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

Jiri Schindler Anastassia Ailamaki Gregory R. Ganger

March 2003

CMU-CS-03-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report also appears in a revised form in the Conference on Very Large Databases (VLDB03), Berlin, Germany, Sept. 9-12, 2003.

Abstract

Database systems work hard to tune I/O performance, but they do not always achieve the full performance potential of modern disk drives. 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 a few observable device-specific characteristics to achieve more robust performance. Most notably, it enables efficiency nearly equivalent to sequential streaming even in the presence of competing I/O traffic. With automatic adaptation to device characteristics, Lachesis simplifies manual configuration and restores optimizer assumptions about the relative costs of different access patterns expressed in query plans. Based on experiments with both IBM DB2 and an implementation inside the Shore storage manager, Lachesis improves performance of TPC-H queries on average by 10% when running on dedicated hardware. More importantly, it speeds up TPC-H by up to 3× when running concurrently with an OLTP workload, which is simultaneously improved by 7%.

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.

Keywords: Database storage management, Performance evaluation

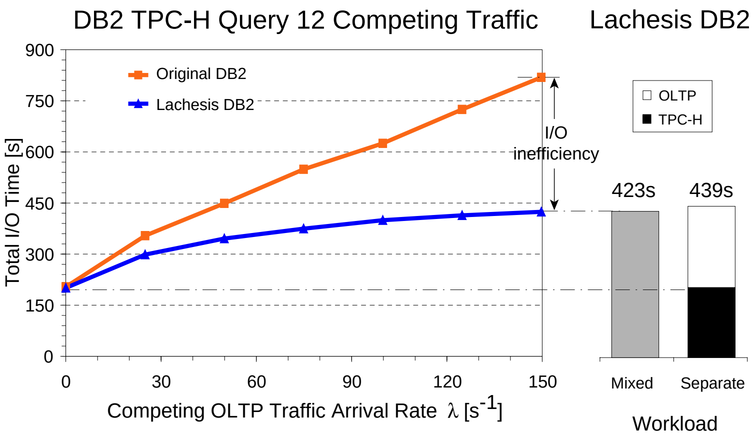

Figure 1: **TPC-H query 12 execution on DB2.** The graph on the left shows the amount of time spent in I/O operations as a function of increasing competing OLTP workload, simulated by random 8 KB I/Os with arrival rate λ. The I/O inefficiency in the original DB2 case is due to extraneous rotational delays 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 adds the total I/O time for the TPC-H and the OLTP-like workloads run separately.

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 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.

For many years, the high cost of I/O operations has been a dominant factor of query execution. Thus, a main 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 abstraction inside the DBMS, as well as between the storage manager and the storage devices, make it difficult to accurately quantify the efficiency of different access patterns. 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. Combined, these two factors lead to loss of I/O performance and longer query execution times, as illustrated by the *Original DB2* line in Figure 1.

To regain the lost performance, we propose a new storage manager architecture, called *Lachesis*. 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 sequential I/O performance, as illustrated by the bars in Figure 1. *Lachesis* simplifies the task of DBMS configuration and performance tuning. By automatically detecting and transparently exploiting device characteristics, it eliminates the need for several parameters that were previously manually set by a DBA. Most importantly, it restores the validity of the assumptions made by the query optimizer about the relative costs of different storage access patterns.

For example, in Figure 1, *Lachesis* ensures that TPC-H query 12 (dominated by sequential table scan), enjoys streaming I/O performance, as anticipated by the optimizer, regardless of the rate of concurrent OLTP-like I/O traffic. A storage manager could choose to favor one query over another, but neither will suffer inefficiency by mixing them together.

This paper describes an implementation of the *Lachesis* architecture within the Shore storage manager [4]. It involves three parts: extraction of characteristics, modifications to data placement algorithms, and modifications to prefetch and buffer management policies. The extraction, which runs once before data are placed on a new device, uses test patterns [34] to identify track boundaries [30]. 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 this prototype implementation and DB2 trace replay simulations showed modest performance improvements for stand-alone workloads. Decision support workloads (DSS) improved on average by 10% (with a maximum of 33%) and on-line transaction processing (OLTP) workloads were unaffected. When running compound workloads, however, Lachesis exhibited up to a *three-fold* performance improvement on DSS queries, while simultaneously improving OLTP throughput by 7%.

The rest of this paper is organized as follows. Section 2 overviews current methods in database systems that help in query optimization and execution. Section 3 describes the *Lachesis* architecture. Section 4 describes our implementation inside the Shore storage manager. Section 5 evaluates our design on a state-of-the art disk drive.

2 Background and Related Work

The architecture of a typical DBMS includes two major components relevant to the topic of this paper: the query optimizer and the execution unit, which includes the storage manager. As illustrated in Figure 2, the optimizer takes individual queries and determines a plan for their execution by evaluating the cost of each alternative and selecting the one with the lowest cost. The execution unit accepts the chosen query plan and allocates resources for its execution, with the storage manager generating accesses to storage when needed.

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 [31] collected by the storage manager to estimate the number of I/O operations executed by each algorithm [13] (e.g., loading a page from a storage device into the buffer pool or writing a page back to the device). Because of the physical characteristics of disk drives, random I/O is significantly less efficient than sequential access.

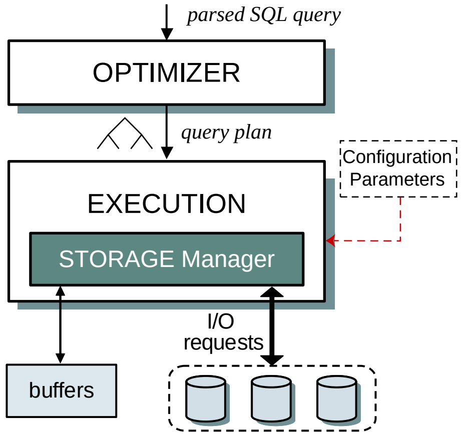

Figure 2: **Relevant components of a database system.**

Therefore, today's commercial DBMS optimizers also consider the data access pattern dictated by a particular query operator, employing cost models that distinguish between random and sequential I/O.

Although this differentiation captures the most important disk performance feature, it alone does not guarantee that the efficiencies of the respective access patterns are achieved during execution. Thus, the execution unit must be designed to uphold the assumed performance across disk types and concurrent query execution.

2.2 Ensuring Efficient I/O Execution

Transforming query plans into individual I/O requests requires the storage manager to make decisions about request sizes, locations, and the temporal relationship to other requests (i.e., scheduling). While doing so, a storage manager considers resource availability and the level of contention for them caused by other queries running in parallel. The decisions are also influenced by several manually configured parameters set by a DBA. For example, IBM DB2's `EXTENTSIZE` and `PREFETCHSIZE` parameters determine the maximal size of a single I/O operation [14], and the `DB2_STRIPED_CONTAINERS` parameter instructs the storage manager to align I/Os on stripe boundaries.

Clearly, the storage manager has a difficult task. It 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 the device-specific characteristics and dynamic query mixes. This results in inefficiency and performance degradation.

Recent efforts have proposed storage mechanisms that exploit freedom to reorder storage accesses. Storage latency estimator descriptors estimate the latency for accessing the first byte of data and the expected bandwidth for subsequent transfer [23]. Steere proposed a construct, called a set iterator, that exploits asynchrony and non-determinism in data accesses to reduce aggregate I/O latency [32]. The River projects use efficient streaming from distributed heterogenous nodes to maximize I/O performance for data-intensive applications [2, 20]. Other research exploited simi-

			Head	Avg.	Sectors	No. of		
Disk	**Year**	**RPM**	**Switch**	**Seek**	**Per Track**	**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
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.

lar ideas at the file system level to achieve higher I/O throughput from the underlying, inherently parallel, hardware [16]. Our goal is to achieve maximum storage efficiency within the traditional DBMS structure.

2.3 Exploiting Observed Storage Characteristics

To achieve more robust performance, storage managers need to exploit observed storage system performance characteristics. DBAs cannot be expected to get detailed tuning knobs right, and high-level knobs do not provide sufficient information to the storage manager. 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, these methods are error-prone and often yield sub-optimal results (see Section 5.5 for an example).

The alternate approach promoted in this paper is for the storage manager to understand a bit more about underlying device characteristics. With storage devices that provide relevant performance characteristics directly, a storage manager can automatically adjust its access patterns to generate I/Os that are most efficient for the device. Such information could also be acquired automatically by experimenting with the storage device before loading data sets onto it. For example, Worthington et al. describe algorithms for online extraction of parameters from SCSI disk drives [34]. An automated tool called DIXtrac [29] extends these algorithms to extract detailed disk characteristics in a few minutes, including complete logical-to-physical mappings, mechanical overheads, and caching/prefetching algorithms.

Many research efforts have proposed exploiting low-level disk characteristics to improve performance. Ganger [11] and Denehy et al. [9] 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 [30] 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.

Years ago, database systems exploited disk characteristics. For example, the Gamma system [10] accessed data in track-sized I/Os by simply knowing the number of sectors per track. Unfortunately, simple mechanisms like this are no longer possible without the architectural changes proposed here, because of high-level device interfaces and built-in firmware functions. For example, zoned geometries and advanced defect management in current disks (e.g., Quantum Atlas 10k [25]), result in cylinders being composed of tracks with variable number of sectors (see Table 1). No single value for the number of sectors per track is correct across the device.

2.4 Exploiting Memory Hierarchy Characteristics

For compute and memory-intensive database applications, memory latency is a big issue [3]. Thus, some recent research focuses on improving utilization of processor cache characteristics by in-page partitioning of data [1] and cache-friendly indexing structures [6, 26]. Another approach calculates access costs based on data access patterns and hardware parameters such as cache access time, size, and associativity [19]. These costs are then incorporated into the functions that evaluate alternative query plans.

Unfortunately, these techniques do not suffice for tuning and estimating disk access costs. I/O access times, and in particular seek and rotational latency, are compex functions of spatial and temporal request inter-relationships. Therefore, a more dynamic solution is needed to optimize I/O access while executing a database query.

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 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 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 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 ex-

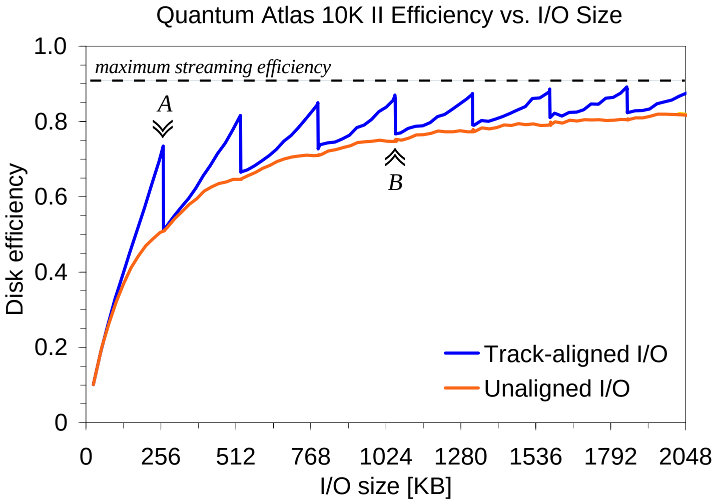

Figure 3: **Disk access efficiency.** This graph plots disk efficiency as a function of I/O size for track-unaligned and track-aligned random requests within disk's first zone. *Disk efficiency* is the fraction of total access time spent moving data to or from the media. The streaming efficiency is 0.9 because of head switches between adjacent tracks. The peaks in the track-aligned curve correspond to multiples of the first zone's track size.

isting storage manager structures, which call these groups segments, or extents [14, 17]. Hence, implementing *Lachesis* requires only minimal changes, as discussed in Section 4.

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*, the disk efficiency increases with increasing 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 [15, 28]. However, this creates a tension between increased efficiency and higher demand for buffer space.

As observed before [30], track-aligned access for the same set of random requests can be much more efficient as shown in Figure 3 by the line labeled *Track-aligned I/O*. Point *A* highlights the higher efficiency of track-aligned access (0.73, or 82% of the maximum) over 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*, with I/O size 4 times as large, 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 (see Table 1), making it a robust choice for automatically sizing 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 [25]. 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 without unduly penalizing any of the ongoing queries.

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

Despite the inherent inefficiency of an OLTP workload (where disk utilization is typically 3–5% [27]), *Lachesis* can indirectly improve its performance as well as when OLTP requests are interleaved with larger I/O activity. First, with higher efficiency of the large I/Os, the small random I/Os experience smaller queue 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) during data transfer.

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. Our implementation, described in Section 4, however, only utilizes the second attribute.

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. This attribute is useful in determining the appropriate level of parallelism for different parallel sort and join algorithms [13, 21] and proper data placement in parallel database systems [22].

ACCESS DELAY BOUNDARIES captures the track-alignment performance characteristic described in Section 2.3. By allocating and accessing data within these boundaries, the storage device offers 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 [30, 34] and tools [29]. 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 (e.g., during volume initialization) rather than having a DBA set the values manually.

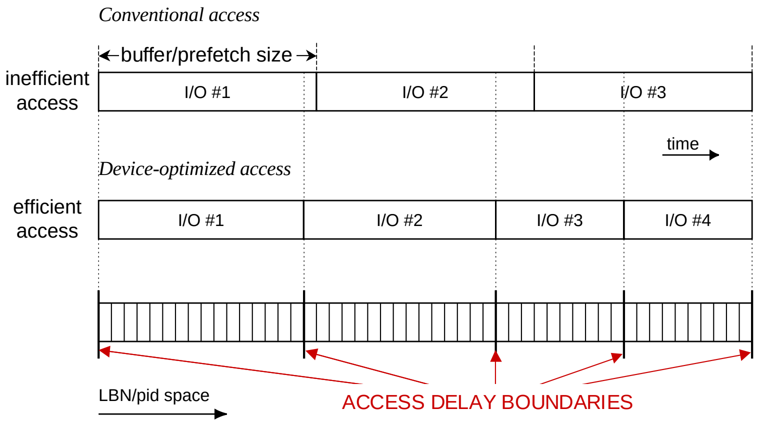

Figure 4: **Buffer space allocation with performance attributes.**

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. The DBA can concentrate on other important configuration issues.

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 is free to set them accordingly as desired.

Preserving access costs across abstraction layers. *Lachesis* does not modify the optimizer's cost estimator 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 efficient access. This allows smaller buffers, freeing memory for other tasks as illustrated in Figure 4.

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 [4]. 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 [24]. 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.

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 some number of pages and an additional unallocated amount of space less than one page in size.

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 (e.g., 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 [29] 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 and takes about 4 hours for a 9 GB disk (Note, however, that the total runtime is not a function of capacity, but rather the track size). The SCSI-specific approach uses query operations (e.g., TRANSLATE ADDRESS of the RECEIVE DIAGNOSTIC command) and expertise to determine compete mappings in under a minute regardless of disk size. Both algorithms are detailed elsewhere [30, 33].

I/O request generation. We also augmented Shore's page I/O policies. Curiously, the 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. However, the runs do not match extent boundaries. We increased the run size and divided each run into extent-sized I/Os.

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 `/dev/sg` interface.

5 Experiments

This section evaluates the *Lachesis* architecture. A 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. A second set of experiments measures our Lachesis implementation running TPC-C and (a subset of) TPC-H.

TPC-H [8] consists of 22 different queries, and two batch update statements, representative of a decision support system workload. We ran the queries in sequence: a single query at a time. Each query processes a large portion of the data. The TPC-C benchmark [7] 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 and runs 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 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 similar trends in both cases provide strong evidence that similar benefits can be obtained in other DBMSes.

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 that included an I/O trace collection facility. For capturing the I/O traces, 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 replayed the original captured traces using a trace-replay tool we wrote and compared their 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%.

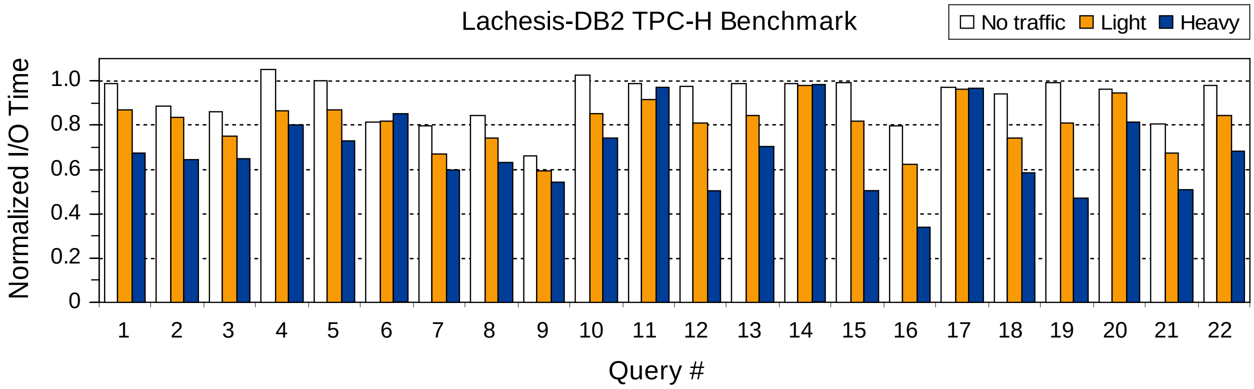

Figure 5: **DB2 TPC-H I/O times given differing amounts of competing traffic.**

Adding up all 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. Since our focus is on improving I/O efficiency, we subtracted the pure CPU time from the traces to be able to make a more direct comparison.

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, always keeping two requests outstanding at the disk. This ensures that the disk head is always busy and preserves the order of request issues and completions [18]. 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 thus 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., 42 8 KB-pages that fit into a single track of the outermost zone of the Atlas 10k III disk).

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 TPC-H benchmark are shown in Figure 5 and 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. The original query times varied between 7.0 s (for query 22) and 507.9 s (for query 9).

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 such as query 1, saw only a small benefit; the original DB2 access pattern already uses highly efficient large sequential disk accesses thanks to our manual performance tuning of the DB2 setup. On the other hand, queries that include multiple nested joins, such as query 9, benefited much more (i.e., 33% shorter or 1.5× speedup). Interestingly, such queries are also the most expensive ones. On average, the 22 queries experienced an 11% speedup and completed in 3888 s vs. 4354 s.

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 acceses in effect interfered with each other and caused additional seek and rotational delays. With *Lachesis*, this adverse effect was mitigated, resulting in large performance improvements. These improvements are similar to the results with compound workloads presented below.

5.2.2 TPC-C

Since *Lachesis* targets track-sized I/Os, we do not expect to see any benefit to small random I/Os stemming from an OLTP workload. To ensure that Lachesis does not introduce a slowdown to the TPC-C benchmark, 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 was 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 1GB of initial data), 10 clients per warehouse, zero keying/think time.

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

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

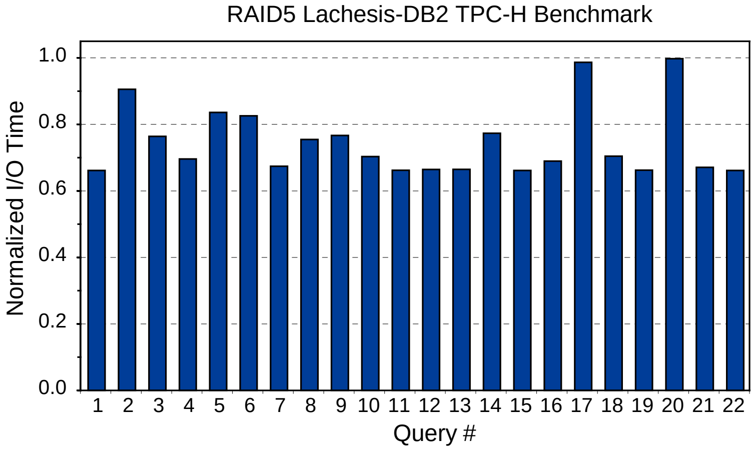

Figure 6: **DB2 TPC-H trace replay on RAID5 configuration.**

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 socond, 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*' resiliency 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 saw little or no benefit, all other queries saw greater benefit under the *Heavy traffic* scenario.

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 arrray simulated by a detailed storage subsystem simulator, called DiskSim [12]. We created a logical volume on a RAID5 group with 4 disks configured with validated Atlas 10k III characteristics [29].

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 [5], 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

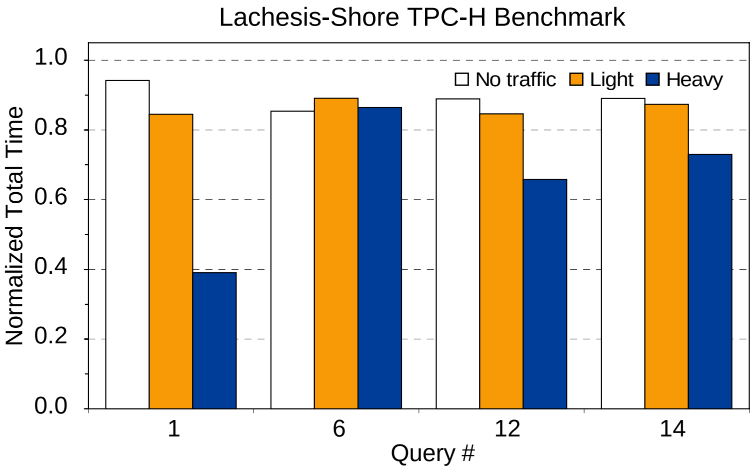

Figure 7: **Shore TPC-H competing traffic.**

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.

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

The Shore TPC-H kit derived 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 [8]. We used an 8 KB page size, a 64 MB buffer pool, and the default 8 pages per extent in the Basic-Shore implementation. 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 TPC-H queries 1, 6, 12, and 14. *Lachesis* improved run times between 6% and 15% in the *No traffic* scenario.

5.3.2 TPC-C

To ensure that *Lachesis* does not harm the performance of small random I/Os in OLTP workloads, we compared the TPC-C random transaction mix on our Basic- and Lachesis-Shore implemen-

	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 and CPU utilization.** The slightly better throughput for the Lachesis-Shore implementation is due to proper alignment of pages to track boundaries.

tations 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 can be seen in Table 2, our implementation had no effect on the TPC-C benchmark performance when run in isolation.

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. Because of the TPC-H and TPC-C kit limitations, we could not run both benchmarks in the same Shore 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 serves as an approximation to the situation of both OLTP and DSS accessing the same tables.

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 the background traffic to the DSS queries by changing the keying/think time of the TPC-C benchmark to achieve a rate of 0 to TpmC_{MAX}.

The results for Lachesis-Shore are shown in Figure 7 and normalized to the Basic-Shore as described earlier. As with the DB2 trace replay experiments, Lachesis provides more substantial improvement in the face of competing traffic, compared to the *No traffic* scenario. Similarly, as the amount of competing traffic increased, the relative improvement grew as well. On average, the improvement for these four 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 side-by-side 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}=426.1, Lachesis-Shore achieved 7% higher maximal throughput of 456.7 transactions per minute. Thus, Lachesis not only improved the performance of the TPC-H queries alone, but also offered higher peak performance to the TPC-C workload under the *Heavy traffic* scenario.

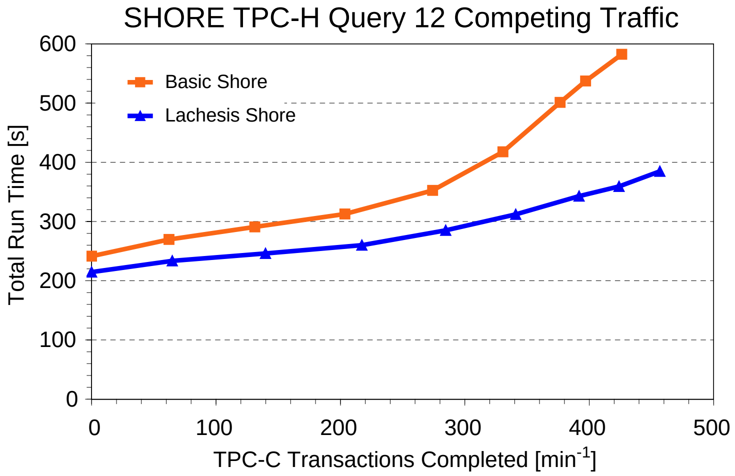

Figure 8: **Shore TPC-H query 12 execution with competing traffic.**

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, the added work of doing explicit lookup, rather than a simple computation from the page `pid` in Basic-Shore, does not result in any observable 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 amount of competing traffic to queries 1 and 12 increased, the relative performance benefit of *Lachesis* increased. Similarly, the relative improvement for query 6 stayed the same (around 20%) under the three scenarios both for DB2 and Shore. Unfortunately, we cannot compare the queries with the most interesting access patterns (e.g., query 9 or 21) because the Shore TPC-H kit does not implement them.

There are also two important differences between the DB2 and Shore experiments that 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. DB2, on the other hand prefetched 768 disk blocks in a single I/O, while *Lachesis*-modified traces used at most 672 blocks. Thus, the base case for DB2 issues more efficient I/Os relative to its Shore counterpart, and hence the relative improvement for Lachesis-Shore is bigger.

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

Finally, we compared the run times for TPC-H query 12 in Figure 8 against Figure 1. We chose this query because its plan, and hence the resulting access patterns, are relatively straightforward. This query first scans through the LINEITEM table, applying all (sargable) predicates, and then performs a join against the ORDERS table data. Although the x-axes use different units, and hence the particular 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 improvement grows and yields a 1.5× speedup in Shore and 2× speedup in DB2 under the *Heavy traffic* scenario.

5.5 Disk Cache Experiments with DB2

As an interesting aside, we observed different DB2 behavior when executing the same query of the TPC-H benchmark with two different disk drive settings. In the first case, we used the disk's default setting with prefetching and caching turned on, in the second case, we turned them off.

With caching and prefetching turned on, the disk invokes its prefetching algorithm when it observes accesses to consecutive LBNs. This algorithm internally issues a prefetch command for the next set of LBNs, typically the next track, and buffers the data in its on-board cache [25]. Thus, most of the requests are cache hits and the storage manager observed fast response times. Because of these fast response times, DB2 continued to issue many small requests. Even though the storage device executed this access pattern most efficiently (i.e., it correctly detected sequential access and invoked prefetch requests), the storage manager created an unnecessary load on the CPU, memory system, and SCSI bus caused by the overhead of many small I/Os. For query 12, it issued total of 61831 I/Os and finished in 240.2 s.

With disk cache disabled, all read requests must be served from the media and thus experience much longer responses time than cache hits. The DB2 storage manager initially issued small I/Os and gradually ramped up the size to `PREFETCHSIZE`. However, throughout a single query execution, the storage manager would on occasion set the I/O size to one page, and ramp it up again, causing an unnecessary performance degradation due to the smaller, less inefficient I/Os used during ramp-up. For query 12, DB2 used only 10714 I/Os and the query finished in 345.2 s.

6 Summary

This report describes a design and a sample 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 increase.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] Tobias Mayr and Jim Gray. Performance of the One-to-One Data Pump. *http://research.microsoft.com/Gray/river/PerformanceReport.pdf*, 2000.

[21] Manish Mehta and David J. DeWitt. Managing intra-operator parallelism in parallel database systems. *International Conference on Very Large Databases* (Zurich, Switzerland, 11–15 September 1995), pages 382–394. Morgan Kaufmann Publishers, Inc., 1995.

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] Nisha Talagala, Remzi H. Dusseau, and David Patterson. *Microbenchmark-based extraction of local and global disk characteristics*. Technical report CSD–99–1063. University of California at Berkeley, 13 June 2000.

[34] 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.