Matching Database Access Patterns to Storage Characteristics

Jiri Schindler Anastassia Ailamaki Gregory R. Ganger

Carnegie Mellon University

Abstract

Today's storage interfaces hide device-specific details, simplifying system development and device interoperability. However, they prevent database systems from exploiting devices' unique performance characteristics. Abstract and device-independent annotations to existing storage interfaces can cleanly expose key device characteristics that improve performance and simplify manual tuning. By automatically matching access patterns to device strengths, a database storage manager can achieve robust performance even with workloads competing for the same storage resource. For example, disk-optimized accesses result in simultaneous improvement of up to 3x for DSS workloads and 7% for a competing OLTP workload. As another example, accesses to relational tables can take advantage of MEMS-based storage parallelism to achieve order of magnitude improvements in selective scans.

1 Introduction

Database storage managers employ sophisticated algorithms attempting to exploit the performance available inside today's storage systems. However, because the communication between the storage manager (SM) and a storage device is limited by a high-level protocol, both the SM and the device make decisions largely in isolation and do not realize the device's full potential. Based on a few implicit assumptions, a SM guesses device characteristics to improve I/O performance. Similarly, a storage device observes I/O access patterns coming from the storage manager (but not the queries themselves) and tries to adapt its behavior to satisfy requests more efficiently. This absence of communication leads to inefficient utilization of storage device resources and degraded performance especially under workloads that change dynamically.

Exposing key storage device performance characteristics in a well-defined (small) set of attributes will allow SMs to realize the performance potential hidden behind today's high-level storage interfaces. With explicit information, provided by the storage device and encapsulated in these performance attributes, a SM can ensure more efficient use of storage devices. Being close to the queries being executed, the SM can make its decisions within the context of the true query access patterns.

Combining detailed knowledge of access patterns of currently executed queries with explicit performance hints provides three major benefits. First, a storage manager can make more informed decisions that result in shorter query execution times. Even when several queries execute in parallel and contend for the same storage device, a SM can automatically tune each query's access patterns, resulting in more efficient execution. Second, this automatic adjustment simplifies a difficult and error-prone task of manual performance tuning. With devices explicitly providing key characteristics, there is no need to expose a set of hard-to-understand manually-set configuration parameters. Instead, the SM can automatically tune its performance to dynamically changing workloads without administrator's intervention.

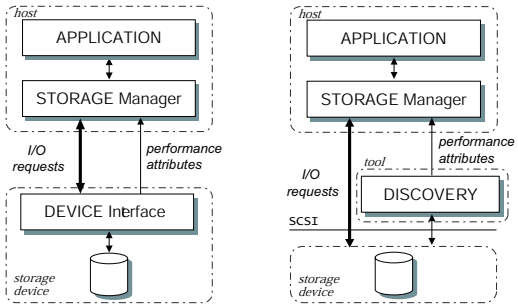
As demonstrated by our implementation of a database storage manager [3], two attributes can effectively capture characteristics of different storage devices (e.g., disk drives, disk arrays, and MEMS-based storage) to bring about significant performance improvements. For example, running compound workloads of decision support system queries (DSS) and on-line transactional processing (OLTP), we can achieve up to a $3\times$ performance improvement on DSS queries, while simultaneously improving OLTP throughput by 7%. Most importantly, these improvements do not break the abstractions within a storage manager and require only minimal changes to existing data structures.

The research described here presents a new, more robust approach to database storage management, which automatically adjusts query access patterns to unique strengths of different storage devices and dynamic workload changes. This results in better and more predictable performance for workloads competing for the same storage resources. This approach also eliminates the need for manual performance-tuning knobs, which simplifies the configuration of database systems.

2 Storage Model

Virtually all of today's storage devices use an interface (i.e., SCSI or IDE) that presents them as a linear space of equally-sized blocks. Each block is uniquely addressed by an integer, called a *logical block number* (LBN), which grows from 0 to LBN_{max}. This view offers a simple programming model and does not require a storage manager to include any device-specific knowledge. Unfortunately, it hides important non-linearities in access times to different *LBN*s.

The access time non-linearities stem from the combination of access history and device properties. To work around the non-linear access time, there exists an un-



(a) **Storage model.** (b) **Discovery tool.**

Figure 1: **Two approaches to conveying performance attributes.** (a) illustrates a host and a storage device that communicate via the extended interface that conveys the necessary performance attributes. (b) shows an alternate apporach where the storage device speaks a conventional SCSI. A device-specific discovery tool extracts performance chatacteristics and exports them to the SM. Note that the same SM works with either approach. In database systems, the depicted application corresponds to the query optimizer and the execution engine.

written contract between storage managers and storage devices, which states that (i) logical blocks near each other can be accessed faster than those farther away and that (ii) accessing logical blocks sequentially is generally much more efficient than random access. Rather than working on these loose assumptions, explicitly stated device characteristics via a well-defined interface allow a storage manager to take advantage of the unique storage device strengths and avoid access patterns that are inefficient.

Encapsulating these characteristics into a few attributes that annotate the device's address space will maintain today's abstractions and does not require code modifications to the storage manager with each introduction of a new device. When a new device is plugged into a system, a storage manager can simply query the device to obtain the values of these performance attributes.

Performance attributes can be provided to a storage manager by two different approaches, as illustrated in Figure 1. In the first one, specialized tools automatically determine device's characteristics without any additional support beyond current standardized interfaces [4]. This tool then encapsulates the explored characteristics into the set of performance attributes. In the second approach, the storage device itself encapsulates its performance characteristics and exports them directly to the storage manager.

2.1 Performance Attributes

ACCESS DELAY BOUNDARIES: This attribute denotes preferred storage request access patterns i.e., the block groupings into units that yield most efficient accesses. These groupings, called *ensembles* of contiguous *LBN*s, are a manifestation of variable-sized disk track boundaries [5] or stripe unit sizes in RAID configurations. The *ensemble* sizes may be different at different parts of the device's address space.

This attribute captures the following notions: (i) requests that are exactly aligned on the reported boundaries and span all the blocks of a single unit are most efficient, (ii) requests smaller than the preferred groupings, should be, if possible, aligned at the boundary. If they are not aligned, they should not cross a boundary. In a striped RAID configuration, for example, a stripe-unit-sized READ request that spans two stripe units because it is not aligned, will be split into two requests issued to two different disks. These two smaller requests will be less efficient and, if one of the disks is very busy, the overall request latency will be high.

PARALLELISM: Given a device's linear address space, this attribute describes how many *LBN*s can be accessed in parallel. The device interface provides to the storage manager an *equivalent* set of non-contiguous *LBN*s that can all be accessed simultaneously.

For example, a a logical volume of a storage array which has *n* mirrored replicas can access *n* different pieces of data in parallel. Another example is a MEMS-based storage device (under development at IBM, HP, and Carnegie Mellon University), called MEMStore, which can access data in parallel by a subset of the thousands of available read/write heads. Using this parallel access, even data that are not laid contiguously, can be accessed efficiently. Thus, both a row-major and column-major order access to a two-dimensional database table can be efficient.

2.2 Acquisition of Storage Characteristics

Today's storage interfaces cannot convey performance attributes directly. However, they can be discovered via specialized tools that encapsulate the discovered characteristics into these performance attributes. These tools are tailored to a particular storage device as they make assumptions about the device's inner-workings and work seamlessly within the proposed architecture as depicted in Figure 1(b).

The discovery tool that we build, called DIXtrac, assembles test vectors of READ and WRITE commands which are individually timed [4]. These test vector requests can be either interjected into the stream of requests coming from the storage manager or the discovery can be done off-line as a one-time cost during device initialization. The former approach has the advantage that the discovery tool can dynamically tune the performance characteristics attributes at a cost of interfering with the normal stream of requests. The latter approach does not slow down the device, but depending on the type of the device, its characteristics can change over time.

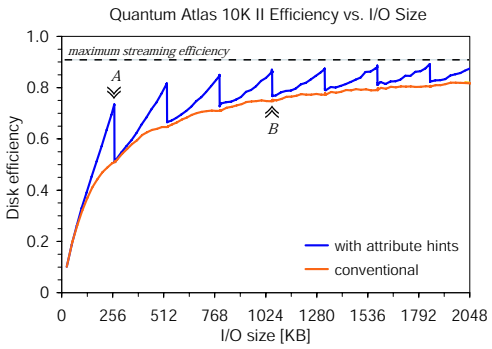


Figure 2: **Measured advantage of accesses with explicit performance attributes.** *Disk efficiency* is the fraction of total access time spent moving data to or from the media. The *conventinal* and *with attribute hints* lines show disk access efficiency for random, constant-sized reads. *with attribute hints* access takes the same stream of random requests, but aligns them on disk track boundaries (a manifestation of the ACCESS DELAY BOUNDARIES attribute). The peaks are the device's ensemble unit sizes and correspond to multiples of the track size. Point A highlights the higher efficiency of access with performance attribute hints (0.73, or 82% of the maximum). Point B shows where conventional efficiency catches up to to efficiency at Point A. The maximum streaming efficiency is less than 1.0, because no data is transferred when switching from one track to the next.

2.3 Quantifying I/O Access Efficiency

Because of disk drive characteristics, random small accesses to data are much less efficient than larger ones. As shown in Figure 2 by the line labeled *conventional*, the disk efficiency increases with increasing I/O size by amortizing the positioning cost over larger data transfer. To take advantage of this trend, storage managers buffer data and issue large I/O requests.

With explicit information about access delay boundaries, the same set of random requests can be aligned within these (track) boundaries to achieve much higher efficiency as illustrated in Figure 2 by the line labeled *with attribute hints*. The difference between the *with attribute hints* access and the maximum streaming efficiency is due to an average seek of 2.2 ms. Moreover, the access *with attribute hints* consistently provides this level of efficiency across disks from at least the past 10 years, making it robust choice for automatically sizing efficient disk accesses.

The increased efficiency of the track-aligned disk access comes from a combination of three factors. First, a track-aligned I/O whose size is one track or less does not suffer a head switch, which, for modern disks is equivalent to a 1–5 cylinder seek. Second, a disk firmware technology known as zero-latency access eliminates rotational latency by reading data off the track out-of-order as soon as the head is positioned. The on-board disk cache buffers the data and sends them to the host in ascending order. Third, with several requests in the disk queue, seeking to the next request's location can overlap with previous request's data transfer to the host.

3 Robust Storage Management

With I/O operations being the dominant cost in query execution, the focus of query optimization, both at the optimizer and the storage manager levels, has traditionally been to achieve efficient storage access patterns. Unfortunately, two problems make this difficult. First, the many layers of abstractions between the optimizer and the storage manager, together with high-level storage interfaces, make it difficult to quantify the efficiency of different access patterns. Second, when there is contention for data or resources, the resulting access pattern is less efficient. For example, an efficient sequential access assumed during query planning stage is now interleaved with other requests, introducing unplanned-for seek and rotational delays. Combined, these two factors lead to loss of I/O performance and longer query execution times.

Transforming query plans into individual I/O requests requires the storage manager to make decisions about request sizes, locations, and the temporal relationship (i.e., scheduling) to other requests. While doing so, a storage manager considers resource availability and the level of contention for them caused by other queries running in parallel. This balance is quite sensitive and prone to human errors. To tune its performance, DBMS rely on generic parameters set by a database administrator (DBA). This, however, makes it difficult to adapt to the device-specific characteristics and dynamic query mixes, resulting in inefficiency and performance degradation.

We promote an alternate approach that (i) does not expect DBAs to get tuning knobs right and (ii) does not rely on crude SM built-in methods for dynamically determining the I/O efficiency of differently-sized requests be correct. Instead, the storage manager understands a bit more about underlying device characteristics by receiving hints explicitly from the storage device. It can thus automatically adjust its access patterns to generate I/Os that are most efficient for the device. Most importantly, this provides guarantees to the query optimizer that access patterns are as efficient as originally assumed when the query plan was composed.

With explicit information about access pattern efficiency, the storage manager can focus solely on data allocation and access. It groups pages together such that they can be accessed with efficient I/Os prescribed by the storage device characteristics. Such grouping meshes well with existing storage manager structures, which call these groups segments, or extents. Hence, implementing these changes requires only minimal changes; our implementation within a Shore prototype database storage manager [1], called Lachesis, modified only 680 lines of C++ code of a total of quarter million. The clean high-level performance attribute abstractions leave the current interfaces between the storage device, the storage manager, and the query optimizer unchanged.

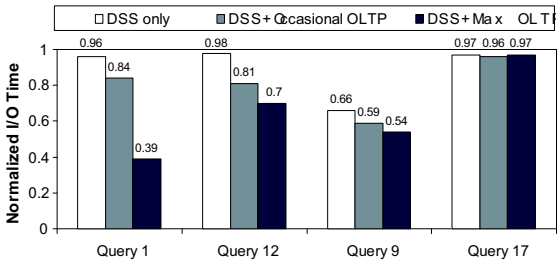


Figure 3: **DB2 TPC-H query execution.** This graph shows four different trends in query I/O time improvements for our Lachesis system normalized to the runs on the original system . Queries 5, 10, 13, 15, 18, 19, 20, and 22 have the same profile as the graphed query 1, queries 6 and 16 as query 12, queries 2, 3, 7, and 8 as query 9, and queries 11 and 14 show the same limited improvement as query 17.

3.1 Compound Workload Results

To demonstrate that with explicit performance storage characteristics, a storage manager can execute queries efficiently even in the face of competing traffic, we ran a compound workload of OLTP and DSS queries. We measured the runtime of DSS queries of the TPC-H benchmark and modeled competing device traffic by simultaneously running a TPC-C random transaction mix.

We ran all experiments on a uni-processor system with the TPC-C and TPC-H data instances stored on a single disk. The TPC-H instance included a 1 GB database and ran one query at a time. The TPC-C instance run one transaction at a time to allow fine-grain of control the I/O activity. We varied the amount of the competing traffic to the DSS queries by changing the keying/think time of the TPC-C benchmark transactions to achieve a target rate of 0 to TpmC$_{MAX}$ transactions per minute.

We also evaluated the benefits on a commercial database system. Since we did not have access to the IBM DB2 source code, we ran all 22 queries of the TPC-H benchmark in DB2 and captured their device-level I/O traces. We simulated the effects of Lachesis adjusting request sizes in the original traces and replaying these modified traces. We simulated the competing OLTP traffic by injecting 8 KB random I/Os during the TPC-H trace replay and varied their I/O arrival rate. The results obtained from our modified Shore prototype implementation and the DB2 trace replay experiments show identical trends.

Figure 3 shows the execution time of representative TPC-H queries with Lachesis storage manager. The times are normalized to the original unmodified system. For each query, the left bar, labeled *DSS only*, shows the run time of the given TPC-H query in isolation. The middle bar, labeled *DSS+Occasional OLTP*, shows the case when the TPC-H query experiences a small amount of competing traffic from OLTP activity. The third bar, labeled *DSS+Max OLTP*, shows the case when the OTLP is running at maximal transactional throughput while a TPC-H query executes.

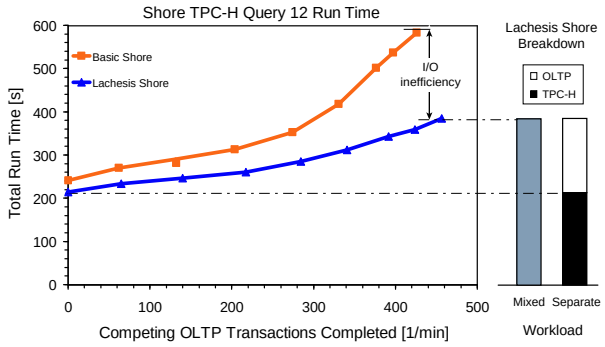


Figure 4: **Shore TPC-H query 12 execution.** The graph on the left shows the amount of time spent in I/O operations as a function of increasing competing OLTP workload. The I/O inefficiency in the basic case is due to extraneous rotational delays and disk head switches when running the compound workload. The two bars illustrate the robustness of our Lachesis design; at each point, both the TPC-H and OLTP traffic achieve their best case efficiency. The **Mixed** workload bar in the right graph corresponds to the $\lambda = 150$ Lachesis-Shore datapoint of the left graph. The **Separate** bar adds the total I/O time for the TPC-H and the OLTP-like workloads run separately.

Lachesis shows modest improvements in the *DSS only* scenario since we hand-tuned the configuration of our original system for efficient sequential accesses that dominate TPC-H query execution. However, there are substantial improvement in the face of competing traffic. Further, the relative improvement grows as the amount of competing traffic increases, indicating the robustness of our system to competing traffic. On average, the improvement for the *DSS+Occasional OLTP* and *DSS+Max OLTP* scenarios was 21% (or 1.3× speedup) and 33% (1.5× speedup) respectively.

Figure 4 compares the execution of the TPC-H query 12 on our prototype system as a function of increasing competing traffic. First, with no competing traffic, we achieve shorter run time by 11%. However, as the amount of competing traffic increases, the difference becomes larger, and results in a 1.5× speedup of execution when the OLTP workload is running at full speed. Second, the two Shore implementations achieve different TpmC$_{MAX}$. While for the Basic-Shore implementation TpmC$_{MAX}$=426.1, Lachesis-Shore achieved 7% higher maximal throughput of 456.7 transactions per minute. Thus, Lachesis not only improves the performance of the TPC-H queries alone, but also offers higher peak performance to the TPC-C workload.

4 Efficient DB Scan Operator

The page layout prevalent in commercial DBMS, called NSM, stores a fixed number of records for all *n* attributes in a single page. Thus, when scanning a table to fetch records of only one attribute (i.e., column-major access), the scan operator still fetches pages with data for *all* attributes, effectively reading the entire table even though only a subset of the data is needed.

Operation	Data Layout	
	NSM	*PAR*
entire table scan	22.44 s	22.93 s
a_1 scan	22.44 s	2.43 s
a_1 + a_2 scan	22.44 s	12.72 s
100 records of a_1	1.58 ms	1.31 ms

Table 1: **Database access results.** The table shows the runtime of the specific operation on the 10,000,000 record table with 4 attributes for the *NSM* and *PAR* stored on a simulated for MEMStore device. The rows labeled a_1 scan and $a_1 + a_2$ represent the scan through all records when specific attributes are desired. the last row shows the time to access the data for attribute a_1 from 100 records.

With proper allocation of data that utilizes the PARALLELISM attribute, an arbitrary subset of attributes of a single record can be accessed in parallel (i.e., row-major access). Thus, the records are fetched in lock-step, eliminating the need for a join required in the vertically partitioned page layout. Furthermore, our parallelism-aware layout ensures that sequential streaming of one attribute (i.e., column-major access) is realized at the device's full bandwidth by engaging the parallelism to read more records of the same attribute in parallel.

Our parallel-aware layout puts data into pages that span non-contiguous *LBN*s that belong to the same *equivalent* set, ensuring that they can be accessed in parallel. Adjacent pages are laid next to each other such that records of the same attribute in two adjacent pages are mapped to sequential *LBN*s. For each page, the storage manager-maintains its starting *LBN* from which it can determine the disjoint *LBN*s of all attributes in the page by calling the *get_equivalent()* function provided by the storage device interface.

4.1 Results for Selective Scan with MEMStore

Our sample database table consists of 4 attributes a_1, a_2, a_3, and a_4 sized at 8, 32, 15, and 16 bytes respectively. The *NSM* layout consists of 8 KB pages that include 115 records. The *DSM* layout vertically partitions data into 9 separate tables. The *PAR* layout produces pages of 9 *LBN*s (each 512 bytes) with a total of 60 records. The table size is 10,000,000 records (694 MB of data).

Table 1 summarizes the table scan results for the *NSM* and *PAR* cases using a simulated MEMStore device [2]. Scanning the entire table takes approximately the same time for both *NSM* and *PAR* cases. The small run time difference is because the *NSM* packs records into a page slightly more efficiently. Our storage manager is highly efficient when only a subset of the attributes are required. A table scan of a_1 or $a_1 + a_2$ in the *NSM* case always takes 22.44 s, since entire pages including the undesired attributes must be scanned. The *PAR* case only requires a fraction of the time corresponding to the amount of data occupied by each attribute e.g., one-ninth (2.43 s vs. 22.93 s) of the full table scan for the a_1 scan.

The *DSM* case (not shown in the table) exhibits similar results for individual attribute scans as the *PAR* case. In contrast, scanning the entire table is much more expensive as it requires reading several disjoint regions, each containing one vertically partitioned attribute, and additional joins on the attributes.

Comparing the latency of accessing one complete random record under the three different scenarios shows an interesting behavior. The *PAR* case gives an average access time of 1.385 ms, the *NSM* case 1.469 ms, and the *DSM* case 4.0 ms. The difference is due to different access patterns. The *PAR* access includes a random seek to the page's location followed by access to one equivalence class proceeding in parallel. The *NSM* access involves a random seek followed by a sequential access to 16 *LBN*s. Finally, the *DSM* access requires 9 accesses each consisting of a random seek and one *LBN* access.

5 Summary of Contributions

This work makes three contributions. It identifies what performance characteristics information should be exposed, demonstrates how they can be used by storage managers, and quantifies the performance benefits to applications e.g., query executions for different workloads. The performance attributes exposed to a SM are abstract from details about a storage device, device-independent, and yet specific-enough to allow a SM to tune its access patterns to the given device. These attributes do not break established abstractions between the storage device and the SM; they simply annotate the current abstraction of a storage system, namely the linear address space of fixed-size logical blocks (*LBN*s).

References

[1] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. *ACM SIGMOD International Conference on Management of Data* (Minneapolis, MN, 24–27 May 1994). Published as *SIGMOD Record*, **23**(2):383–394, 1994.

[2] J. L. Griffin, J. Schindler, S. W. Schlosser, J. C. Bucy, and G. R. Ganger. Timing-accurate storage emulation. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 75–88. USENIX Association, 2002.

[3] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. *International Conference on Very Large Databases* (Berlin, Germany, 9–12 September 2003). Morgan Kaufmann Publishing, Inc., 2003.

[4] J. Schindler and G. R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[5] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.