IMPLICATIONS

The goal of this paper is to evaluate different components and optimizations in the database storage hierarchy for flash storage. We present a holistic view of mechanisms spanning the design of database, OS I/O scheduling, and the characteristics of different storage devices. Our analytical models back our experimental findings on the performance tradeoffs of these mechanisms.

In this section, we present the design implications on future database and operating systems for effectively embracing flash storage.

Storage Layouts. We find that, unlike on disks, column stores outperform row stores on flash devices for a wide variety of query workloads. Similar to disks, they outperform row stores for low projectivity queries because they make better use of I/O bandwidth. Unlike disks, this tradeoff holds for high projectivity queries as well because SSDs possess negligible seek overheads. Our findings are consistent across different flash device models and disk configurations. At a high level, these findings make a strong argument: database storage layouts that improve effective utilization of bandwidth best suit the performance characteristics of flash storage.

Database Compression. We find that data compression, which optimizes I/O bandwidth, has a greater benefit for SSDs than for disks, because I/O bandwidth accounts for a greater portion of performance. However, upcoming faster flash devices over new host interconnects, such as Fusion-IO ioXtreme SSD over PCI-e bus [2] and Sun F5100 flash arrays over SAS interfaces [30], can effectively reduce the overlap between CPU and I/O wait times. Such devices may lead to CPU-bound workloads that do not benefit from compression, or will require new compression schemes that balance CPU and I/O utilization.

Database Prefetching. We find that prefetching contiguous blocks to compensate for slow disk seeks is no longer beneficial for flash storage. Furthermore, the fast random access of SSDs provides new opportunities for the redesign of database prefetching. Rather than prefetching only sequential data, database storage managers can leverage their knowledge about the block access patterns of different query workloads. For example, they can effectively prefetch more distant pages with better temporal locality by using stride prefetching at a negligible cost of seeking on flash.

Storage Indexing. For low selectivity queries, indexes accelerate execution time by caching index pages in main memory for both disks and SSDs. However, for high selectivity queries, the effective utilization of indexes increases and storing indexes on flash offers significant performance improvement. Therefore, future database storage managers can benefit from the design of new hybrid systems that use flash for storing indexes.

Concurrency. We find that the performance of flash storage scales linearly with an increase in the degree of concurrency. In contrast, competing database queries can degenerate into a seeking workload and significantly degrade disk performance. Database mechanisms that optimize for disk performance by sharing the same scanner across different queries [19] can be significantly simplified. Thus, flash storage offers a cleaner alternative for redesigning such database mechanisms and also providing scalable and faster performance.

Disk Scheduling. In addition to database mechanisms for managing seeking workloads, operating systems also cluster nearby I/O requests by reordering or delaying them. We find less than 3 percent difference for SSDs between the performance of NOOP and CFQ I/O scheduling at the block layer in the Linux kernel. Therefore, flash storage requires rethinking the design of a light-weight block layer in the operating system which keeps up with an order of magnitude low access latency of flash than disks.

7. RELATED WORK

This paper draws on past work investigating database optimizations for flash and disk storage. We categorize this work into two broad classes: data layouts for flash and disk storage, and measurement studies on understanding the performance characteristics of flash devices.

Database Storage Layouts. Traditionally, database systems have mostly used the N-ary storage model (NSM), a page-based storage layout to store tuples contiguously. To save on the memory and disk bandwidths for queries projecting on a small fraction of tuples, Copeland et al. first proposed the decomposition storage model (DSM) [13]. Recently, more DSM-like (column store) commercial products and research prototypes have appeared, such as SybaseIQ, Vertica, C-Store [29] and MonetDB/X100 [10]. PAX (Partition Attribute Across) [9] is a hybrid approach which uses a DSM-like organization within a NSM page, thereby optimizing for memory bandwidth. All these layouts trade I/O performance between different workloads. Harizopoulos et al. first investigated the performance tradeoffs for row and column stores [18]. Later, Holloway et al. [20] and Abadi et al. [7] answered many unresolved questions by focusing on a wider variety of scenarios. However, these studies only focus on the performance characteristics of disks, which widely differ from flash devices.

The most closely related work that focuses on flash-based database storage is by Tsirogiannis et al. [31, 27]. The authors investigate the suitability of a hybrid column-based page layout, based on PAX architecture, and compare it with NSM on a single flash device. They propose a new scan and join algorithm which leverages the column-based page layout to improve read efficiency. In contrast, we focus on the broader performance differences between disk and flash. We isolate the scenarios where the performance tradeoffs between row and column stores differ for flash devices from disks and provide analytical models for such differences. Furthermore, we analyze the tradeoffs for other disk-oriented optimizations like data compression, prefetching and I/O scheduling and highlight the additional benefits of flash devices for concurrent workloads.

Flash Measurement Benchmarks. Many studies have benchmarked the read and write performance of different flash devices to reveal their internals and provide hints for their optimal usage for different access patterns [8, 11, 12, 25]. Agrawal et al. present a taxonomy of design tradeoffs for the internal organization of SSDs [8]. They find that SSD performance is highly sensitive to workload and that FTL design choices greatly impact performance. Bouganim et al. describe uFLIP, a benchmark for measuring the response times for different flash access patterns [11]. Chen et al. present a measurement study investigating the intrinsic characteristics and system implications of solid-state disks [12]. Similar to our work, all these studies acknowledge the high variance in the performance characteristics across different flash devices. However, in contrast to these past studies, which only focus on the basic performance characteristics of flash, this paper steps forward by investigating the impact of flash on storage optimizations in database and operating systems.

8. CONCLUSIONS

Database storage has been heavily optimized for disks over the last few decades. Compared to disks, flash devices provide an order of magnitude lower read/access latencies, much higher bandwidths and negligible seek overheads. In the light of these differences, we revisit major database storage optimizations in this paper, including data layouts, compression, database prefetching and indexes on flash. We analytically model the performance tradeoffs of these mechanisms for flash storage across different workload variations. Our results show that most optimizations for disk are still useful for flash, but differ in the degree of benefit. Furthermore, we pro-

vide interesting design implications on future database and operating systems for effectively embracing flash storage.

9. REFERENCES

[1] C-Store: A Column-Oriented Database. `http://db.csail.mit.edu/projects/cstore`.

[2] Fusion-IO ioXtreme PCI-e SSD Datasheet. `http://www.fusionio.com/ioxtreme/PDFs/ioXtremeDS_v.9.pdf`.

[3] NYTimes: Counting Down to the End of Moore's Law, May 2009. `http://tinyurl.com/o2nz2j`.

[4] PostgreSQL Database Server. `http://www.postgresql.org`.

[5] TPC-H Toolkit. `http://www.tpc.org/tpch`.

[6] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[7] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, 2008.

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX*, 2008.

[9] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[10] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[11] L. Bouganim, B. por Jonsson, and P. Bonnet. uflip: Understanding flash io patterns. In *CIDR*, 2009.

[12] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS*, 2009.

[13] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[14] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, 1998.

[15] J. Gray. Tape is dead, disk is tape, flash is disk, ram locality is king, Dec. 2006. `http://tinyurl.com/d2enxp`.

[16] A. Halverson, J. Beckmann, J. Naughton, and D. J. DeWitt. A comparison of c-store and row-store in a common framework. In *Technical Report, University of Wisconsin-Madison, TR1566*, 2006.

[17] R. A. Hankins and J. M. Patel. Data morphing: An adaptive cache-conscious storage technique. In *VLDB*, 2003.

[18] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[19] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[20] A. L. Holloway and D. J. Dewitt. Read-optimized databases, in depth. In *VLDB*, 2008.

[21] Intel. X-25 mainstream ssd datasheet, May 2009. http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf.

[22] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous IO. In *SOSP*, 2001.

[23] H. Kim and S. Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. In *USENIX FAST*, 2008.

[24] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In *ACM TODS, Volume-6, Issue 2*, 1981.

[25] D. Myers. On the use of nand flash memory in high-performance relational databases. In *MIT MSc. Thesis*, 2008.

[26] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant array of inexpensive disks (raid). In *SIGMOD*, 1988.

[27] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *Fourth Workshop on Data Management on New Hardware (DaMoN), SIGMOD*, 2008.

[28] E. Shriver, C. Small, and K. A. Smith. Why does file system prefetching work? In *USENIX*, 1999.

[29] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column oriented database. In *VLDB*, 2005.

[30] Sun-Online. Sun Storage F5100 Flash Array. `http://www.sun.com/F5100`.

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

[32] T. Westmann, D. Kossman, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. In *SIGMOD Rec, 29(3)*, 2000.