Lachesis: Robust Database Storage Management Based on Device-specific Performance Characteristics

Jiri Schindler Anastassia Ailamaki Gregory R. Ganger

Carnegie Mellon University

Abstract

Database systems work hard to tune I/O performance, but do not always achieve the full performance potential of modern disk systems. Their abstracted view of storage components hides useful device-specific characteristics, such as disk track boundaries and advanced built-in firmware algorithms. This paper presents a new storage manager architecture, called *Lachesis*, that exploits and adapts to observable device-specific characteristics in order to achieve and sustain high performance. For DSS queries, *Lachesis* achieves I/O efficiency nearly equivalent to sequential streaming even in the presence of competing random I/O traffic. In addition, *Lachesis* simplifies manual configuration and restores the optimizer's assumptions about the relative costs of different access patterns expressed in query plans. Experiments using IBM DB2 I/O traces as well as a prototype implementation show that *Lachesis* improves standalone DSS performance by 10% on average. More importantly, when running concurrently with an on-line transaction processing (OLTP) workload, *Lachesis* improves DSS performance by up to 3×, while OLTP also exhibits a 7% speedup.

1 Introduction

The task of ensuring optimal query execution in database management systems is indeed daunting. The query optimizer uses a variety of metrics, cost estimators, and run-time statistics to devise a query plan with the lowest cost. The storage manager orchestrates the execution of queries, including I/O generation and caching. To do so, it uses hand-tuned parameters that describe various characteristics of the underlying resources to balance the resource requirements encapsulated in each query plan. To manage complexity, the optimizer makes decisions without runtime details about other queries and modules, trusting that its cost estimates are accurate. Similarly, the storage manager trusts that the plan for each query is indeed well-chosen and that the database administrator (DBA) has tuned the knobs correctly.

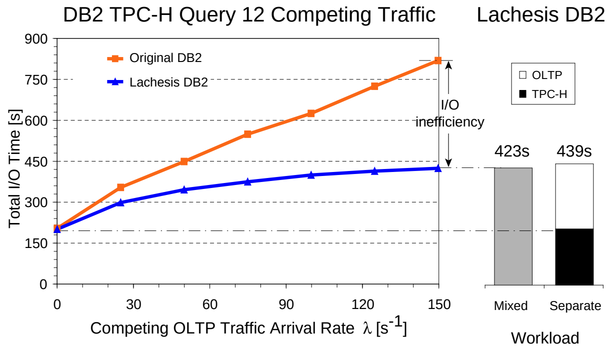

Figure 1: **TPC-H query 12 execution (DB2)** The graph on the left shows the amount of time spent in I/O operations as a function of increasing competing OLTP workload, simulated by random 8 KB I/Os with arrival rate λ. The I/O inefficiency in the original DB2 case is due to extraneous rotational delays and disk head switches when running the compound workload. The two bars illustrate the robustness of *Lachesis*; 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-DB2 datapoint of the left graph. The **Separate** bar shows the total I/O time for the TPC-H and the OLTP-like workloads when run separately.

To minimize I/O costs, query optimizers and storage managers have traditionally focused on achieving efficient storage access patterns. Unfortunately, two issues complicate this task. First, the multiple layers of abstraction between the query execution engine and the storage devices complicate the evaluation of access pattern efficiency. Second, when there is contention for data or resources, efficient sequential patterns are broken up. The resulting access pattern is less efficient because the originally sequential accesses are now interleaved with other requests, introducing unplanned-for seek and rotational delays. These two factors lead to significant degradation of I/O performance and longer execution times.

This paper proposes a new storage manager architecture, called *Lachesis*, which automatically detects and transparently exploits device characteristics. By utilizing a few automatically-measurable performance characteristics, a storage manager can specialize to its devices and provide more robust performance in the presence of concurrent query execution. In particular, it can support high I/O concurrency without disrupting planned sequen-

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

tial I/O performance. *Lachesis* eliminates the need for several (previously DBA-specified) parameters, thereby simplifying DBMS configuration and performance tuning. Most importantly, it restores the validity of the assumptions made by the query optimizer about the relative costs of different storage access patterns.

Figure 1 demonstrates the effect of *Lachesis* on the performance of a decision-support (DSS) workload as a function of competing on-line transaction processing (OLTP) I/O traffic. As shown in the graph, *Lachesis* guarantees that TPC-H [10] query 12 (dominated by sequential table scans) enjoys streaming I/O performance as anticipated by the optimizer, regardless of the rate of concurrent OLTP traffic.

We have implemented the *Lachesis* architecture within the Shore storage manager [5]. The implementation involves three parts: extraction of device characteristics, modifications to data placement algorithms, and modifications to prefetch and buffer management policies. The extraction, which runs once before loading a new device, uses test patterns [35] to identify track boundaries [31]. The placement routines are modified to avoid putting pages across these boundaries. The read/write routines are modified to utilize aligned, full-track accesses where possible. With fewer than 800 lines of C++ code changed, and no modifications to existing interfaces and abstractions, *Lachesis* prevents query concurrency from interfering with I/O efficiency.

Experiments with the prototype implementation, as well as with DB2 trace simulations, show modest performance improvements for standalone workloads. DSS workloads exhibit a 10% average performance improvement (with a maximum of 33%), whereas OLTP workloads (dominated by random I/O) remain unaffected. When running compound workloads, however, *Lachesis* exhibits up to a three-fold performance improvement on DSS queries, while simultaneously improving OLTP throughput by 7%. Similar benefits are shown for a disk array with stripe unit sizes matching track sizes.

The rest of this paper is organized as follows. Section 2 discusses background and related work. Section 3 describes the *Lachesis* architecture. Section 4 describes our implementation inside Shore storage manager. Section 5 evaluates *Lachesis*. Section 6 summarizes the paper's results.

2 Background and Related Work

Modern database management systems typically consist of several cooperating modules; the query optimizer, the query execution engine, and the storage manager are relevant to the topic of this paper. As illustrated in Figure 2, for each query the optimizer evaluates the cost of each alternative execution plan and selects the one with the lowest cost. The execution engine allocates resources for the execution of the selected query plan, while the storage manager communicates with the storage devices as needed.

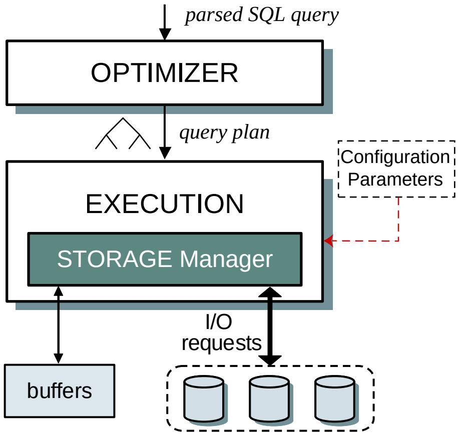

Figure 2: **Query optimization and execution in a typical DBMS.**

2.1 Optimizing for I/O

Query optimizers use numerous techniques to minimize the cost of I/O operations. Early optimizers used static cost estimators and run-time statistics [33] collected by the storage manager to estimate the number of I/O operations executed by each algorithm [16] (e.g., loading a page from a storage device into the buffer pool or writing a page back to the device). Due to the physical characteristics of disk drives, random I/O is significantly slower than sequential scan. To reflect the performance difference, today's commercial DBMS optimizers distinguish between random and sequential I/O using cost models that take into account the data access pattern dictated by each query operator. Despite capturing this important disk performance feature, however, each query is optimized separately. In addition, the optimizer has limited knowledge of the characteristics of the underlying disk subsystem. The calculated access pattern costs are therefore not likely to be observed during execution. To maintain robust performance, the execution engine must uphold the assumed performance across disk system types and concurrent query execution.

2.2 Ensuring Efficient I/O Execution

During query execution, the execution engine asks the storage manager for data from the disks. The storage manager considers factors such as resource availability and contention across concurrent queries and decides each request's size, location, and the temporal relationship to other requests. Request-related decisions are also influenced by DBA-specified parameters. For instance, IBM DB2's `EXTENTSIZE` and `PREFETCHSIZE` parameters determine the maximal size of a single I/O operation [17], and the `DB2_STRIPED_CONTAINERS` pa-

Disk	Year	RPM	Head Switch	Avg. Seek	Sectors Per Track	No. of Tracks	Heads	Capacity
HP C2247	1992	5400	1.0 ms	10 ms	96–56	25649	13	1 GB
Quantum Atlas III	1997	7200	1.0 ms	8.5 ms	256–168	80570	10	9 GB
IBM Ultrastar 18LZX	1999	10000	0.8 ms	5.9 ms	382–195	116340	10	18 GB
Quantum Atlas 10K	1999	10000	0.8 ms	5.0 ms	334–224	60126	6	9 GB
Seagate Cheetah X15	2000	15000	0.8 ms	3.9 ms	386–286	103750	6	18 GB
Maxtor Atlas 10K III	2002	10000	0.6 ms	4.5 ms	686–396	124088	4	36 GB

Table 1: **Representative SCSI disk characteristics.** Although there exist versions of the Seagate and Maxtor drives with higher capacities, the lower capacity drives are typically installed in disk arrays to maximize the number of available spindles. Note the small change in head switch time relative to other characteristics.

rameter instructs the storage manager to align I/Os on stripe boundaries.

In order for the queries to be executed as planned, the storage manager must balance the competing resource requirements of the queries being executed while maintaining the access cost assumptions the query optimizer used when selecting a query plan. This balance is quite sensitive and prone to human errors. In particular, high-level generic parameters make it difficult for the storage manager to adapt to device-specific characteristics and dynamic query mixes. This results in inefficiency and performance degradation.

Recent efforts address the problem for a subset of disk workloads by proposing storage mechanisms that exploit freedom to reorder storage accesses. Storage latency estimation descriptors [24] estimate the latency for accessing the first byte of data and the expected bandwidth for subsequent transfer. Steere [34] proposed a construct, called a set iterator, that exploits asynchrony and non-determinism in data accesses to reduce aggregate I/O latency. The River project [3] uses efficient streaming from distributed heterogenous nodes to maximize I/O performance for data-intensive applications. Other research [8, 13, 19] exploited similar ideas at the file system level to achieve higher I/O throughput from the underlying, inherently parallel, hardware. Our goal is to achieve maximum storage efficiency without changing the existing DBMS structure and policies.

2.3 Exploiting Observed Storage Characteristics

To achieve robust performance, storage managers need to exploit observed storage system performance characteristics. DBA-specified parameters are too high level to provide sufficient information to the storage manager, therefore DBAs cannot finely tune all performance knobs. Current storage managers complement DBA knob settings with methods that dynamically determine the I/O efficiency of differently-sized requests by issuing I/Os of different sizes and measuring their response time. Unfortunately, such methods are unreliable and error-prone.

The approach proposed in this paper provides the storage manager with a better understanding of the underlying storage characteristics. If the storage device is capable of directly exporting its performance characteristics, the storage manager can automatically adjust its access patterns to generate I/Os that are most efficient for the device. If the storage device cannot directly export characteristics, existing experimentation tools can still acquire sufficient information before loading the device with data. For example, Worthington et al. describe algorithms for online extraction of parameters from SCSI disk drives [35]. An automated tool called DIXtrac [30] extends these algorithms to extract detailed disk characteristics in a few minutes, including complete logical-to-physical mappings, mechanical overheads, and caching/prefetching algorithms.

Several research efforts have proposed exploiting low-level disk characteristics to improve performance. Ganger [14] and Denehy et al. [11] promote two similar approaches for doing so, which *Lachesis* shares, wherein host software knows more about key device characteristics; those authors also survey prior disk system research and how it fits into the general approach. The most relevant example [31] evaluates the use of track-aligned extents in file systems, showing up to 50% performance increase due to avoidance of rotational latency and head switch costs. Our work builds on this previous work with a clean architecture for including the ideas into DBMS storage managers to automate storage performance tuning and achieve robust storage performance.

The database community exploited disk characteristics many years ago. The Gamma Database Machine [12], for instance, accessed data in track-sized I/Os simply by knowing the number of sectors per track. Unfortunately, due to high-level device interfaces and built-in firmware functions, straightforward mechanisms like the one used in Gamma are no longer possible. As indicated in Table 1, zoned geometries and advanced defect management in modern disks result in cylinders being composed of tracks with varied numbers of sectors. Consequently, there exists no single value for the number of sectors per track across the entire device.

2.4 Exploiting Memory Hierarchy Characteristics

In the past five years, memory latency has been recognized as an increasingly important performance bottleneck for several compute and memory-intensive database applications [2, 4]. To address the problem, recent research improves utilization of processor caches by in-page data partitioning [1] and cache-friendly indexing structures [7, 27]. Another approach [22] factors processor cache access parameters into the optimization process by incorporating data access pattern and cache characteristics into the query operator cost functions. These approaches, however, cannot be used to tune and estimate disk access costs, because I/O access times are functions of spatial and temporal request inter-relationships. Proposed techniques optimized for cache and memory performance are therefore orthogonal to the work described in this paper.

3 Robust Storage Management

This section describes the *Lachesis* storage manager architecture, which bridges the information gap between database systems and underlying storage devices. This allows a DBMS to exploit device characteristics to achieve robust performance for queries even with competing traffic. At the same time, *Lachesis* retains clean high-level abstractions between the storage manager and the underlying storage devices and leaves unchanged the interface between storage manager and query optimizer.

3.1 Lachesis Overview

The cornerstone of the *Lachesis* architecture is to have a storage device (or automated extraction tool) convey to the storage manager explicit information about efficient access patterns. While the efficiency may vary for different workloads (i.e., small random I/Os are inherently less efficient than large sequential ones), this information allows a storage manager to always achieve the best possible performance regardless of the workload mix. Most importantly, this information provides guarantees to the query optimizer that access patterns are as efficient as originally assumed when the query plan was composed.

Ideally, the storage manager learns about efficient device accesses directly from the storage device, which encapsulates its performance characteristics in a few well-defined and device-independent attributes. Section 3.3 describes these attributes and how to obtain them in more detail. During query execution, the storage manager uses these hints to orchestrate I/O patterns appropriately. No run-time performance measurements are needed to determine efficient I/O size.

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 [17, 20]. Hence, implementing *Lachesis* requires only minimal changes, as discussed in Section 4.

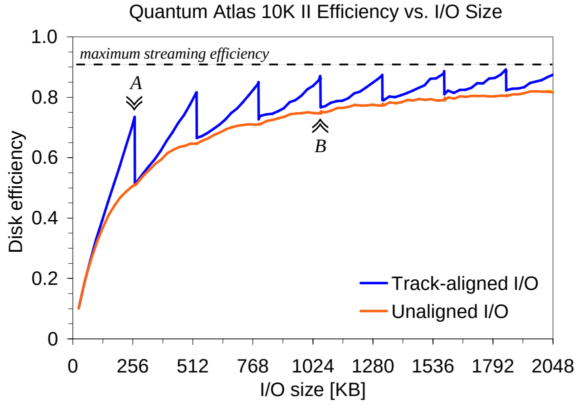

Figure 3: **Disk access efficiency.** This graph plots disk efficiency as a function of I/O size for track-unaligned and track-aligned random requests within disk's first zone. *Disk efficiency* is the fraction of total access time (which includes seek and rotational latency) spent moving data to or from the media. The maximum streaming efficiency (i.e., sequential access without seeks and rotational latencies) is due to head switches between accesses to adjacent tracks. The peaks in the track-aligned curve correspond to multiples of the track size.

3.2 Efficient I/O Accesses

Because of disk drive characteristics, random small accesses to data are much less efficient than larger ones. As shown in Figure 3 by the line labeled *Unaligned I/O*, disk efficiency increases as a function of the I/O size by amortizing the positioning cost over more data transfer. To take advantage of this trend, storage managers buffer data and issue large I/O requests [18, 29]; this, however, creates a tension between increased efficiency and higher demand for buffer space.

As observed before [31], track-aligned access for the same set of random requests, which include seeks and rotational latency, can be much more efficient as shown in Figure 3 by the line labeled *Track-aligned I/O*. Point *A* illustrates that the efficiency of a track-aligned random access (0.73, or 82% of the maximum) is higher than that of an unaligned access for a track-sized request. The difference between maximum streaming efficiency and track-aligned access is due to an average seek of 2.2 ms. At point *B*, the unaligned I/O efficiency catches up to the track-aligned efficiency at point *A*. Thus, a track-aligned I/O 4× smaller than conventional (track-unaligned) I/O, can achieve the same disk efficiency as illustrated by points *A* and *B* in Figure 3. This alleviates the tension between request size and buffer space.

Moreover, track-based access consistently provides this level of efficiency across disks developed over at least the past 10 years. As shown in Table 1, head switch times have decreased by a factor of less than two, while, in the same time span, rotational speeds have increased by 3×. Because of this trend, the head switch time becomes increasingly significant in normal disk accesses, which are not track-aligned. Faster seeks and higher bit densities make this even worse; they make the head control mechanism, which must position heads over smaller areas, more difficult. Therefore, track-aligned and track-sized access continues to be a robust choice for automatically determining efficient disk accesses.

The increased efficiency of track-aligned requests 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 [26]. 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 it to the host in ascending order. Third, with several requests in the disk queue, seeking to the next request's location can overlap with a previous request's data transfer to the host.

A database storage manager exercises several types of access patterns. Small sequential writes are used for synchronous updates to the log. Small, single-page, random I/Os are prevalent in OLTP workloads. Large, mostly sequential I/Os occur in DSS queries. When running compound workloads, there is contention for data or resources, and this sequentiality is broken. In such cases, it is important that the storage manager achieve near-streaming bandwidth when possible without unduly penalizing any of the ongoing queries.

The *Lachesis* architecture exploits the efficiency of track-aligned accesses. Even with significant interleaving, it can achieve efficiency close to that of purely sequential I/O. Furthermore, it does so without using exceptionally large requests, which would require the use of more buffer space and interfere more with competing traffic.

Despite the inherent inefficiency of an OLTP workload (where disk efficiency is typically 3–5% [28]), *Lachesis* can indirectly improve its performance when OLTP requests are interleaved with larger I/O activity. First, with the large I/Os being more efficient, the small random I/Os experience lower queueing times. Second, with explicit information about track boundaries, pages are always aligned. Thus, a single-page access never suffers a head switch (caused by accessing data on two adjacent tracks). Eliminating head switch (0.8 ms for the Atlas 10K disk in Table 1), however, provides only a limited improvement, because the small-random OLTP accesses are dominated by seek and rotational latency (averages of 5 and 3 ms respectively), as shown in Section 5.3.3.

3.3 Storage Performance Attributes

To maintain a clean system architecture while benefiting from *Lachesis*, a storage device's performance characteristics must be captured in a few well-defined device-independent attributes. This section describes two specific storage attributes, called ACCESS PARALLELISM and ACCESS DELAY BOUNDARIES that are relevant to DBMS.

ACCESS PARALLELISM exposes the inherent parallelism inside a storage device. It describes how many I/Os issued to a specific device can be serviced in parallel. For example, a disk drive can only service one request at a time, while a RAID-1 mirrored logical volume can service two reads in parallel. Thousands of read/write tips in a MEMS-based storage device, combined with exposed parallelism information, offer substantial performance improvements to table scans that require only subsets of attributes [32, 36]. Explicitly providing the (correct) level of parallelism to a storage manager is important for parallel sort and join algorithms [16] and proper data placement in parallel database systems [23].

ACCESS DELAY BOUNDARIES captures the track-alignment performance characteristic described in Section 2.3. Allocating and accessing data within these boundaries allows the storage device to offer the most efficient access. These units are also the natural choice for prefetch requests. For disk drives, the boundaries correspond to the sizes of each track. For RAID configurations, they correspond to stripe unit sizes. These boundaries teach the storage manager about the sizes and alignments that can be used to achieve near-streaming bandwidth even with interleaved I/O.

Ideally, a storage device would provide these attributes directly. Even though current storage interfaces do not provide them, they can usually be extracted by existing algorithms [31, 35] and tools [30]. Importantly, these boundaries cannot be approximated. If the values are incorrect, their use will provide little benefit and may even hurt performance. Therefore, a *Lachesis* storage manager pays a one-time cost of obtaining storage characteristics out-of-band (for instance, during volume initialization) rather than having a DBA set the values manually.

Our implementation, described in Section 4, focuses on utilization of the second attribute, which is crucial to dynamic changes to workloads with different query mixes. The ACCESS PARALLELISM attribute, on the other hand, is more important for data layout. A combination of these two attributes will result in additional performance improvements. We describe the details (as well as an implementation) of selective table scans utilizing the ACCESS PARALLELISM attribute elsewhere [32].

3.4 Lachesis Benefits

Lachesis has several advantages over current database storage manager mechanisms.

Simplified performance tuning. Since a *Lachesis* storage manager automatically obtains performance characteristics directly from storage devices, some difficult and error-prone manual configuration is eliminated. In particular, there is no need for hand-tuning such DB2 parameters as `EXTENTSIZE`, `PPREFETCHSIZE`, `DB2_STRIPED_CONTAINERS` or their equivalents in other DBMSes.

Minimal changes to existing structures. *Lachesis* requires very few changes to current storage manager structures. Most notably, the extent size must be made variable and modified according to the performance attribute values. However, decisions that affect other DBMS components, such as page size or pre-allocation for future appends of related data, are not affected; the DBMS or the DBA are free to set them as desired.

Preserving access costs across abstraction layers. *Lachesis* does not modify the optimizer's cost estimation functions that determine access pattern efficiency. In fact, one of its major contributions is ensuring that the optimizer-expected efficiency is preserved across the DBMS abstraction layers *and* materialized at the lowest level by the storage device.

Reduced buffer space pressure. With explicit delay boundaries, *Lachesis* can use smaller request sizes to achieve more efficient access. Figure 4 illustrates that smaller I/O requests allow for smaller buffers, freeing memory for other tasks. Despite sometimes requiring more smaller I/Os to finish the same job, the much increased efficiency of these smaller requests reduces the overall run time [31].

Lower inter-query interference. *Lachesis* consistently provides nearly streaming bandwidth for table scans even in the face of competing traffic. It can do so with smaller I/Os and still maintain high access efficiency. When there are several request streams going to the same device, the smaller size results in shorter response times for each individual request and provides better throughput for all streams (queries).

4 Lachesis Implementation

This section describes the key elements of a *Lachesis* prototype in the latest supported release (interim-release-2) of the Shore storage manager [5]. Shore consists of several key components, including a volume manager for storing individual pages, a buffer pool manager, lock and transaction managers, and an ARIES-style recovery subsystem [25]. Shore's basic allocation unit is called an extent and each extent's metadata, located at the beginning of the volume, identifies which of its pages are used.

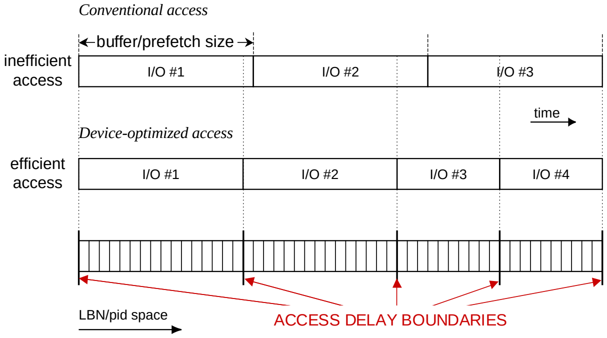

Figure 4: **Buffer space allocation with performance attributes.**

On-disk data placement. Shore's implementation assumes a fixed number of pages per extent (8 by default) and allocates extents contiguously in a device's logical block space. Therefore, the *LBN* (logical block number) and the `extnum` (extent number) for a given page, identified by a `pid` (page identifier), can be easily computed. To match allocation and accesses to the values of the ACCESS DELAY BOUNDARIES attribute, we modified Shore to support variable-sized extents. A single extent now consists of a number of pages and an additional unallocated amount of space, the size of which is less than one page.

Although page sizes are typically in powers of two, disk tracks are rarely sized this way (see Table 1). Thus, the amount of internal fragmentation (i.e., the amount of unallocated space at the end of each extent) can result in loss of some disk capacity. Fortunately, with an 8 KB page, internal fragmentation amounts to less than 2% and this trend gets more favorable as media bit density, and hence the number of sectors per track, increases.

We lookup the `extnum` and *LBN* values in a new metadata structure. This new structure contains information about how many *LBN*s correspond to each extent and is small compared to the disk capacity (for example, 990 KB for a 36 GB disk). Lookups do not represent significant overhead, as shown in Section 5.3.4.

Acquiring attribute values. To populate this new metadata structure, we added a system call to Shore that takes values for extent sizes during volume initialization and formatting. Currently, we determine the ACCESS DELAY BOUNDARIES attribute values out-of-band with the DIXtrac tool [30] and link a system call stub containing the attribute values with Shore. Conceptually, this is equivalent to a storage interface providing the information directly via a system call.

The DIXtrac tool supports two automatic approaches to detecting track boundaries: a general approach applicable to any disk interface supporting a READ command, and a specialized approach for SCSI disks. The general approach detects each track boundary by measuring response times of differently sized requests and detects the discontinuities illustrated in Figure 3, requiring about 4 hours for a 9 GB disk. The SCSI-specific approach uses query operations (e.g., TRANSLATE ADDRESS of the RECEIVE DIAGNOSTIC command) and expertise to determine complete mappings in under a minute regardless of disk size. A recent study [31] contains a detailed description of both algorithms.

I/O request generation. Shore's base implementation issues one I/O system call for each page read, relying on prefetching and buffering inside the underlying OS. We modified Shore to issue SCSI commands directly to the device driver to avoid double buffering inside the OS. We also implemented a prefetch buffer inside Shore. The new prefetch buffer mechanism detects sequential page accesses, and issues extent-sized I/Os. Thus, pages trickle from this prefetch buffer into the main buffer pool as they are requested by each page I/O request. For writes, a background thread in the base implementation collects dirty pages and arranges them into contiguous extent-sized runs. The runs, however, do not match extent boundaries; therefore, we increased the run size and divided each run into extent-sized I/Os aligned on proper extent boundaries.

Our modifications to Shore total less than 800 lines of C++ code, including 120 lines for the prefetch buffer. Another 400 lines of code implement direct SCSI access via the Linux `/dev/sg` interface.

5 Experiments

This section evaluates *Lachesis* using two sets of experiments. The first set of experiments replays modified I/O traces to simulate the performance benefits of *Lachesis* inside a commercial database system. The original traces were captured while running the TPC-C and TPC-H benchmarks on an IBM DB2 relational database system. The second set of experiments evaluates the *Lachesis* implementation inside Shore using TPC-C and (a subset of) TPC-H.

TPC-H [10], the decision-support benchmark, consists of 22 different queries, and two batch update statements. We ran all the queries in sequence, one at a time. Each query processes a large portion of the data. The TPC-C benchmark [9] emulates OLTP activity, measuring the number of committed transactions per minute. Each transaction involves a few read-modify-write operations to a small number of records.

For each of the two sets of experiments, we first show the performance of the TPC-H and TPC-C benchmarks run in isolation. We then look at the performance benefits *Lachesis* offers when both benchmarks run concurrently. In particular, we consider three scenarios:

No traffic simulates a dedicated DSS setup that runs only single-user TPC-H queries.

Light traffic simulates an environment with occasional background traffic introduced while executing the primary TPC-H workload. This represents a more realistic DSS setup with updates to data and other occasional system activity.

Heavy traffic simulates an environment with DSS queries running concurrently with a heavy OLTP workload. This represents a scenario when decision DSS queries are run on a live production system.

Finally, we contrast the results of the experiments with simulated Lachesis-DB2 and our implementation. The same trends in both cases provide strong evidence that other DBMS implementations using *Lachesis* are likely to obtain similar benefits.

5.1 Experimental Setup

We conducted all experiments on a system with a single 2 GHz Intel Pentium 4 Xeon processor, 1 GB of RAM, and a 36 GB Maxtor Atlas 10K III disk attached to a dedicated Adaptec 29160 SCSI card with 160 MB/s transfer rate. The basic parameters for this disk are summarized in Table 1. The system also included a separate SCSI host bus adapter with two additional disks; one with the OS and executables and the other for database logs. We ran our experiments on RedHat 7.3 distribution under Linux kernel v. 2.4.19, modified to include an I/O trace collection facility. For DB2 runs, we used IBM DB2 v. 7.2.

5.2 DB2 Trace Replay

We do not have access to the DB2 source code. To evaluate the benefits of *Lachesis* for DB2, we simulated its effect by modifying traces obtained from our DB2 setup. We ran all 22 queries of the TPC-H benchmark and captured their device-level I/O traces. We then wrote a trace-replay tool and used it to replay the original captured traces. Finally, we compared the trace replay time with the DB2 query execution time. The trace-replay method is quite accurate; the measured and replayed execution times differed by at most 1.5%.

The sum of all the periods between the completion time of the last outstanding request at the device and the issue time of the next request determined the *pure CPU time* from the original captured traces. This time expresses the periods when the CPU is busy, while the storage device is idling with no requests outstanding. As our goal is to improve I/O efficiency, we subtracted the pure CPU time from the traces.

Having verified that the original captured TPC-H traces never had more than two outstanding requests at the disk, we replayed the no-CPU-time traces in a closed loop by always keeping two requests outstanding at the disk. This ensures that the disk head is always busy (see Sec-

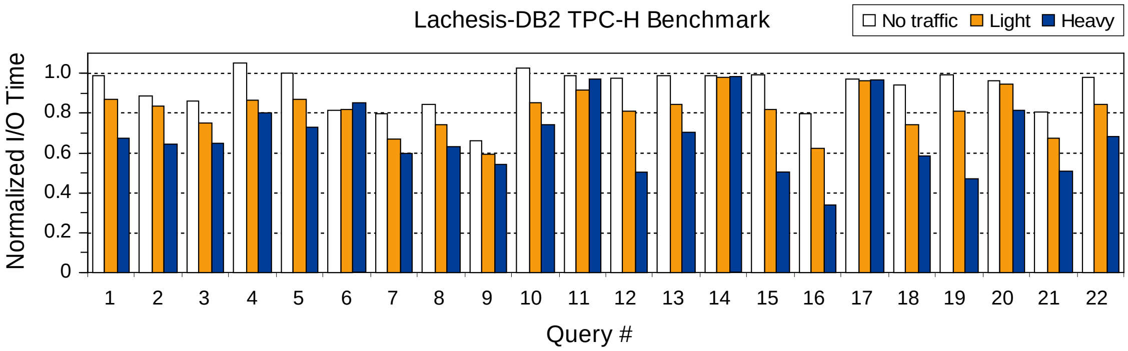

Figure 5: **TPC-H I/O times with competing traffic (DB2).**

tion 3.2) and preserves the order of request issues and completions [21]. Because of our trace replay method, the numbers reported in this section represent the query *I/O time*, which is the portion of the total query execution time spent on I/O operations.

To simulate *Lachesis* behavior inside DB2, we modified the DB2 captured traces by compressing back-to-back sequential accesses to the same table or index into one large I/O. We then split this large I/O into individual I/Os according to the values of the ACCESS DELAY BOUNDARIES attribute. Thus, the I/O sizes are an integral number of pages that fit within two adjacent boundaries.

The resulting modified traces preserve the sequence of blocks returned by the I/Os (i.e., no out-of-order issue and completion). Allowing requests to complete out of order might provide additional performance improvements due to request scheduling. However, we did not want to violate any unknown (to us) assumptions inside DB2 that require data on a single device to return in strictly ascending order.

The modified traces also preserve the DB2 buffer pool usage. The original DB2 execution requires buffers for I/Os of 768 blocks (determined by the `PREFETCHSIZE` parameter) whereas DB2 with *Lachesis* would generate I/Os of at most 672 blocks (i.e., a single track of the Atlas 10K III outermost zone accomodates 42 8 KB-pages).

5.2.1 TPC-H

We hand-tuned our DB2 configuration to give it the best possible performance on our hardware setup. We set the `PREFETCHSIZE` to 384 KB (48 pages × 8 KB page, or 768 blocks), which is comparable to the I/O sizes that would be generated by *Lachesis* inside DB2 running on the disk we used for our experiments[1]. We also turned on `DB2_STRIPED_CONTAINERS` to ensure proper ramp-up and prefetching behavior of sequential I/Os. We configured the DB2 TPC-H kit with the following parameters: scaling factor 10 (10 GB total table space), 8 KB pages, and a 768 MB buffer pool. We put the TPC-H tablespace on a raw device (a partition of the Atlas 10K III disk) to avoid double buffering inside Linux kernel.

The results of the DB2 TPC-H experiments are shown in Figure 5 with bars labeled *No traffic*. For each TPC-H query, the bar shows the resulting I/O time of the Lachesis-simulated trace normalized to the I/O time of the original trace (thus, shorter bars represent better performance).

With the exception of queries 4 and 10, whose run times were respectively 4% and 1% longer, all queries benefited. Queries that are simple scans of data (e.g., queries 1, 4, 15) in general do not benefit; the original DB2 access pattern already uses highly efficient large sequential disk accesses thanks to our manual performance tuning of the DB2 setup. The minor I/O size adjustments of these sequential accesses cause small changes in performance (e.g., 1% improvement for queries 1 and 15 and 4% slowdown for query 4). On the other hand, queries that include multiple nested joins, such as query 9, benefited much more (i.e., 33% shorter execution time or 1.5× speedup) because of inherently interleaved access patterns. Interestingly, such queries are also the most expensive ones. On average, the 22 queries in the workload experienced an 11% speedup.

The access pattern of query 10 is dominated by runs of 2–4 sequential I/O accesses (sized at `EXTENTSIZE` of 768 blocks). At the end of each run, the disk head seeks to another nearby location a few cylinders away and performs another short sequential run. The *Lachesis*-simulated trace, however, transforms the sequential runs into 3–5 I/Os (the track size is at most 686 blocks) with the last being less than full-track in size.

Because of data dependencies in queries with several nested joins (e.g., queries 9 or 21), the I/O accesses were not purely sequential. Instead, they contained several interleaved data and index scans. Even when executing such queries one at a time, these interleaved sequential accesses in effect interfered with each other and caused additional seek and rotational delays. *Lachesis* mitigated this adverse effect, resulting in significant performance improvements.

[1] `PREFETCHSIZE` setting of 512 KB triggers a Linux kernel “feature”: a single `PREFETCHSIZE`-d I/O to the raw device generated by DB2 was broken into two I/Os of 1023 and 1 block. Naturally, this results in highly inefficient accesses. Therefore, we chose 768 instead.

The plans for queries 17 and 20 include two nested-loop joins and index scans. Hence, most I/Os are small random requests of 1–2 pages; the limited performance improvement of *Lachesis* comes from the elimination of head-switches with delay boundary-aligned accesses, just like in the OLTP experiments described below.

5.2.2 TPC-C

Since *Lachesis* targets track-sized I/Os, we do not expect any benefit to small random I/Os stemming from an OLTP workload. To ensure that *Lachesis* does not hurt TPC-C performance we captured I/O traces on our DB2 system running the TPC-C benchmark, applied the same transformations as for the TPC-H workload, and measured the trace replay time. Eliminating CPU time was not necessary because there were no storage device idle periods in the trace.

The DB2 configuration for the TPC-C benchmark is identical to the one described in Section 5.2.1 (8 KB pages, a 768 MB buffer pool, a raw device partition of the Atlas 10K III disk holding the TPC-C data and indexes). We used the following parameters for the TPC-C benchmark: 10 warehouses (approximately 1 GB of initial data), 10 clients per warehouse, zero keying/think time. As expected, the experiments showed that *Lachesis* does not affect TPC-C performance.

5.2.3 Compound Workload

To demonstrate *Lachesis*' ability to increase I/O efficiency under competing traffic, we simulated the effects of running TPC-C simultaneously with TPC-H queries by injecting small 8 KB random I/Os (a reasonable approximation of TPC-C traffic) into the disk traffic during the TPC-H trace replay. We used a Poisson arrival process for the small-random I/O traffic and varied the arrival rate between 0 and MAX arrivals per second. Using our hardware setup, we determined MAX to be 150 by measuring the maximal throughput of 8 KB random I/Os issued one-at-a-time.

The results are shown in Figure 5. As in the *No traffic* scenario, we normalize the *Lachesis* runs to the base case of replaying the no-CPU-time original traces. However, since there is additional traffic at the device, the absolute run times increase (see Section 5.4 for details). For the *Light traffic* scenario, the arrival rate λ was 25 arrivals per second, and for the *Heavy traffic* scenario λ was 150. Under *Heavy traffic*, the original query I/O times varied between 19.0 and 1166.2 s, yielding an average 2.6× increase in I/O time compared to *No traffic*.

The Lachesis-modified traces exhibit substantial improvement in the face of competing traffic. Further, the relative value grows as the amount of competing traffic increases, indicating the *Lachesis*' robustness to competing traffic. On average, the improvement for the *Light traffic* and *Heavy traffic* scenarios was 21% (or 1.3× speedup) and 33% (1.5× speedup) respectively. Query 16 experienced the highest improvement, running 3 times faster in the *Heavy traffic* scenario. With the exception of queries 6, 11, 14, and 17, which benefit little or none, all other queries benefit the most under the *Heavy traffic* scenario.

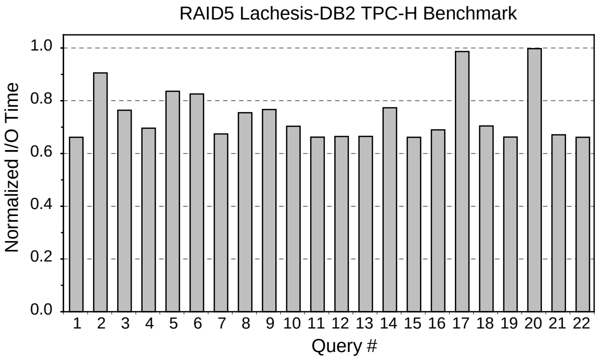

Figure 6: **TPC-H trace replay on RAID5 configuration (DB2).**

5.2.4 TPC-H on Disk Arrays

To evaluate the benefit of having explicit performance attributes from disk arrays, we replayed the captured DB2 traces against a disk array simulated by a detailed storage subsystem simulator, called DiskSim [15]. We created a logical volume on a RAID5 group with 4 disks configured with validated Atlas 10K III characteristics [30].

In the base case scenario, called *Base-RAID5*, we set the stripe unit size to 256 KB (or 512 disk blocks) and fixed the I/O size to match the stripe unit size. This value approximates the 584 sectors per track in one of the disk's zones and, as suggested by Chen [6], provides the best performance in the absence of exact workload information. In the second scenario, called *Lachesis-RAID5*, both the RAID controller and the database storage manager can explicitly utilize the precise track-size, and therefore both the stripe unit and I/O sizes are equal to 584 blocks.

The resulting I/O times of the 22 TPC-H queries, run in isolation without any competing traffic, are shown in Figure 6. The graph shows the Lachesis-RAID5 time normalized to the Base-RAID5. Comparing this with the TPC-H runs on a single disk, we immediately notice a similar trend. Queries 17 and 20 do not get much improvement. However, most queries enjoy more significant improvement (on average 25%, or 1.3× speedup) than in the single disk experiments.

The performance benefits in the RAID experiments are larger because the parity stripe, which rotates among the four disks, causes a break in sequential access to each individual's disk in the *Base-RAID5*. This is not a problem, however, in the *Lachesis-RAID5* case, which achieves efficiency close to streaming bandwidth with track-sized stripe units. Even larger benefits are achieved for compound workloads.

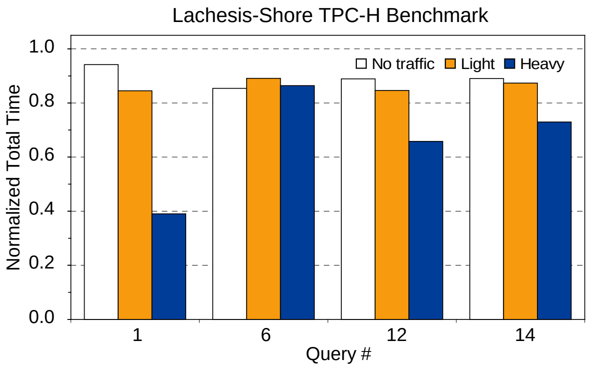

Figure 7: **TPC-H queries with competing traffic (Shore).**

	Basic Shore		*Lachesis Shore*	
Clients	**TpmC**	**CPU**	**TpmC**	**CPU**
1	844	34%	842	33%
3	1147	46%	1165	45%
5	1235	50%	1243	49%
8	1237	53%	1246	51%
10	1218	55%	1235	53%

Table 2: **TpmC (transactions-per-minute) and CPU utilization.** The slightly better throughput for the Lachesis-Shore implementation is due to proper alignment of pages to track boundaries.

5.3 Lachesis Implementation Experiments

We compared the performance of our implementation, called *Lachesis-Shore* and described in Section 4, to that of baseline Shore. For fair comparison, we added (one-extent) prefetching and direct SCSI to the Shore interim-release 2 and call it *Basic-Shore*. Unless stated otherwise, the numbers reported in the following section represent an average of 5 measured runs of each experiment.

5.3.1 TPC-H

Our Shore TPC-H kit (obtained from earlier work [1]) implements queries 1, 6, 12, and 14. We used a scaling factor of 1 (1 GB database) with data generated by the `dbgen` program [10]. We used an 8 KB page size, a 64 MB buffer pool, and the default 8 pages per extent in Basic-Shore. The Lachesis-Shore implementation matches an extent size to the device characteristics, which, given the location of the volume on the Atlas 10K III disk, varied between 26 and 24 pages per extent (418 and 396 disk blocks). Figure 7 shows the normalized total run time for all four TPC-H queries implemented by the TPC-H kit. *Lachesis* improved run times between 6% and 15% in the *No traffic* scenario.

5.3.2 TPC-C

To ensure that *Lachesis* does not hurt the performance of small random I/Os in OLTP workloads, we compared the TPC-C random transaction mix on our Basic- and Lachesis-Shore implementations configured as described in Section 5.3.1. We used 1 warehouse (approximately 100 MB of initial data) and varied the number of clients per warehouse. We set the client keying/think time to zero and measured the throughput of 3000 transactions. As shown in Table 2, our implementation had minimal effect on the performance of the standalone TPC-C benchmark.

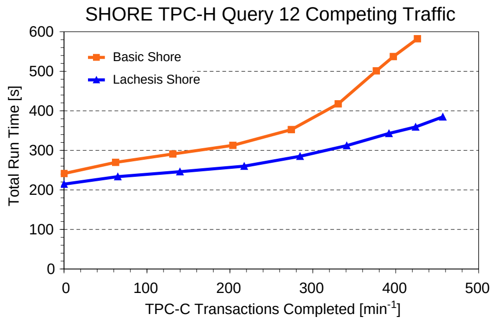

Figure 8: **TPC-H query 12 execution time as a function of TPC-C competing traffic (Shore).**

5.3.3 Compound Workload

We modeled competing device traffic for DSS queries of the TPC-H benchmark by running a TPC-C random transaction mix. Due to the limitations of the Shore TPC-H and TPC-C kit implementations, we could not run the TPC-H and TPC-C benchmarks in the same instance. Thus, we ran two instances whose volumes were located next to each other on the same disk. Because the volumes occupied a small part of the disk, this experiment approximates the scenario of OLTP and DSS workloads accessing the same database.

The TPC-H instance was configured as described in Section 5.3.1 while the TPC-C instance was configured with 1 warehouse and 1 client per warehouse, which ensured that no transactions were aborted due to resource contention or deadlock. We varied the amount of background OLTP traffic by changing the keying/think time of the TPC-C benchmark to achieve a rate of 0 to TpmC$_{MAX}$ (maximum transactions per minute).

Figure 7 shows the performance results for Lachesis-Shore, normalized to the Basic-Shore execution time. As with the DB2 trace replay experiments, *Lachesis* provides higher speedups in the face of competing traffic, than under the *No traffic* scenario. Additional experiments show that *Lachesis*' relative improvement increases as a function of the amount of competing traffic. The average improvement for the four TPC-H queries under the *Light traffic* and *Heavy traffic* scenarios was 14% (or 1.2× speedup) and 32% (1.5× speedup) respectively.

An important result of this experiment is shown in Figure 8. This figure compares the absolute run times for each query as a function of increasing transactional throughput. The two Shore implementations achieve different TpmC$_{MAX}$. While for the Basic-Shore implementation TpmC$_{MAX}$ was 426.1, Lachesis-Shore achieved 7% higher maximal throughput (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 under the *Heavy traffic* scenario.

5.3.4 Extent Lookup Overhead

Since a *Lachesis* implementation uses variable size extents, we also wanted to evaluate the potential overhead of `extnum` and *LBN* lookup. Accordingly, we configured our Lachesis-Shore implementation with extents of uniform size of 128 blocks to match the default 8-page extent size in the Basic-Shore implementation and ran the four TPC-H queries. In all cases, the difference was less than 1% of the total runtimes. Thus, explicit lookup, instead of a simple computation from the page `pid` in Basic-Shore, does not result in a noticeable slowdown.

5.4 Comparing DB2 and Shore

The results for compound workloads with the Lachesis-Shore implementation and the Lachesis-modified DB2 traces show similar trends. For example, as the competing traffic to queries 1 and 12 increases, the relative performance benefit of *Lachesis* increases as well. Similarly, the relative improvement for query 6 remained stable (around 20%) under the three scenarios for both DB2 and Shore.

Two important differences between the DB2 and Shore experiments warrant a closer look. First, under the *No traffic* scenario, Lachesis-Shore experienced bigger speedup. This is because the Basic-Shore accesses are less efficient than the accesses in the original DB2 setup. Basic-Shore uses 8-page extents, spanning 128 blocks, compared to *Lachesis*' variable-sized extents of 418 and 396 blocks (track size sizes of the two inner-most zones). DB2, on the other hand, prefetched 768 disk blocks in a single I/O, while *Lachesis*-modified traces used at most 672 blocks (the outer-most zone has 686 sectors). Consequently, the base case for DB2 issues more efficient I/Os relative to its Shore counterpart, hence the relative improvement for Lachesis-Shore is higher.

Second, TPC-H query 14 with DB2 trace replay did not improve much, whereas Lachesis-Shore's improvement grew with increasing traffic. The reason lies in the different access patterns resulting from different join algorithms. While DB2 used a nested-loop join with an index scan and intermediate sort of one of its inputs, Lachesis-Shore used hash-join. Hence, we see a higher improvement in Lachesis-Shore, whereas in DB2 trace replay, the improvement does not change.

Finally, we compared the run times for TPC-H query 12 in Figure 8 against Figure 1, whose access patterns are relatively straightforward. This query first scans through the LINEITEM table applying all predicates, and then performs a join against the ORDERS table data. Although the x-axes use different units, and hence the shapes of the curves are different, the trends in those two figures are the same. With small amounts of competing traffic, the relative improvement of *Lachesis* is small. However, as the amount of competing traffic increases, the speedup grows to 1.5× for Shore and 2× for DB2 under the *Heavy traffic* scenario.

6 Summary

This paper describes a design and a prototype implementation of a database storage manager that provides robust performance in the face of competing traffic. By automatically extracting and utilizing high-level device-specific characteristics, *Lachesis* ensures efficient I/O execution despite competing traffic. For compound workloads and complex DSS queries, the result is a substantial (up to 3×) performance improvement.

Acknowledgements

We thank Mengzhi Wang and Jose-Jaime Morales for providing and helping with TPC-C and TPC-H toolkits for Shore, and Gary Valentin from IBM Toronto for helping us understand the intricacies of DB2 configuration and for explaining to us the workings of the DB2 storage manager. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This work is funded in part by NSF grants CCR-0113660, IIS-0133686, and CCR-0205544.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. *International Conference on Very Large Databases* (Rome, Italy, 11–14 September 2001), pages 169–180. Morgan Kaufmann Publishing, Inc., 2001.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? *International Conference on Very Large Databases* (Edinburgh, UK, 7–10 September 1999), pages 266–277. Morgan Kaufmann Publishing, Inc., 1999.

[3] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: making the fast case common. *Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[4] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: memory access. *International Conference on Very Large Databases* (Edinburgh, UK, 07–10 September 1999), pages 54–65. Morgan Kaufmann Publishers, Inc., 1999.

[5] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael 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.

[6] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. *ACM International Symposium on Computer Architecture* (Seattle, WA), pages 322–331, June 1990.

[7] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B+-trees: optimizing both cache and disk performance. *ACM SIGMOD International Conference on Management of Data* (Madison, WI, 03–06 June 2002), pages 157–168. ACM Press, 2002.

[8] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, **14**(3):225–264. ACM Press, August 1996.

[9] Transactional Processing Performance Council. TPC Benchmark C. Number Revision 5.1.0, 2002.

[10] Transactional Processing Performance Council. TPC Benchmark H. Number Revision 2.0.0, 2002.

[11] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. *Summer USENIX Technical Conference* (Monterey, CA, 10–15 June 2002), 2002.

[12] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-IHsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, **2**(1):44–62, March 1990.

[13] Craig S. Freedman, Josef Burger, and David J. DeWitt. SPIFFI-a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, **7**(11):1185–1200, November 1996.

[14] Gregory R. Ganger. *Blurring the line between OSs and storage devices*. Technical report CMU–CS–01–166. Carnegie Mellon University, December 2001.

[15] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim simulation environment version 1.0 reference manual*, Technical report CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[16] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, **25**(2):73–170, June 1993.

[17] IBM Corporation. *IBM DB2 Universal Database Administration Guide: Implementation*, Document number SC09-2944-005, 2000.

[18] IBM Corporation. *DB2 V3 Performance Topics*, Document number GC24-4284-005, August 1994.

[19] Orran Krieger and Michael Stumm. HFS: a performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, **15**(3):286–321. ACM Press, August 1997.

[20] Kevin Loney and George Koch. *Oracle 8i: The Complete Reference*. Osborne/McGraw-Hill, 2000.

[21] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 275–288. USENIX Association, 2002.

[22] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. *International Conference on Very Large Databases* (Hong Kong, China, 20–23 August 2002), pages 191–202. Morgan Kaufmann Publishers, Inc., 2002.

[23] Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, **6**(1):53–72, February 1997.

[24] Rodney Van Meter. Sleds: storage latency estimation descriptors. *IEEE Symposium on Mass Storage Systems* (Greenbelt, MD, 23–26 March 1998). USENIX, 1998.

[25] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, **17**(1):94–162, March 1992.

[26] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB SCSI product manual*, Document number 81-119313-05, August 1999.

[27] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. *International Conference on Very Large Databases* (Edinburgh, UK, 07–10 September 1999), pages 78–89. Morgan Kaufmann Publishers, Inc., 1999.

[28] Erik Riedel, Christos Faloutsos, Gregory R. Ganger, and David F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD International Conference on Management of Data* (Dallas, TX, 14–19 May 2000), pages 13–21, 2000.

[29] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Symposium on Operating System Principles* (Pacific Grove, CA, 13–16 October 1991). Published as *Operating Systems Review*, **25**(5):1–15, 1991.

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

[31] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory 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.

[32] Steven W. Schlosser, Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. *Exposing and exploiting internal parallelism in MEMS-based storage*. Technical Report CMU–CS–03–125. March 2003.

[33] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. *ACM SIGMOD International Conference on Management of Data* (Boston, MA, 1979), pages 23–34. ACM Press, 1979.

[34] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggreggate file I/O latency. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):252–263. ACM, 1997.

[35] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.

[36] Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. *Tabular placement of relational data on MEMS-based storage devices*. UCSB Department of Computer Science Technical Report 2003-06. March 2003.