*Appears in Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04).*
*San Francisco, CA. March 2004.*

# Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks

Jiri Schindler[*], Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, Gregory R. Ganger
*Carnegie Mellon University*

## Abstract

The *Atropos* logical volume manager allows applications to exploit characteristics of its underlying collection of disks. It stripes data in track-sized units and explicitly exposes the boundaries, allowing applications to maximize efficiency for sequential access patterns even when they share the array. Further, it supports efficient diagonal access to blocks on adjacent tracks, allowing applications to orchestrate the layout and access to two-dimensional data structures, such as relational database tables, to maximize performance for both row-based and column-based accesses.

## 1 Introduction

Many storage-intensive applications, most notably database systems and scientific computations, have some control over their access patterns. Wanting the best performance possible, they choose the data layout and access patterns they believe will maximize I/O efficiency. Currently, however, their decisions are based on manual tuning knobs and crude rules of thumb. Application writers know that large I/Os and sequential patterns are best, but are otherwise disconnected from the underlying reality. The result is often unnecessary complexity and inefficiency on both sides of the interface.

Today's storage interfaces (e.g., SCSI and ATA) hide almost everything about underlying components, forcing applications that want top performance to guess and assume [7, 8]. Of course, arguing to expose more information highlights a tension between the amount of information exposed and the added complexity in the interface and implementations. The current storage interface, however, has remained relatively unchanged for 15 years, despite the shift from (relatively) simple disk drives to large disk array systems with logical volume managers (LVMs). The same information gap exists inside disk array systems—although their LVMs sit below a host's storage interface, most do not exploit device-specific features of their component disks.

This paper describes a logical volume manager, called *Atropos* (see Figure 1), that exploits information about its component disks and exposes high-level information about its data organization. With a new data organization and minor extensions to today's storage interface,

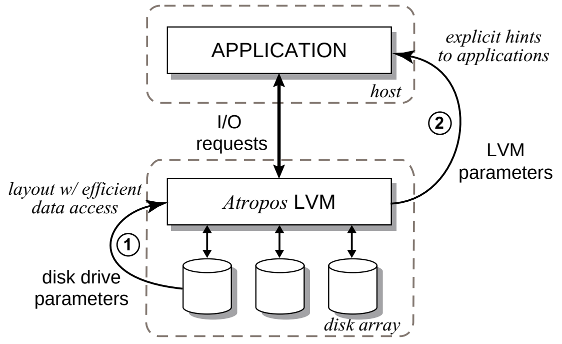


Figure 1: ***Atropos* logical volume manager architecture.** *Atropos* exploits disk characteristics (arrow 1), automatically extracted from disk drives, to construct a new data organization. It exposes high-level parameters that allow applications to directly take advantage of this data organization for efficient access to one- or two-dimensional data structures (arrow 2).

it accomplishes two significant ends. First, *Atropos* exploits automatically-extracted knowledge of disk track boundaries, using them as its stripe unit boundaries. By also exposing these boundaries explicitly, it allows applications to use previously proposed "track-aligned extents" (*traxtents*), which provide substantial benefits for mid-sized segments of blocks and for streaming patterns interleaved with other I/O activity [22].

Second, *Atropos* uses and exposes a data organization that lets applications go beyond the "only one dimension can be efficient" assumption associated with today's linear storage address space. In particular, two-dimensional data structures (e.g., database tables) can be laid out for almost maximally efficient access in both row- and column-orders, eliminating a trade-off [15] currently faced by database storage managers. *Atropos* enables this by exploiting automatically-extracted knowledge of track/head switch delays to support *semi-sequential* access: diagonal access to ranges of blocks (one range per track) across a sequence of tracks.

In this manner, a relational database table can be laid out such that scanning a single column occurs at streaming bandwidth (for the full array of disks), and reading a single row costs only 16%–38% more than if it had been the optimized order. We have implemented *Atropos* as a host-based LVM, and we evaluate it with both database workload experiments (TPC-H) and analytic models. Because *Atropos* exposes its key parameters explicitly, these performance benefits can be realized with no manual tuning of storage-related application knobs.

[*]Now with EMC Corporation.

The rest of the paper is organized as follows. Section 2 discusses current logical volume managers and the need for *Atropos*. Section 3 describes the design an implementation of *Atropos*. Section 4 describes how *Atropos* is used by a database storage manager. Section 5 evaluates *Atropos* and its value for database storage management. Section 6 discusses related work.

## 2 Background

This section overviews the design of current disk array LVMs, which do not exploit the performance benefits of disk-specific characteristics. It highlights the features of the *Atropos* LVM, which addresses shortcomings of current LVMs, and describes how *Atropos* supports efficient access in both column- and row-major orders to applications accessing two-dimensional data structures.

### 2.1 Conventional LVM design

Current disk array LVMs do not sufficiently exploit or expose the unique performance characteristics of their individual disk drives. Since an LVM sits below the host's storage interface, it could internally exploit disk-specific features without the host being aware beyond possibly improved performance. Instead, most use data distribution schemes designed and configured independently of the underlying devices. Many stripe data across their disks, assigning fixed-sized sets of blocks to their disks in a round-robin fashion; others use more dynamic assignment schemes for their fixed-size units. With a well-chosen unit size, disk striping can provide effective load balancing of small I/Os and parallel transfers for large I/Os [13, 16, 18].

A typical choice for the stripe unit size is 32–64 KB. For example, EMC's Symmetrix 8000 spreads and replicates 32 KB chunks across disks [10]. HP's AutoRAID [25] spreads 64 KB "relocation blocks" across disks. These values conform to the conclusions of early studies [4] of stripe unit size trade-offs, which showed that a unit size roughly matching a single disk track (32–64 KB at the times of these systems' first implementations) was a good rule-of-thumb. Interestingly, many such systems seem not to track the growing track size over time (200–350 KB for 2002 disks), perhaps because the values are hard-coded into the design. As a consequence, medium- to large-sized requests to the array result in suboptimal performance due to small inefficient disk accesses.

### 2.2 Exploiting disk characteristics

**Track-sized stripe units**: *Atropos* matches the stripe unit size to the exact track size of the disks in the volume. In addition to conforming to the rule-of-thumb as disk technology progresses, this choice allows applications (and the array itself [11]) to utilize track-based accesses: accesses aligned and sized for one track. Recent research [22] has shown that doing so increases disk efficiency by up to 50% for streaming applications that share the disk system with other activity and for components (e.g., log-structured file systems [17]) that utilize medium-sized segments. In fact, track-based access provides almost the same disk efficiency for such applications as would sequential streaming.

The improvement results from two disk-level details. First, firmware support for zero-latency access eliminates rotational latency for full-track accesses; the data of one track can be read in one revolution regardless of the initial rotational offset after the seek. Second, no head switch is involved in reading a single track. Combined, these positioning delays represent over a third of the total service time for non-aligned, approximately track-sized accesses. Using small stripe unit sizes, as do the array controllers mentioned above, increases the proportion of time spent on these overheads.

*Atropos* uses automated extraction methods described in previous work [21, 22] to match stripe units to disk track boundaries. As detailed in Section 3.3, *Atropos* also deals with multi-zoned disk geometries, whereby tracks at different radial distances have different numbers of sectors. that are not multiples of any useful block size.

**Efficient access to non-contiguous blocks**: In addition to exploiting disk-specific information to determine its stripe unit size, *Atropos* exploits disk-specific information to support efficient access to data across several stripe units mapped to the same disk. This access pattern, called *semi-sequential*, reads some data from each of several tracks such that, after the initial seek, no positioning delays other than track switches are incurred. Such access is appropriate for two-dimensional data structures, allowing efficient access in both row- and column-major order.

In order to arrange data for efficient semi-sequential access, *Atropos* must know the track switch time as well as the track sizes. Carefully deciding how much data to access on each track, before moving to the next, allows *Atropos* to access data from several tracks in one full revolution by taking advantage of the Shortest-Positioning-Time-First (SPTF) [12, 23] request scheduler built into disk firmware. Given the set of accesses to the different tracks, the scheduler can arrange them to ensure efficient execution by minimizing the total positioning time. If the sum of the data transfer times and the track switch times equals the time for one rotation, the scheduler will service them in an order that largely eliminates rotational latency (similar to the zero-latency feature for single track access). The result is that semi-sequential accesses are much more efficient than a like number of random or unorchestrated accesses.

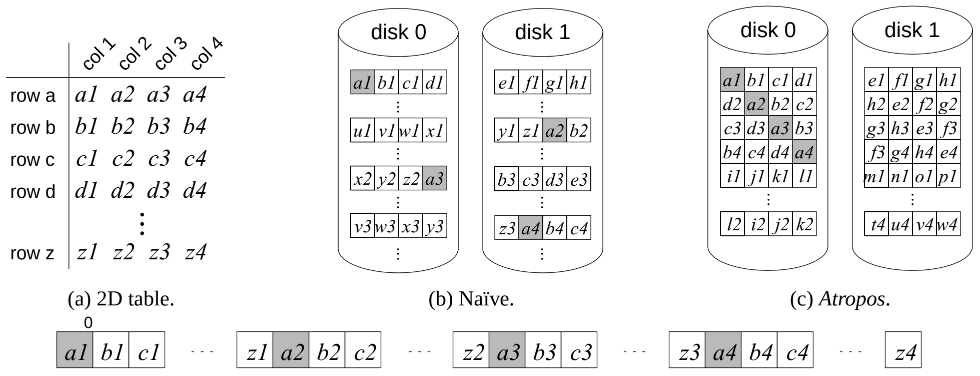


Figure 2: **Two layouts for parallel access to a two-dimensional data structure mapped to a linear *LBN* address space.**

## 2.3 Access to 2D data structures

Figure 2 uses a simple example to illustrate *Atropos*'s benefits to applications that require efficient access to two-dimensional structures in both dimensions and contrasts it with conventional striping in disk arrays. The example depicts a two-dimensional data structure (e.g., a table of a relational database) consisting of four columns $[1,\ldots,4]$ and many rows $[a,\ldots,z]$. For simplicity, each element (e.g., $a_1$) maps to a single *LBN* of the logical volume spanning two disks.

To map this two-dimensional structure into a linear space of *LBN*s, conventional systems decide a priori which order (i.e., column- or row-major) is likely to be accessed most frequently [5]. In the example in Figure 2, a column-major access was chosen and hence the runs of $[a_1,b_1,\ldots,z_1]$, $[a_2,b_2,\ldots,z_2]$, $[a_3,b_3,\ldots,z_3]$, and $[a_4,b_4,\ldots,z_4]$ assigned to contiguous *LBN*s. The mapping of each element to the *LBN*s of the individual disks is depicted in Figure 2(b) in a layout called *Naïve*. When accessing a column, the disk array uses (i) sequential access within each disk and (ii) parallel access to both disks, resulting in maximum efficiency.

Accessing data in the other order (i.e., row-major), however, results in disk I/Os to disjoint *LBN*s. For the example in Figure 2(b), an access to row $[a_1,a_2,a_3,a_4]$ requires four I/Os, each of which includes the high positioning cost for a small random request. The inefficiency of this access pattern stems from the lack of information in conventional systems; one column is blindly allocated after another within the *LBN* address space.

*Atropos* supports efficient access in both orders with a new data organization, depicted in Figure 2(c). This layout maps columns such that their respective first row elements start on the same disk and enable efficient row-order access. This layout still achieves sequential, and hence efficient, column-major access, just like the *Naïve* layout. Accessing the row $[a_1,a_2,a_3,a_4]$, however, is much more efficient than with *Naïve*. Instead of small random accesses, the row is now accessed semi-sequentially in (at most) one disk revolution, incurring much smaller positioning cost (i.e., eliminating all but the first seek and *all* rotational latency). Section 3 describes why this semi-sequential access is efficient.

## 2.4 Efficient access for database systems

By mapping two-dimensional structures (e.g., large non-sparse matrices or database tables) into a linear *LBN* space without providing additional information to applications, efficient accesses in conventional storage systems are only possible in one of row- or column-major order. Database management systems (DBMS) thus predict the common order of access by a workload and choose a layout optimized for that order, knowing that accesses along the other major axis will be inefficient.

In particular, online transaction processing (OLTP) workloads, which make updates to full records, favor efficient row-order access. On the other hand, decision support system (DSS) workloads often scan a subset of table columns and get better performance using an organization with efficient column-order access [15]. Without explicit support from the storage device, however, a DBMS system cannot efficiently support both workloads with one data organization.

The different storage models (a.k.a. page layouts) employed by DBMSs trade the performance of row-major and column-major order accesses. The page layout prevalent in commercial DBMS, called the N-ary storage model (NSM), stores a fixed number of full records (all *n* attributes) in a single page (typically 8 KB). This page layout is optimized for OLTP workloads with row-major access and random I/Os. This layout is also efficient for scans of entire tables; the DBMS can sequentially scan one page after another. However, when only a subset of attributes is desired (e.g., the column-major access prevalent in DSS workloads), the DBMS must fetch full pages with *all* attributes, effectively reading the entire table even though only a fraction of the data is needed.

To alleviate the inefficiency of column-major access with NSM, a decomposition storage model (DSM) vertically partitions a table into individual columns [5]. Each DSM page thus contains a *single* attribute for a fixed number of records. However, fetching full records requires $n$ accesses to single-attribute pages and $n-1$ joins on the record ID to reconstruct the entire record.

The stark difference between row-major and column-major efficiencies for the two layouts described above is so detrimental to database performance that some have even proposed maintaining two copies of each table to avoid it [15]. This solution requires twice the capacity and must propagate updates to each copy to maintain consistency. With *Atropos*'s data layout, which offers efficient access in both dimensions, database systems do not have to compromise.

## 2.5 A more explicit storage interface

Virtually all of today's disk arrays use an interface (e.g., SCSI or ATA) that presents the storage device as a linear space of equally-sized blocks. Each block is uniquely addressed by an integer, called a logical block number (*LBN*). This linear abstraction hides non-linearities in storage device access times. Therefore, applications and storage devices use an unwritten contract, which states that large sequential accesses to contiguous *LBN*s are much more efficient than random accesses and small I/O sizes. Both entities work hard to abide by this implicit contract; applications construct access patterns that favor large I/O and LVMs map contiguous *LBN*s to media locations that ensure efficient execution of sequential I/Os. Unfortunately, an application decides on I/O sizes without any more specific information about the *LBN* mappings chosen by an LVM because current storage interfaces hide it.

In the absence of clearly defined mechanisms, applications rely on knobs that must be manually set by a system administrator. For example, the IBM DB2 relational database system uses the `PREFETCHSIZE` and `EXTENTSIZE` parameters to determine the maximal size of a prefetch I/O for sequential access and the number of pages to put into a single extent of contiguous *LBN*s [6]. Another parameter, called `DB2_STRIPED_CONTAINERS`, instructs DBMS to align I/Os on stripe unit boundaries. Relying on proper knob settings is fragile and prone to human errors: it may be unclear how to relate them to LVM configuration parameters. Because of these difficulties, and the information gap introduced by inexpressive storage interfaces, applications cannot easily take advantage of significant performance characteristics of modern disk arrays. *Atropos* exposes explicit information about stripe unit sizes and semi-sequential access. This information allows applications to directly match their access patterns to the disk array's characteristics.

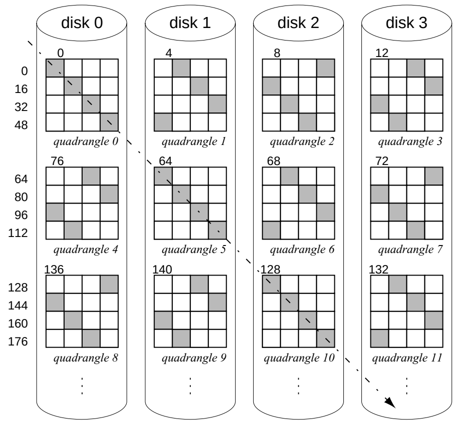


Figure 3: ***Atropos* quadrangle layout.** The numbers to the left of disk 0 are the *VLBN*s mapped to the gray disk locations connected by the arrow (not the first block of each quadrangle row). The arrow illustrates efficient access in the other-major.

## 3 Atropos logical volume manager

The *Atropos* disk array LVM addresses the aforementioned shortcomings of many current disk array LVM designs. It exploits disk-specific characteristics to construct a new data organization. It also exposes high-level features of this organization to higher-levels of the storage stack, allowing them to directly take advantage of key device-specific characteristics. This section details the new data organization and the information *Atropos* exposes to applications.

## 3.1 Atropos data organization

As illustrated in Figure 3, *Atropos* lays data across $p$ disks in basic allocation units called quadrangles. A quadrangle is a collection of logical volume *LBN*s, here referred to as *VLBN*s, mapped to a single disk. Each successive quadrangle is mapped to a different disk.

A quadrangle consists of $d$ consecutive disk tracks, with $d$ referred to as the quadrangle's *depth*. Hence, a single quadrangle is mapped to a contiguous range of a single disk's logical blocks, here referred to as *DLBN*s. The *VLBN* and *DLBN* sizes may differ; a single *VLBN* consists of $b$ *DLBN*s, with $b$ being the block size of a single logical volume block. For example, an application may choose a *VLBN* size to match its allocation units (e.g., an 8 KB database block size), while a *DLBN* is typically 512 bytes.

Each quadrangle's dimensions are $w \times d$ logical blocks (*VLBN*s), where $w$ is the quadrangle width and equals the number of *VLBN*s mapped to a single track. In Figure 3, both $d$ and $w$ are four. The relationship between the dimensions of a quadrangle and the mappings to individual logical blocks of a single disk are described in Section 3.2.2.

The goal of the *Atropos* data organization is to allow efficient access in two dimensions. Efficient access of the primary dimension is achieved by striping contiguous *VLBN*s across quadrangles on all disks. Much like ordinary disk arrays, which map *LBN*s across individual stripe units, each quadrangle row contains a contiguous run of *VLBN*s covering a contiguous run of a single disk's *DLBN*s on a single track. Hence, sequential access naturally exploits the high efficiency of track-based access explained in Section 2.2. For example, in Figure 3, an access to 16 sequential blocks starting at *VLBN* 0, will be broken into four disk I/Os executing in parallel and fetching full tracks: *VLBN*s 0–3 from disk 0, *VLBN*s 4–7 from disk 1, *VLBN*s 8–11 from disk 2, and *VLBN*s 12–15 from disk 3.

Efficient access to the secondary dimension is achieved by mapping it to semi-sequential *VLBN*s. Figure 3 indicates the semi-sequential *VLBN*s with a dashed line. Requests to the semi-sequential *VLBN*s in a single quadrangle are all issued together in a batch. The disk's internal scheduler then chooses the request that will incur the smallest positioning cost (the sum of seek and rotational latency) and services it first. Once the first request is serviced, servicing all other requests will incur only a track switch to the adjacent track. Thanks to the semi-sequential layout, no rotational latency is incurred for any of the subsequent requests, regardless of which request was serviced first.

Naturally, the sustained bandwidth of semi-sequential access is smaller than that of sequential access. However, semi-sequential access is more efficient than reading $d$ effectively-random *VLBN*s spread across $d$ tracks, as would be the case in a normal striped disk array. Accessing random *VLBN*s will incur rotational latency, averaging half a revolution per access. In the example of Figure 3, the semi-sequential access, depicted by the arrow, proceeds across *VLBN*s $0, 16, 32, \ldots, 240$ and occurs on all $p$ disks, achieving the aggregate semi-sequential bandwidth of the disk array.

## 3.2 Quadrangle layout parameters

The values that determine efficient quadrangle layout depend on disk characteristics, which can be described by two parameters. The parameter $N$ describes the number of sectors, or *DLBN*s, per track. The parameter $H$ describes the track skew in the mapping of *DLBN*s to physical sectors. The layout and disk parameters are summarized in Table 1.

Track skew is a property of disk data layouts as a consequence of track switch time. When data is accessed sequentially on a disk beyond the end of a track, the disk must switch to the next track to continue accessing. Switching tracks takes some amount of time, during which no data can be accessed. While the track switch is in progress, the disk continues to spin, of course. Therefore, sequential *LBN*s on successive tracks are physically skewed so that when the switch is complete, the head will be positioned over the next sequential *LBN*. This skew is expressed as the parameter $H$ which is the number of *DLBN*s that the head passes over during the track switch time.

| **Symbol** | **Name** | **Units** |
|---|---|---|
| *Quadrangle layout parameters* | | |
| $p$ | Parallelism | # of disks |
| $d$ | Quadrange depth | # of tracks |
| $b$ | Block size | # of *DLBN*s |
| $w$ | Quadrange width | # of *VLBN*s |
| *Disk physical parameters* | | |
| $N$ | Sectors per track | |
| $H$ | Head switch | in *DLBN*s |

Table 1: **Parameters used by Atropos.**

Figure 4 shows a sample quadrangle layout and its parameters. Figure 4(a) shows an example of how quadrangle *VLBN*s map to *DLBN*s. Along the $x$-axis, a quadrangle contains $w$ *VLBN*s, each of size $b$ *DLBN*s. In the example, one *VLBN* consists of two *DLBN*s, and hence $b = 2$. As illustrated in the example, a quadrangle does not always use all *DLBN*s when the number of sectors per track, $N$, is not divisible by $b$. In this case, there are $R$ residual *DLBN*s that are not assigned to quadrangles. Figure 4(b) shows the physical locations of each $b$-sized *VLBN* on individual tracks, accounting for track skew, which equals 3 sectors ($H$= 3 *DLBN*s) in this example.

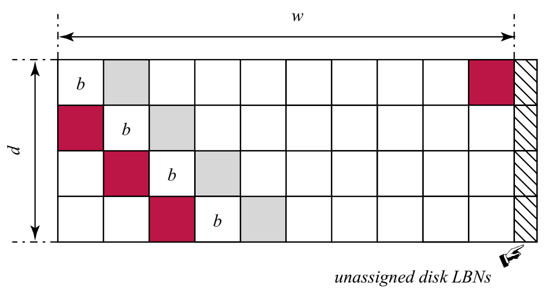
### 3.2.1 Determining layout parameters

To determine a suitable quadrangle layout at format time, *Atropos* uses as its input parameters the automatically extracted disk characteristics, $N$ and $H$, and the block size, $b$, which are given by higher level software. Based on these input parameters, the other quadrangle layout parameters, $d$ and $w$, are calculated as described below.

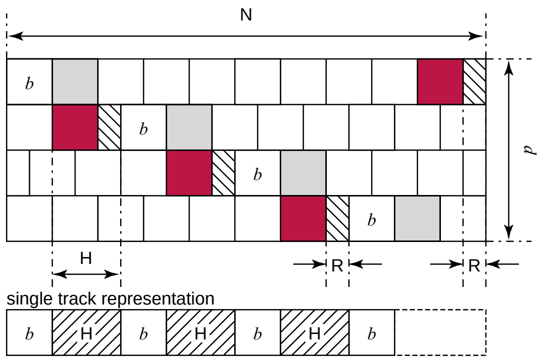
To explain the relationship between the quadrangle layout parameters and the disk physical parameters, let's assume that we want to read one block of $b$ *DLBN*s from each of $d$ tracks. This makes the total request size, $S$, equal to $db$. As illustrated in Figure 4(b), the locations of the $b$ blocks on each track are chosen to ensure the most efficient access. Accessing $b$ on the next track can commence as soon as the disk head finishes reading on the previous track and repositions itself above the new track. During the repositioning, $H$ sectors pass under the heads.

To bound the response time for reading the $S$ sectors, we need to find suitable values for $b$ and $d$ to ensure that the entire request, consisting of $db$ sectors, is read in at most one revolution. Hence,

$$\frac{db}{N} + \frac{(d-1)H}{N} \leq 1 \qquad (1)$$



(a) **Logical layout in disk *LBN* space.**



(b) **Physical layout on disk sectors.**

Figure 4: **Single quadrangle layout.** In this example, the quadrangle layout parameters are *b*=2 (a single *VLBN* consists of two *DLBN*s), *w*=10 *VLBN*s, and *d*= 4 tracks. The disk physical parameters are *H*=3 *DLBN*s and *N*=21 *DLBN*s. Given these parameters, *R*=1.

where $db/N$ is the media access time needed to fetch the desired $S$ sectors and $(d-1)H/N$ is the fraction of time spent in head switches when accessing all $d$ tracks. Then, as illustrated at the bottom of Figure 4(b), reading $db$ sectors is going to take the same amount of time as if we were reading $db+(d-1)H$ sectors on a single track of a zero-latency access disk.

The maximal number of tracks, $d$, from which at least one sector each can be read in a single revolution is bound by the number of head switches that can be done in a single revolution, so

$$d \leq \left\lfloor \frac{N}{H} \right\rfloor - 1 \tag{2}$$

If we fix $d$, the number of sectors, $b$, that yield the most efficient access (i.e., reading as many sectors on a single track as possible before switching to the next one) can be determined from Equation 1 to get

$$b \leq \frac{N+H}{d} - H \tag{3}$$

Alternatively, if we fix $b$, the maximal depth, called $D_{max}$, can be expressed from Equation 1 as

$$D_{max} \leq \frac{N+H}{b+H} \tag{4}$$

For certain values of $N$, $db$ sectors do not span a full track. In that case, $db+(d-1)H < N$ and there are $R$ residual sectors, where $R < b$, as illustrated in Figure 4. The number of residual *DLBN*s on each track not mapped to quadrangle blocks is $R = N \bmod w$, where

$$w = \left\lfloor \frac{N}{b} \right\rfloor \tag{5}$$

Hence, the fraction of disk space that is wasted with *Atropos*' quadrangle layout is $R/N$; these sectors are skipped to maintain the invariant that $db$ sectors can be accessed in at most one revolution. Section 5.2.4 shows that this number is less than 2% of the total disk capacity.

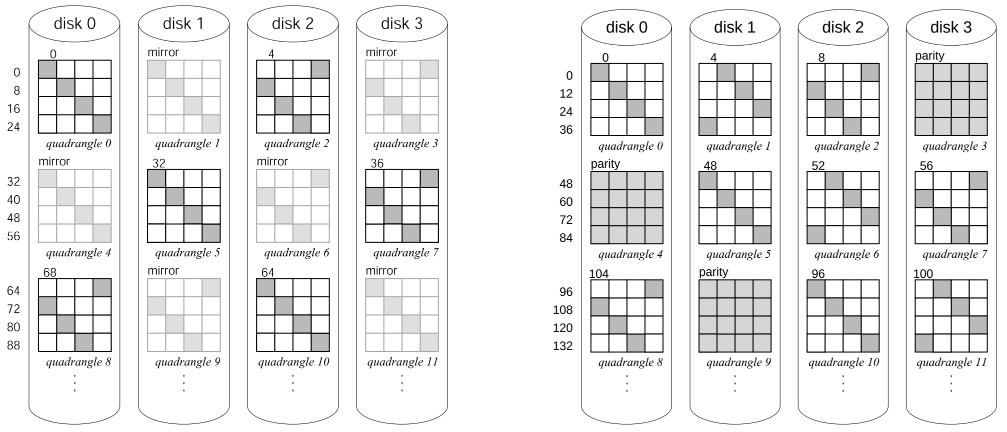
While it may seem that relaxing the one revolution constraint might achieve better efficiency, Appendix B shows that this intuition is wrong. Accessing more than $D_{max}$ tracks is detrimental to the overall performance unless $d$ is some multiple of $D_{max}$. In that case, the service time for such access is a multiple of one-revolution time.

### 3.2.2 Mapping *VLBN*s to quadrangles

Mapping *VLBN*s to the *DLBN*s of a single quadrangle is straightforward. Each quadrangle is identified by $DLBN_Q$, which is the lowest *DLBN* of the quadrangle and is located at the quadrangle's top-left corner. The *DLBN*s that can be accessed semi-sequentially are easily calculated from the $N$ and $b$ parameters. As illustrated in Figure 4, given $DLBN_Q = 0$ and $b = 2$, the set $\{0, 24, 48, 72\}$ contains blocks that can be accessed semi-sequentially. To maintain rectangular appearance of the layout to an application, these *DLBN*s are mapped to *VLBN*s $\{0, 10, 20, 30\}$ when $b = 2$, $p = 1$, and $VLBN_Q = DLBN_Q = 0$.

With no media defects, *Atropos* only needs to know the $DLBN_Q$ of the first quadrangle. The $DLBN_Q$ for all other quadrangles can be calculated from the $N$, $d$, and $b$ parameters. With media defects handled via slipping (e.g., the primary defects that occurred during manufacturing), certain tracks may contain fewer *DLBN*s. If the number of such defects is less than $R$, that track can be used; if it is not, the *DLBN*s on that track must be skipped. If any tracks are skipped, the starting *DLBN* of each quadrangle row must be stored.

To avoid the overhead of keeping a table to remember the *DLBN*s for each quadrangle row, *Atropos* could reformat the disk and instruct it to skip over any tracks that contain one or more bad sectors. By examining twelve Seagate Cheetah 36ES disks, we found there were, on average, 404 defects per disk; eliminating all tracks with defects wastes less than 5% of the disk's total capacity. The techniques for handling grown defects still apply.



(a) RAID 1 layout. (b) RAID 5 layout.

Figure 5: ***Atropos*** **quadrangle layout for different RAID levels.**

## 3.3 Practical system integration

Building an *Atropos* logical volume out of $p$ disks is not difficult thanks to the regular geometry of each quadrangle. *Atropos* collects a set of disks with the same basic characteristics (e.g., the same make and model) and selects a disk zone with the desired number of sectors per track, $N$. The *VLBN* size, $b$, is set according to application needs, specifying the access granularity. For example, it may correspond to a file system block size or database page size. With $b$ known, *Atropos* uses disk parameters to determine the resulting $d \leq D_{max}$.

In practice, volume configuration can be accomplished in a two-step process. First, higher-level software issues a FORMAT command with desired values of volume capacity, level of parallelism $p$, and block size $b$. Internally, *Atropos* selects appropriate disks (out of a pool of disks it manages), and formats the logical volume by implementing a suitable quadrangle layout.

### 3.3.1 Zoned disk geometries

With zoned-disk geometries, the number of sectors per track, $N$, changes across different zones, which affects both the quadrangle width, $w$, and depth, $d$. The latter changes because the ratio of $N$ to $H$ may be different for different zones; the track switch time does not change, but the number of sectors that rotate by in that time does. By using disks with the same geometries (e.g., same disk models), we opt for the simple approach: quadrangles with one $w$ can be grouped into one logical volume and those with another $w$ (e.g., quadrangles in a different zone) into a different logical volume. Since modern disks have fewer than 8 zones, the size of a logical volume stored across a few 72 GB disks would be tens of GBs.

### 3.3.2 Data protection

Data protection is an integral part of disk arrays and the quadrangle layout lends itself to the protection models of traditional RAID levels. Analogous to the parity unit, a set of quadrangles with data can be protected with a parity quadrangle. To create a RAID5 homologue of a parity group with quadrangles, there is one parity quadrangle unit for every $p-1$ quadrangle stripe units, which rotates through all disks. Similarly, the RAID 1 homologue can be also constructed, where each quadrangle has a mirror on a different disk. Both protection schemes are depicted in Figure 5.

### 3.3.3 Explicit information to applications

To allow applications to construct efficient streaming access patterns, *Atropos* needs to expose the parameter $w$, denoting the stripe unit size. I/Os aligned and sized to stripe unit boundaries can be executed most efficiently thanks to track-based access and rotating stripe units through all $p$ disks. Applications with one-dimensional access (e.g., streaming media servers) then exercise access patterns consisting of $w$-sized I/Os that are aligned on disk track boundaries.

For applications that access two-dimensional data structures, and hence want to utilize semi-sequential access, *Atropos* also needs to expose the number of disks, $p$. Such applications then choose the primary order for data and allocate $w \times p$ blocks of this data, corresponding to a portion of column 1 $\{a_1, \ldots, h_1\}$ in Figure 2. They allocate to the next $w \times p$ *VLBN*s the corresponding data of the other-major order (e.g., the $\{a_2, \ldots, h_2\}$ portion of column 2) and so on, until all are mapped. Thus, the rectangular region $\{a_1, \ldots, h_4\}$ would be mapped to $4wp$ contiguous *VLBN*s.

Access in the primary-major order (columns in Figure 2) consists of sequentially reading $wp$ *VLBN*s. Access in the other-major order is straightforward; the application simply accesses every *wp*-th *VLBN* to get the data of the desired row. *Atropos* need not expose to applications the parameter *d*. It is computed and used internally by *Atropos*.

Because of the simplicity of information *Atropos* exposes to applications, the interface to *Atropos* can be readily implemented with small extensions to the commands already defined in the SCSI protocol. The parameters *p* and *w* could be exposed in a new mode page returned by the MODE SENSE SCSI command. To ensure that *Atropos* executes all requests to non-contiguous *VLBN*s for the other-major access together, an application can link the appropriate requests. To do so, the READ or WRITE commands for semi-sequential access are issued with the Link bit set.

### 3.3.4 Implementation details

Our *Atropos* logical volume manager implementation is a stand-alone process that accepts I/O requests via a socket. It issues individual disk I/Os directly to the attached SCSI disks using the Linux raw SCSI device `/dev/sg`. With an SMP host, the process can run on a separate CPU of the same host, to minimize the effect on the execution of the main application.

An application using *Atropos* is linked with a stub library providing API functions for reading and writing. The library uses shared memory to avoid data copies and communicates through the socket with the *Atropos* LVM process. The *Atropos* LVM organization is specified by a configuration file, which functions in lieu of a format command. The file lists the number of disks, *p*, the desired block size, *b*, and the list of disks to be used.

For convenience, the interface stub also includes three functions. The function *get_boundaries(LBN)* returns the stripe unit boundaries between which the given *LBN* falls. Hence, these boundaries form a collection of *w* contiguous *LBN*s for constructing efficient I/Os. The *get_rectangle(LBN)* function returns the *wp* contiguous *LBN*s in a single row across all disks. These functions are just convenient wrappers that calculate the proper *LBN*s from the *w* and *p* parameters. Finally, the stub interface also includes a *batch()* function to explicitly group READ and WRITE commands (e.g., for semi-sequential access).

With no outstanding requests in the queue (i.e., the disk is idle), current SCSI disks will immediately schedule the first received request of batch, even though it may not be the one with the smallest rotational latency. This diminishes the effectiveness of semi-sequential access. To overcome this problem, our *Atropos* implementation "pre-schedules" the batch of requests by sending first the request that will incur the smallest rotational latency. It uses known techniques for SPTF scheduling outside of disk firmware [14]. With the help of a detailed and validated model of the disk mechanics [2, 21], the disk head position is deduced from the location and time of the last-completed request. If disks waited for all requests of a batch before making a scheduling decision, this pre-scheduling would not be necessary.

Our implementation of the *Atropos* logical volume manager is about 2000 lines of C++ code and includes implementations of RAID levels 0 and 1. Another 600 lines of C code implement methods for automatically extracting track boundaries and head switch time [22, 26].

## 4 Efficient access in database systems

Efficient access to database tables in both dimensions can significantly improve performance of a variety of queries doing selective table scans. These queries can request (i) a subset of columns (restricting access along the primary dimension, if the order is column-major), which is prevalent in decision support workloads (TPC-H), (ii) a subset of rows (restricting access along the secondary dimension), which is prevalent in online transaction processing (TPC-C), or (iii) a combination of both.

A companion project [24] to *Atropos* extends the Shore database storage manager [3] to support a page layout that takes advantage of *Atropos*'s efficient accesses in both dimensions. The page layout is based on a cache-efficient page layout, called PAX [1], which extends the NSM page layout to group values of a single attribute into units called "minipages". Minipages in PAX exist to take advantage of CPU cache prefetchers to minimize cache misses during single-attribute memory accesses. We use minipages as well, but they are aligned and sized to fit into one or more 512 byte *LBN*s, depending on the relative sizes of the attributes within a single page.

The mapping of 8 KB pages onto the quadrangles of the *Atropos* logical volume is depicted in Figure 6. A single page contains 16 equally-sized attributes, labeled A1–A16, where each attribute is stored in a separate minipage that maps to a single *VLBN*. Accessing a single page is thus done by issuing 16 batched requests to every $16^{th}$ (or more generally, *wp*-th) *VLBN*. Internally, the *VLBN*s comprising this page are mapped diagonally to the blocks marked with the dashed arrow. Hence, 4 semi-sequential accesses proceeding in parallel can fetch the entire page (i.e., row-major order access).

Individual minipages are mapped across sequential runs of *VLBN*s. For example, to fetch attribute A1 for records 0–399, the database storage manager can issue one efficient sequential I/O to fetch the appropriate minipages. *Atropos* breaks this I/O into four efficient, track-based disk accesses proceeding in parallel. The database storage manager then reassembles these minipages into appropriate 8 KB pages [24].

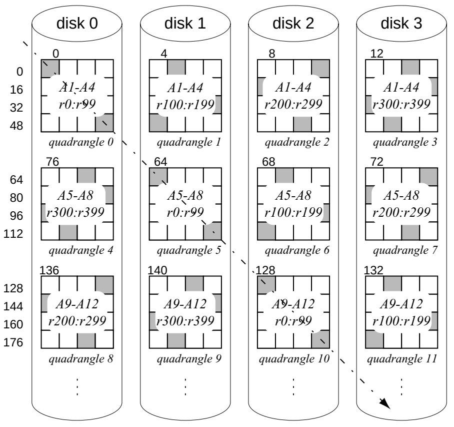


Figure 6: **Mapping of a database table with 16 attributes onto *Atropos* logical volume with 4 disks.**

Fetching any subset of attributes for a given page (record range) is thus a simple matter of issuing the corresponding number of I/Os, each accessing a contiguous region of the *VLBN* space mapped to a contiguous region on the disk. If several I/Os fall onto stripe units mapped to the same disk, the internal disk scheduler optimizes their execution by minimizing positioning times.

## 5 Evaluation

This section evaluates the performance of *Atropos*. First, it quantifies the efficiencies of sequential, semi-sequential and random accesses and shows the impact of disk trends on the layout parameter values. For all access patterns, *Atropos* achieves performance comparable or superior to conventional disk array data organizations. Second, trace replay experiments of a TPC-H workload on the *Atropos* implementation shows the benefit of matching the stripe-unit size to the exact track size and exposing it to applications. Third, the benefits of *Atropos*'s data organizations are shown for (a subset of) queries from the TPC-H benchmark running on the Shore database storage manager [3] and our *Atropos* implementation.

### 5.1 Experimental setup

The experiments were performed on a four-way 500 MHz Pentium III machine with 1 GB of memory running Linux kernel v. 2.4.7 of RedHat 7.1 distribution. The machine was equipped with two Adaptec Ultra160 Wide SCSI adapters on two separate PCI buses, each controlling two 36 GB Maxtor Atlas 10K III disks.

### 5.2 Quantifying access efficiency

Traditional striped layouts of data across disks in a RAID group offer efficient (sequential) access along one major. The efficiency of accessing data along the other major is much lower, essentially involving several random accesses. *Atropos*'s quadrangle layout, on the other hand, offers streaming efficiency for sequential accesses and much higher efficiency for the other-major access. We define "access efficiency" as the fraction of total access time spent reading/writing data from/to the media. The access efficiency is reduced by activities other than data transfer, including seeks, rotational latencies, and track switches. The efficiencies and response times described in this subsection are for a single disk. With $p$ disks comprising a logical volume, each disk can experience the same efficiency while accessing data in parallel.

#### 5.2.1 Efficient access in both majors

Figure 7 graphs the access efficiency of the quadrangle layout as a function of I/O size. It shows two important features of the *Atropos* design. First, accessing data in the primary-order (line 1) matches the best-possible efficiency of track-based access with traxtents. Second, the efficiency of the other other-major order access (line 2) is much higher than the same type of access with the *Naïve* layout of conventional disk arrays (line 3), thanks to semi-sequential access.

The data in the graph was obtained by measuring the response times of requests issued to randomly chosen *DLBN*s, aligned on track boundaries, within the Atlas 10K III's outer-most zone (686 sectors or 343 KB per track ). The average seek time in the first zone is 2.46 ms. The drop in the primary-major access efficiency at the 343 KB mark is due to rotational latency and an additional track switch incurred for I/Os larger than the track size, when using a single disk.

The I/O size for the other-major access with the quadrangle layout is the product of quadrangle depth, $d$, and the number of consecutive *DLBN*s, $b$, accessed on each track. For $d = 4$, a request for $S$ sectors is split into four I/Os of $S/4$ *DLBN*s. For this access in the *Naïve* layout (line 3), servicing these requests includes one seek and some rotational latency for each of the four $b$-sized I/Os, which are "randomly" located on each of the four consecutive tracks.

The efficiency of semi-sequential quadrangle access (line 2) with I/O sizes below 124 KB is only slightly smaller than that of the efficiency of track-based access with traxtents. Past this point, which corresponds to the one-revolution constraint, the efficiency increases at a slower rate, eventually surpassing the efficiency value at the 124 KB mark. However, this increase in efficiency comes at a cost of increased request latency; the access will now require multiple revolutions to service. The continuing increase in efficiency past the 124 KB mark is due to amortizing the cost of a seek by larger data transfer. Recall that each request includes an initial seek.

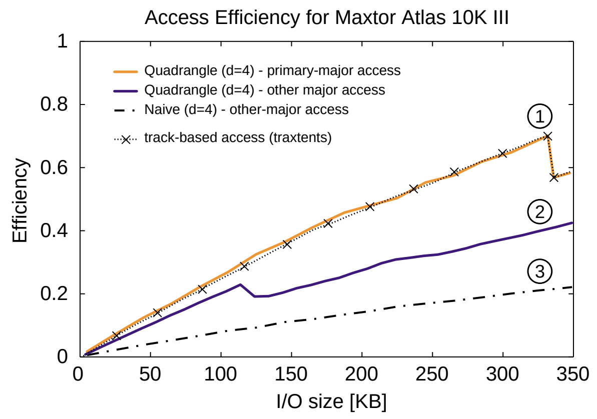


Figure 7: **Comparison of access efficiencies.** The maximal streaming efficiency, i.e., without seeks, for this disk is 0.82 (computed by Equation 6 in Appendix A).

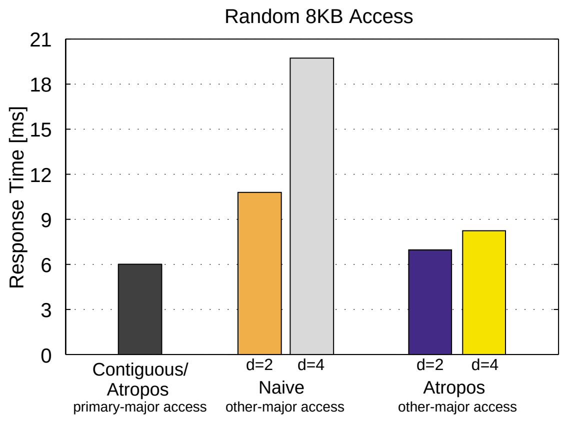


Figure 8: **Comparison of response times for random access.**

### 5.2.2 Random accesses in the other-major

Figure 8 compares access times for a random 8 KB chunk of data with different data layouts. The goal is to understand the cost of accessing data in the other-major order (e.g., row-major order access of the table in Figure 2). For context, a block in the primary-major has its data mapped to consecutive *LBN*s. Such an access incurs an average seek of 2.46 ms and an average rotational latency of half a revolution, followed by an 8 KB media access. The total response time of 5.93 ms is shown by the bar labeled "Contiguous."

Accessing 8 KB of data randomly spread across non-contiguous *VLBN*s (e.g., single row access in the *Naïve* layout of Figure 2) incurs nearly half of a revolution of rotational latency for each of the *d* accesses in addition to the same initial seek. Such an access results in a large response time, as shown by the bars labeled "Naive." Database systems using the DSM data layout decomposed into *d* separate tables suffer this high penalty when complete records are retrieved.

In contrast, with the quadrangle layout, an access in the other-major incurs only a single seek and much less total rotational latency than the access in the traditional *Naïve* layout. This access still incurs one (for $d = 2$) or three (for $d = 4$) track switches, which explains the penalty of 16% and 38% relative to the best case.

### 5.2.3 Access performance analysis

Using parameters derived in Section 3.2 and the analytical model described in Appendix A, we can express the expected response time for a quadrangle access and compare it with measurements taken from a real disk.

Figure 9 plots response times for quadrangle accesses to the disk's outer-most zone as a function of I/O request size, *S*, and compares the values obtained from the analytic model to measurements from a real disk. The close match between these data sets demonstrates that *Atropos* can reliably determine proper values of quadrangle layout analytically rather than empirically, which may be time consuming. The data is shown for the Atlas 10K III disk: $N = 686$, $H = 139$, and 6 ms revolution time.

The plotted response time does not include seek time; adding it to the response time would simply shift the lines up by an amount equivalent to the average seek time. The total I/O request size, *S*, shown along the *x*-axis is determined as $S = db$. With $d = 1$, quadrangle access reduces to normal disk access. Thus, the expected response time grows from 3 to 6 ms. For $d = 6$, the response time is at least 10.8 ms, even for the smallest possible I/O size, because $D_{max} = 5$ for the given disk.

The most prominent features of the graph are the steps from the 6 ms to 10–12 ms regions. This abrupt change in response time shows the importance of the one-revolution constraint. If this constraint is violated by an I/O size that is too large, the penalty in response time is significant.

The data measured on the real disk (dashed lines in Figure 9) match the predicted values. To directly compare the two sets of data, the average seek value was subtracted from the measured values. The small differences occur because the model does not account for bus transfer time, which does not proceed entirely in parallel with media transfer.

### 5.2.4 Effect of disk characteristics

Figure 9 shows the relationship between quadrangle dimensions and disk characteristics of one particular disk with $D_{max} = 5$. To determine how disk characteristics affect the quadrangle layout, we use the analytic model to study other disks. As shown in Table 2, the dimensions of the quadrangles mapped to the disks' outer-most zones remain stable across different disks of the past decade. The smaller $D_{max}$ for the Atlas 10K III is due to an unfortunate track skew/head switch of $H = 139$. If $H = 136$, $D_{max} = 6$ and $b = 1$.

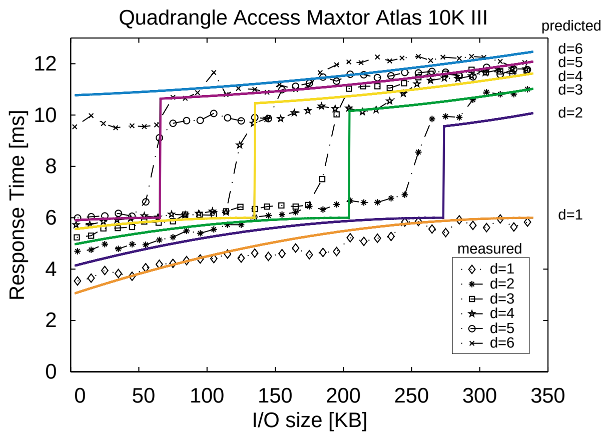


Figure 9: **Response time of semi-sequential access.**

Table 2 also shows that, with *d* set to $D_{max}$, the number of *DLBN*s, *b*, accessed at each disk track remains below 10, with the exception of the Atlas 10K III. The data reveals another favorable trend: the small value of *R* (number of *DLBN*s on each track not mapped to *VLBN*s) is a modest capacity tradeoff for large performance gains. With the exception of the Atlas 10K III disk, less than 1% of the total disk capacity would be wasted. For that disk, the value is 1.5%.

## 5.3 Track-sized stripe units

We now evaluate the benefits of one *Atropos* feature in isolation: achieving efficient sequential access by matching stripe units to exact track boundaries and exposing it to applications. To do so, we replayed block-level I/O traces of the TPC-H benchmark, representing a decision support system workload dominated by large sequential I/O. The original traces were captured on an IBM DB2 v. 7.2 system using 8 KB NSM pages and running each of the 22 TPC-H queries separately. The configuration specifics and the description of the trace replay transformations are detailed elsewhere [20].

For the experiments described in the remainder of this section, we used a single logical volume created from four disks ($p = 4$) and placed it on the disks' outermost zone, giving it a total size of 35 GB. The quadrangle layout was configured as RAID 0 with $d = 4$ and $b = 1$.

To simulate the effects of varying stripe unit size and exposing its value to DB2, we modified the 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 stripe unit size, preserving page boundaries.

To simulate traditional disk arrays with a (relatively small) hard-coded stripe unit size, we set the stripe unit size to 64 KB (128 blocks) and called this base case scenario *64K-RAID*. To simulate systems that approximate track size, but do not take advantage of disk characteristics, we set the stripe unit to 256 KB (512 blocks) and called this scenario *Approximate-RAID*. By taking advantage of automatically-extracted explicit disk characteristics, *Atropos* can set the stripe unit size to the exact track size of 343 KB and we called this scenario *Atropos-RAID*. For all experiments, we used the *Atropos* LVM cofigured for RAID 0 (i.e., $d = 1$ and *w* was 128, 512, and 686 blocks respectively) and I/O sizes matching stripe unit sizes.

The resulting I/O times of the 22 TPC-H queries are shown in Figure 10. The graph shows the speedup of the *Approximate-RAID* and *Atropos-RAID* scenarios relative to the base case scenario *64K-RAID*. The results are in agreement with the expectations of sequential access efficiencies in Figure 7. The larger, more efficient I/Os of the *Approximate-RAID* and *Atropos-RAID* result in the observed speedup. The exact full-track access of *Atropos-RAID* provides additional 2%–23% benefit.

Some fraction of the query I/Os are not sequential or stripe-unit sized (e.g., less than 1% for query 1 vs. 93% and 99% for queries 14 and 17, respectively). These differences explain why some queries are sped up more than others; *Atropos* does not significantly improve the small random I/Os produced by index accesses.

The efficiency of *Atropos-RAID* is automatically maintained with technology changes (e.g., increasing numbers of sectors per track). While *Approximate-RAID* must be manually set to approximate track size (provided it is known in the first place), *Atropos* automatically determines the correct value for its disks and sets its stripe unit size accordingly, eliminating the error-prone manual configuration process.

| **Disk** | **Year** | $D_{max}$ | *b* | *R* |
|---|---|---|---|---|
| HP C2247 | 1992 | 7 | 1 | 0 |
| IBM Ultrastar 18 ES | 1998 | 7 | 5 | 0 |
| Quantum Atlas 10K | 1999 | 6 | 2 | 0 |
| Seagate Cheetah X15 | 2000 | 6 | 4 | 2 |
| Seagate Cheetah 36ES | 2001 | 6 | 7 | 3 |
| Maxtor Atlas 10K III | 2002 | 5 | 26 | 10 |
| Seagate Cheetah 73LP | 2002 | 7 | 8 | 2 |

Table 2: **Quadrangle parameters across disk generations.** For each disk, the amount of space not utilized due to *R* residual *DLBN*s is less than 1% with the exception of the Atlas 10K III, where it is 1.5%.

## 5.4 Two-dimensional data access

To quantify the benefits of both efficient sequential and semi-sequential accesses, we used a TPC-H benchmark run on the Shore database storage manager [3] with three different layouts. The first layout, standard *NSM*, is optimized for row-major access. The second layout, standard *DSM*, vertically partitions data to optimize column-major access. The third layout, here called *AtroposDB*, uses the page layout described in Section 4, which can

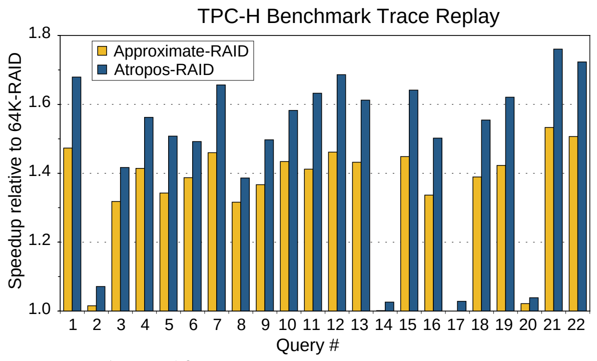


Figure 10: **TPC-H trace replay on *Atropos*.**

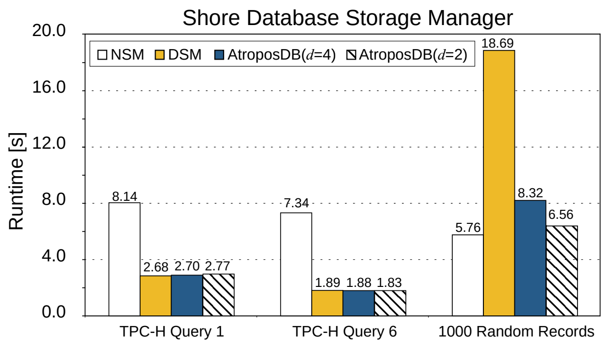


Figure 11: **Database access results.** This figure shows the runtime of TPC-H queries 1 and 6 for three different layouts. The *NSM* and *DSM* layouts are optimized for row- and column-order access respectively, trading off performance in the other-order. *Atropos*, on the other hand, offers efficient execution of TPC-H queries as well as random-page access in OLTP workloads (random access to 1000 records).

take advantage of *Atropos*'s full set of features. Each setup uses an 8 KB page size.

The results for TPC-H queries 1 and 6 are shown in Figure 11.[1] These queries scan through the LINEITEM table (the largest table of the TPC-H benchmark), which consists of 16 attributes, and calculate statistics based on a subset of six (Q1) and four (Q6) attributes. As expected, the performance of *DSM* and *AtroposDB* is comparable, since both storage models can efficiently scan through the table, requesting data only for the desired attributes. *NSM*, on the other hand, fetches pages that contain full records (including the attributes not needed by the queries), which results in the observed 2.5× to 4× worse performance. All scans were performed on a 1 GB TPC-H installation, with the LINEITEM table constituting about 700 MB.

Random record accesses to the LINEITEM table approximate an OLTP workload behavior, which is dominated by row-major access. For *DSM*, which is not suitable for row-major access, this access involves 16 random accesses to the four disks. For *NSM*, this access involves a random seek and rotational latency of half a revolution, followed by an 8 KB page access, resulting in a run time of 5.76 s. For *AtroposDB*, this access includes the same seek, but most rotational latency is eliminated, thanks to semi-sequential access. It is, however, offset by incurring additional track switches. With $d$=4, the run time is 8.32 s; with $d$=2, it is 6.56 s. These results are in accord with the random access results of Figure 8.

## 6 Related work

*Atropos* builds upon many ideas proposed in previous research. Our contribution is in integrating them into a novel disk array organization, cleanly extending the storage interface to allow applications to exploit it, and evaluating the result with a real implementation executing benchmark database queries.

[1]Of the four TPC-H queries implemented by the Shore release available to us, only Q1 and Q6 consist of LINEITEM table scan that is dominated by I/O time.

*Atropos*'s track-based striping is inspired by recent work on track-aligned extents [22], which showed that track-based access could provide significant benefits for systems using a single disk. Most high-end systems use disk arrays and LVMs, and *Atropos* allows those to utilize track-based access.

Gorbatenko and Lilja [9] proposed diagonal disk layout of relational tables, allowing what we call semi-sequential access. *Atropos* integrates such diagonal layout with track-aligned extents to realize its support for two-dimensional data structures with no penalty to the primary access order.

Section 2.4 describes how modern databases address the trade-off between row-major and column-major access to database tables. Recently, Ramamurthy and DeWitt proposed that database systems should address this trade-off by maintaining two copies of data, one laid out in column-major order and the other in row-major order, to ensure efficient accesses in both dimensions [15]. This approach, however, not only doubles the storage requirements, but also makes updates difficult; they must propagate to two copies that are laid out differently. *Atropos* provides efficient accesses in both dimensions with only one copy of the data.

Denehy et al. [7] proposed the ExRAID storage interface that exposes some information about parallelism and failure-isolation boundaries of a disk array. This information was shown to allow application software to control data placement and to dynamically balance load across the independent devices in the disk array. ExRAID exposed coarse boundaries that were essentially entire volumes, which could be transparently spread across multiple devices. *Atropos*, on the other hand, defines a particular approach to spreading data among underlying devices and then exposes information about its specifics. Doing so allows applications to benefit from storage-managed parallelism and redundancy, while optionally exploiting the exposed information to orchestrate its data layouts and access patterns.

## 7 Conclusions

The *Atropos* disk array LVM employs a new data organization that allows applications to take advantage of features built into modern disks. Striping data in track-sized units lets them take advantage of zero-latency access to achieve efficient access for sequential access patterns. Taking advantage of request scheduling and knowing exact head switch times enables semi-sequential access, which results in efficient access to diagonal sets of non-contiguous blocks.

By exploiting disk characteristics, a new data organization, and exposing high-level constructs about this organization, *Atropos* can deliver efficient accesses for database systems, resulting in up to 4× speed-ups for decision support workloads, without compromising performance of OLTP workloads.

## Acknowledgements

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, 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, and by an IBM faculty partnership award.

## References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. 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] J. S. Bucy and G. R. Ganger. *The DiskSim simulation environment version 3.0 reference manual*. Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[3] M. J. Carey et al. 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.

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

[5] G. P. Copeland and S. Khoshafian. A decomposition storage model. *ACM SIGMOD International Conference on Management of Data* (Austin, TX, 28–31 May 1985), pages 268–279. ACM Press, 1985.

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

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. *Summer USENIX Technical Conference* (Monterey, CA, 10–15 June 2002), pages 177–190, 2002.

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

[9] G. G. Gorbatenko and D. J. Lilja. *Performance of two-dimensional data models for I/O limited non-numeric applications*. Laboratory for Advanced Research in Computing Technology and Compilers Technical report ARCTiC–02–04. University of Minnesota, February 2002.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2003.

[11] R. Y. Hou and Y. N. Patt. Track piggybacking: an improved rebuild algorithm for RAID5 disk arrays. *International Conference on Parallel Processing* (Urbana, Illinois), 14–18 August 1995.

[12] D. M. Jacobson and J. Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.

[13] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77, 1987.

[14] C. R. Lumb, J. Schindler, and G. 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.

[15] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *International Conference on Very Large Databases* (Hong Kong, China, 20–23 August 2002), pages 430–441. Morgan Kaufmann Publishers, Inc., 2002.

[16] A. L. N. Reddy and P. Banerjee. A study of parallel disk organizations. *Computer Architecture News*, **17**(5):40–47, September 1989.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[18] K. Salem and H. Garcia-Molina. Disk striping. *International Conference on Data Engineering* (Los Angeles, CA), pages 336–342. IEEE, Catalog number 86CH2261-6, February 1986.

[19] J. Schindler. *Matching application access patterns to storage device characteristics*. PhD thesis. Carnegie Mellon University, 2004.

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

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

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

[23] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990.

[24] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. *Clotho: decoupling memory page layout from storage organization*. Technical report CMU–PDL–04–102. Carnegie-Mellon University, Pittsburgh, PA, February 2004.

[25] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[26] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. 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.

## A Access Efficiency Calculations

Let $T(N,K)$ be the time it takes to service a request of $K$ sectors that fit onto a single track of a disk with $N$ sectors per track (i.e., track-aligned access). Ignoring seek, and assuming no zero-latency access, this time can be expressed as

$$T_{nzl}(N,K) = \frac{N-1}{2N} + \frac{K}{N}$$

where the first term is the average rotational latency, and the second term is the media access time. For disks with zero-latency access, the first term is not constant; rotational latency decreases with increasing $K$. Thus,

$$T_{zl}(N,K) = \frac{(N-K+1)(N+K)}{2N^2} + \frac{K-1}{N}$$

These expressions are derived elsewhere [19].

The efficiency of track-based access is the ratio between the raw one revolution time, $T_{rev}$, and the time it takes to read $S = kN$ sectors for some large $k$. Hence,

$$E_n = \frac{kT_{rev}}{T_n(N,N) + (k-1)(T_{hs} + T_{rev})} \approx \frac{kT_{rev}}{k\,(T_{hs} + T_{rev})}$$

where $T_n(N,N)$ is the time to read data on the first track, and $(k-1)(T_{hs} + T_{rev})$ is the time spent in head switches and accessing the remaining tracks. In the limit, the access efficiency is

$$E_n(N,H) = 1 - \frac{H}{N} \tag{6}$$

which is the maximal streaming efficiency of a disk.

The maximal efficiency of semi-sequential quadrangle access is simply

$$E_q(N,H) = \frac{T_{rev}}{T_q(N,S)} = \frac{T_{rev}}{T_{zl}\,(N, db + (d-1)H)} \tag{7}$$

with $d$ and $b$ set accordingly.

## B Relaxing the one-revolution constraint

Suppose that semi-sequential access to $d$ blocks, each of size $b$, from a single quadrangle takes more than one revolution. Then the inequality in Equation 1 will be larger than 1. With probability $1/N$, a seek will finish with disk heads positioned exactly at the beginning of the $b$ sectors mapped to the first track (the upper left corner of the quadrangle in Figure 12). In this case, the disk will access all $db$ sectors with maximal efficiency (only incurring head switch of $H$ sectors for every $b$-sector read).

However, with probability $1 - 1/N$, the disk heads will land somewhere "in the middle" of the $b$ sectors after a seek, as illustrated by the arrow in Figure 12. Then, the access will incur a small rotational latency to access the beginning of the nearest $b$ sectors, which are, say, on the $k$-th track. After this initial rotational latency, which is, on average, equal to $(b-1)/2N$, the $(d-k)b$ sectors mapped onto $(d-k)$ tracks can be read with maximal efficiency of the semi-sequential quadrangle access.

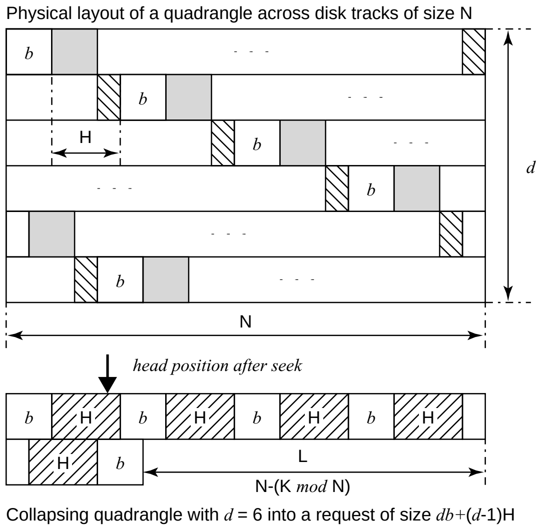

Physical layout of a quadrangle across disk tracks of size N
b
H
d
N
head position after seek
H
L
N-(K mod N)
Collapsing quadrangle with d = 6 into a request of size db+(d-1)H

Figure 12: **An alternative representation of quadrangle access.**

To read the remaining $k$ tracks, the disk heads will need at be positioned to the beginning of the $b$ sectors on the first track. This will incur a small seek and additional rotational latency of $L/N$. Hence, the resulting efficiency is much lower than when the one-revolution constraint holds, which avoids this rotational latency.

We can express the total response time for quadrangle access without the one-revolution constraint as

$$T_q(N,S) = \frac{b-1}{2N} + \frac{K}{N} + P_{lat}\frac{L}{N} \tag{8}$$

where $P_{lat} = (N-H-b-1)/N$ is the probability of incurring the additional rotational latency after reading $k$ out of $d$ tracks, $K = db - (d-1)H$ is the effective request size, $L = N - (K \bmod N)$, and $S = db$ is the original request size. To understand this equation, it may be helpful to refer to the bottom portion of Figure 12.

The efficiencies of the quadrangle accesses with and without the one-revolution constraint are approximately the same when the time spent in rotational latency and seek for the unconstrained access equals to the time spent in rotational latency incurred during passing over $dR$ residual sectors. Hence,

$$\frac{dR}{N} = \frac{N-1}{N}\left(\frac{N-1}{2N} + Seek\right)$$

Ignoring seek and approximating $N-1$ to be $N$, this occurs when $R \neq 0$ and

$$d \approx \frac{N}{2R}.$$

Thus, in order to achieve the same efficiency for the non-constrained access, we will have to access at least $d$ *VLBN*s. However, this will significantly increase I/O latency. If $R = 0$ i.e., when there are no residual sectors, the one-revolution constraint already yields the most efficient quadrangle access.