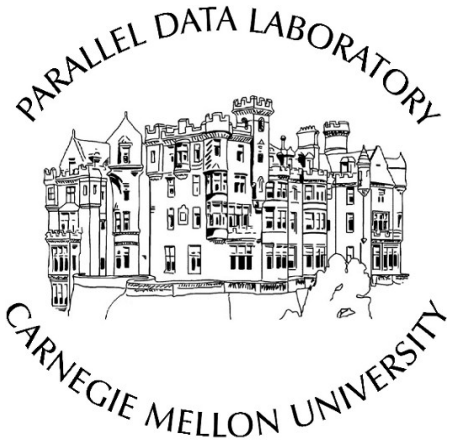


Clotho: Decoupling page layout from storage organization

Minglong Shao, Jiri Schindler, Steven W. Schlosser,
Anastassia Ailamaki, Gregory R. Ganger

CMU-PDL-04-102

March 2004

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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, as well as by an IBM faculty partnership award.

Keywords: Disk arrays, disk performance, database access

Abstract

As database application performance depends on the utilization of the disk and memory hierarchy, and the speed gap between the processor and memory components widens, smart data placement plays a central role in increasing locality and in improving memory utilization. Existing techniques, however, do not optimize accesses to all levels of memory hierarchy and for all the different workloads, because each storage level uses different technology (cache, memory, disks) and each application accesses data using different (often conflicting) patterns. This paper introduces Clotho, a new buffer pool and storage management architecture. Clotho decouples in-memory page layout from data organization on non-volatile storage devices, enabling independent data layout design at each level of the storage hierarchy. Using Clotho, a DBMS can maximize cache and memory utilization by (a) transparently using appropriate data layouts on memory and non-volatile storage, and (b) dynamically synthesizing data pages to follow application access patterns at each level as needed. Clotho enables (a) independently-tailored page layouts for dynamically changing as well as compound workloads, and (b) use of alternative technologies at each level (e.g., disk arrays or MEMS-based storage devices). We describe the Clotho design and implementation using disk array logical volumes and simulated MEMS-based storage devices, and we evaluate performance under a variety of workloads.

1 Introduction

Page structure and storage organization have been the subject of numerous studies [1, 3, 6, 10, 11], because they play a central role in database system performance. Research continues as no single data organization serves all needs within all systems. In particular, the access patterns resulting from queries posed by different workloads can vary significantly. One query, for instance, might access all the attributes in a table (*full-record access*), while another accesses only a subset of them (*partial-record access*). Full-record accesses are typical in transactional (OLTP) applications where insert and delete statements require the entire record to be read or written, whereas partial accesses are often met in decision-support system (DSS) queries. Moreover, when executing compound workloads, one query may access records sequentially while others access the same records "randomly" (e.g., via non-clustered index). Currently, database storage managers implement a single page layout and storage organization scheme, which is utilized by all applications running thereafter. As a result, in an environment using a variety of workloads, only a subset of query types can be serviced well.

Several data page layout techniques have been proposed in the literature, each targeting different query types. Notably, the N-ary Storage Model (*NSM*) [13] stores records consecutively, optimizing for full-record accesses, while penalizing partial-record sequential scans. By contrast, the Decomposition Storage Model (*DSM*) [7] stores values of each attribute in a separate table, optimizing for partial-record accesses, while penalizing queries that need the entire record. More recently, *PAX* [1] optimizes cache performance, but not memory utilization. Fractured mirrors [14] reduce *DSM*'s record reconstruction cost by using an optimized structure and scan operators, but need to keep an *NSM*-organized copy of the database as well to support queries that access full records. None of the previously proposed schemes provides a universally efficient solution, however, because they all make a fundamental assumption that the pages used in main memory must have the same contents as those stored on disk.

This paper proposes *Clotho*, a buffer pool and storage management architecture that decouples the memory page layout from the non-volatile storage data organization, allowing memory page contents to be determined dynamically according to queries being served. This decoupling offers two significant advantages. First, it allows storage access and memory utilization to be optimized by fetching from storage only the data accessed by a given query. Second, it allows new two-dimensional storage mechanisms to be exploited to mitigate the trade-off between the *NSM* and *DSM* storage models. We experiment with *Clotho* using the *Atropos* disk array architecture [17] and choosing data layouts at each level of the memory hierarchy to match *NSM* where it performs best, match *DSM* where it performs best, and outperform both for access types in between (and for query mixes). Likewise, *Clotho* provides the same benefits for systems exploiting the internal parallelism of MEMS-based storage devices (MEMStores) [19, 22].

This paper also describes and evaluates a prototype implementation of *Clotho* within the Shore database storage manager [4]. Experiments with disk arrays show that, with only a single storage organization, performance of DSS and OLTP workloads is comparable to the page layouts best suited for the respective workload (i.e., *DSM* and *PAX*, respectively). Experiments with a simulated MEMStore confirm that similar benefits will be realized with these future devices as well.

The remainder of this paper is organized as follows. Section 2 gives background and related work. Section 3 describes the architecture of a database system that enables decoupling of the in-memory and storage layouts. Section 4 describes the design of a buffer pool manager that supports

query-specific in-memory page layout. Section 5 describes the design of a volume manager that allows efficient access when an arbitrary subset of table attributes are needed by a query. Section 6 describes our initial implementation, and Section 7 evaluates this implementation for several database workloads using both a disk array logical volume and a simulated MEMStore.

2 Background and related work

Conventional relational database systems such as Oracle or IBM DB2 store data in fixed-size pages (typically 4 to 64 KB). To access individual records of a relation (table) requested by a query, a scan operator of a database system accesses main memory. Before accessing data, a page must first be fetched from non-volatile storage (e.g., a logical volume of a disk array) into main memory. Hence, a page is the basic allocation and access unit for non-volatile storage. This access is facilitated by the database's storage manager, which sends a request to a storage device logical volume manager to fetch the necessary blocks.

A single database page contains a header describing what records are contained within and how they are laid out. In order to retrieve data requested by a query, a scan operator must understand the page layout, (a.k.a. storage model). Since the page layout determines what records and which attributes of a relation are stored in a single page, the storage model employed by a database system has far reaching implications on the query performance of a particular workload [2].

The page layout prevalent in commercial database systems, called N-ary storage model (*NSM*), is optimized for queries with full-record access common in an on-line transaction processing (OLTP) workload. *NSM* stores all attributes of a relation in a single page [13] and full records are stored within a page one after another. Accessing a full record is accomplished by accessing a particular record from consecutive memory locations. Using an unwritten rule that access to consecutive logical blocks (*LBN*s) in the storage device is more efficient than random access, a storage manager maps single page to consecutive *LBN*s. Thus, an entire page can be accessed by a single I/O request.

An alternative page layout, called the Decomposition Storage Model (*DSM*) [7], is optimized for decision support systems (DSS) workloads. Since DSS queries typically access a small number of attributes and most of the data in the page is touched in memory by the scan operator, *DSM* stores only one attribute per page. To ensure efficient storage device access, a storage manager maps *DSM* pages with consecutive records containing the same attribute into extents of contiguous *LBN*s. In anticipation of a sequential scan through records stored in multiple pages, a storage manager can prefetch all pages in one extent with a single large I/O, which is more efficient than accessing each page individually by a separate I/O.

A page layout optimized for CPU cache performance, called *PAX* [1], offers good CPU-memory performance for both individual attribute scans of DSS queries and full-record accesses in OLTP workloads. The *PAX* layout partitions data across into separate minipages. A single minipage contains data of only one attribute and occupies consecutive memory locations. Collectively, a single page contains all attributes for a given set of records. Scanning individual attributes in *PAX* accesses consecutive memory locations and thus can take advantage of cache-line prefetch logic. With proper alignment to cache-line sizes, a single cache miss can effectively prefetch data for several records, amortizing the high latency of memory access compared to cache access. However, *PAX* does not address memory-storage performance.

All of the described storage models share the same characteristics. They (i) are highly optimized for one workload type, (ii) focus predominantly on one level of the memory hierarchy, (iii) use a static data layout that is determined *a priori* when the relation is created, and (iv) apply the same layout across all levels of the memory hierarchy, even though each level has unique (and very different) characteristics. As a consequence, there are inherent performance trade-offs for each layout that arise when a workload changes. For example, *NSM* or *PAX* layouts waste memory capacity and storage device bandwidth for DSS workloads, since most data within a page is never touched. Similarly, a *DSM* layout is inefficient for OLTP queries accessing random full records. To reconstruct a full record with n attributes, n pages must be fetched and $n-1$ joins on record identifiers performed to assemble the full record. In addition to wasting memory capacity and storage bandwidth, this access is inefficient at the storage device level; accessing these pages results in random one-page I/Os. In summary, each page layout exhibits good performance for a specific type of access at a specific level of memory hierarchy.

Several researchers have proposed solutions to address these performance trade offs. Ramamurthy et al. proposed fractured mirrors that store data in both *NSM* and *DSM* layouts [14] to eliminate the need to reload and reorganize data when access patterns change. Based on the workload type, a database system can choose the appropriate data organization. Unfortunately, this approach doubles the required storage space and complicates data management; two physically different layouts must be maintained in synchrony to preserve data integrity. Hankins and Patel [10] proposed data morphing as a technique to reorganize data within individual pages based on the needs of workloads that change over time. Since morphing takes place within memory pages that are then stored in that format on the storage device, these fine-grained changes cannot address the trade-offs involved in accessing non-volatile storage. The *Lachesis* database storage manager [15] exploits unique disk drive characteristics to improve performance of DSS workloads and compound workloads that consist of DSS and OLTP queries competing for the same storage device. It matches page allocation and access policies to leverage these characteristics, but the storage model itself is not different; *Lachesis* stores transparently the in-memory *NSM* pages to the storage device's logical blocks.

MEMStore [5] is a promising new technology for on-line storage that has been shown to provide efficient accesses to two-dimensional data. Schlosser et al. proposed data layout for MEMStores that exploits their inherent access parallelism [19]. Yu et al. devised an efficient mapping of database tables to this layout that takes advantage of MEMStore's unique characteristics [22] to improve query performance. Similarly, Gorbatenko and Lilja [9] proposed data organization for two-dimensional access to database tables mapped to individual disk drives. However, these initial works did not explore the implications of this new data organization on in-memory access performance.

In summary, these solutions either address only some of the performance trade-offs or are applicable to only one level of the memory hierarchy. *Clotho* builds on the previous work and uses a decoupled data layout that can adapt to dynamic changes in workloads without the need to maintain multiple copies of data, reorganize data layout, or to compromise between memory and I/O access efficiency.

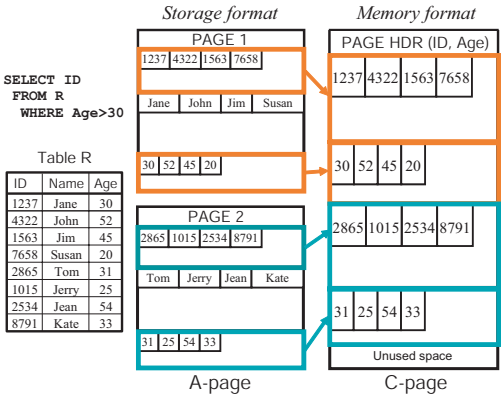


Figure 1: **Decoupled storage device and in-memory layouts.**

3 Decoupling data layouts

From the discussion in the previous section, it is clear that designing a static scheme for data placement in memory and non-volatile storage that performs well across different workloads and different device types and technologies is difficult. Instead of accepting the trade-offs inherent to a particular page layout that affects all levels of the memory hierarchy, we propose a new approach.

As the technology used at each storage level exhibits vastly different performance characteristics, the data organization at each level should be different. *Clotho* decouples the in-memory data layout from the storage organization and implements the most appropriate data layout tailored to each level of the memory hierarchy, without compromising performance at the other levels. The challenge is to ensure that this decoupling works seamlessly within the framework of current DBMS architectures. This section introduces the different data organizations *Clotho* employs at each level of the storage hierarchy, outlines the benefits from using this architecture, and describes its key components.

3.1 Data organization in *Clotho*

Clotho allows for decoupled data layouts and different representations of the same table at the memory and storage levels. Figure 1 depicts an example table, *R*, with three attributes: `ID`, `Name`, and `Age`. At the storage level, the data is organized into A-pages. An A-page contains all attributes of the records; only one A-page needs to be fetched to retrieve a full record. Exploiting the idea used in *PAX* [1], an A-page organizes data into minipages that group values from the same attribute for efficient predicate evaluation, while the rest of the attributes are in the same A-page. To ensure that the record reconstruction cost is minimized regardless of the size of the A-page, *Clotho* allows the device to use optimized methods for placing the contents of the A-page onto the storage medium. Therefore, not only does *Clotho* fully exploit sequential scan for evaluating predicates, but it also places A-pages carefully on the device to ensure near-sequential (or semi-sequential [17]) access when reconstructing a record. The placement of A-pages on the disk is further explained in Section 5.1.

The rightmost part of Figure 1 depicts a C-page, which is the in-memory representation of a page. The page frame is sized by the buffer pool manager and is on the order of 8 KB. A C-page

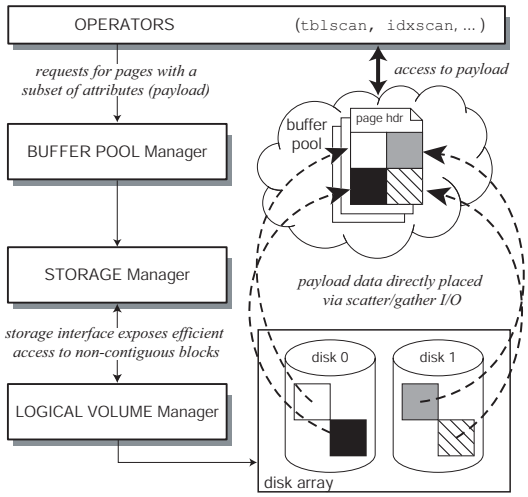


Figure 2: **Clotho architecture.**

is similar to an A-page in that it also contains attribute values grouped in minipages, to maximize processor cache performance. Unlike an A-page, however, a C-page *only* contains values for the attributes the query accesses. Since, the query in the example only uses the `ID` and `Age`, the C-page only includes these two attributes, maximizing memory utilization. Note that the C-page uses data from two A-pages to fill up the space "saved" from omitting `Name`. In the rest of this paper, we refer to the C-page layout as the *Clotho* storage model (*CSM*).

3.2 System architecture

The difficulty in building a database system that can decouple in-memory page layout from storage organization lies in implementing the necessary changes without undue increase in system and code complexity. To allow decoupled data layouts, *Clotho* changes parts of three database system components, namely the buffer pool and the device manager. The changes span limited areas in these components, and do not alter the query processing interface. Each component can independently take advantage of enabling hardware/OS technologies at each level of the memory hierarchy, while hiding the details from the rest of the system. Figure 2 shows the basic architecture of *Clotho*. This section outlines the changes to a standard database system, which are further explained in Sections 4 and 5.1. Specific prototype implementation details are provided in Section 6.

The operators are essentially predicated scan and store procedures that access data from in-memory pages stored in a common buffer pool. They take advantage of the query-specific page layout of C-pages that leverages the L1/L2 CPU cache characteristics and cache prefetch logic for efficient access to data.

The buffer pool manager manages C-pages in the buffer pool and enables sharing across different queries that need the same data. In traditional buffer pool managers, a buffer page is assumed to have the same schema and contents as the corresponding relation. In *Clotho*, however, this page may contain a subset of the table schema attributes. To ensure sharing, correctness during updates,

and high memory utilization, the *Clotho* buffer pool manager maintains a page-specific schema that denotes which attributes are stored within each buffered page (i.e., the page schema). The challenge of this approach is to ensure minimal I/O by determining sharing and partial overlapping across concurrent queries with minimal book-keeping overhead. Section 4 details the buffer pool manager operation in detail.

The storage manager maps A-pages to specific logical volume's logical blocks, called *LBN*s. Since the A-page format is different from the in-memory layout, the storage manager rearranges A-page data on-the-fly into C-pages using the query-specific *CSM* layout. Unlike traditional storage managers where pages are also the smallest access units, the *Clotho* storage manager selectively retrieves a portion of a single A-page. With scatter/gather I/O and hardware direct memory access (DMA), the pieces of individual A-pages can be delivered directly into the proper memory frame(s) in the buffer pool as they arrive from the logical volume. The storage manager simply sets up the appropriate I/O vectors with the destination address ranges for the requested *LBN*s. The data is placed directly to its destinations without the storage manager's involvement or the need for data shuffling and extraneous memory copies. To efficiently access data under a variety of access patterns, *Clotho* storage manager relies on explicit hints provided by the logical volume manager that convey which *LBN*s can be accessed together efficiently and uses them to allocate A-pages.

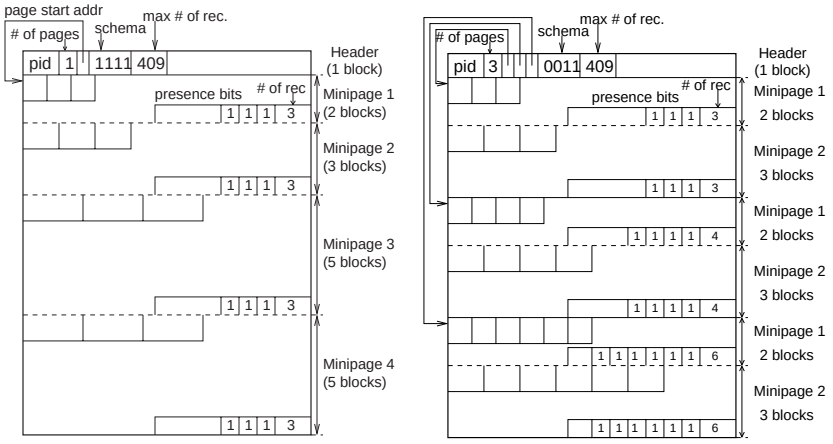
The logical volume manager (LVM) maps volume *LBN*s to the physical blocks of the underlying storage device(s). It leverages device-specific characteristics to create mappings that yield efficient access to a collection of *LBN*s. The LVM exposes the information about these *LBN* collections through an interface that, while abstracting away the device specifics, allows the *Clotho* storage manager to employ allocation policies for efficient access. In particular, the storage interface exports two functions that establish explicit relationships between individual *LBN*s of the logical volume and enable the storage manager to effectively map A-pages to individual *LBN*s. One function returns the set of consecutive *LBN*s that yield efficient access (e.g., all blocks mapped onto one disk track). Another function returns a set of non-contiguous *LBN*s that can be efficiently accessed together (e.g., parallel-accessible *LBN*s mapped to different disks of logical volume). The LVM is briefly described in Section 5.1 and detailed elsewhere [17, 19].

3.3 Benefits of decoupled data layouts

The concept of decoupling data layouts at different levels of the memory hierarchy offers several benefits.

Leveraging unique device characteristics. At the volatile (main-memory) level, *Clotho* uses *CSM*, a data layout that maximizes processor cache utilization by minimizing unnecessary accesses to memory. *CSM* organizes data in C-pages and also groups attribute values to ensure that only useful information is brought into the processor caches [1, 10]. At the storage-device level, the granularity of accesses is naturally much coarser. The objective is to maximize memory utilization for all types of queries by only bringing into the buffer pool data that the query needs.

Query-specific memory layout. With memory organization decoupled from storage layout, *Clotho* can decide what data is needed by a particular query, request only the needed data from a storage device, and arrange the data on-the-fly to an organization that is best suited for the particular query needs. This fine-grained control over what data is fetched and stored also puts less pressure on buffer pool and storage system resources. By not requesting data that will not be needed, a storage device can devote more time to servicing requests for other queries executing concurrently and



(a) **Full records.** (b) **Partial records.**

Figure 3: **C-page layout.**

hence speed up their execution.

Dynamic adaptation to changing workloads. A system with flexible data organization does not experience performance degradation when query access patterns change over time. Unlike systems with static page layouts, where the binding of data representation to workload occurs during table creation, this binding is done in *Clotho* only during query execution. Thus, a system with decoupled data organizations can easily adapt to changing workloads and also fine-tune the use of available resources when they are under contention.

4 The *Clotho* buffer pool manager

As explained in Section 3.1, the *Clotho* buffer pool manager organizes data in C-pages using the *CSM* data layout. *CSM* is a query-optimized in-memory page layout that stores only the subset of attributes needed by a query. Consequently, a C-page can contain a single attribute (similar to *DSM*), a few attributes, or all attributes of a given set of records (similar to *PAX*) depending on query needs. This section describes how the buffer pool manager constructs and maintains C-pages and ensures data sharing and consistency in the buffer pool.

4.1 In-memory C-page layout

Figure 3 depicts two examples of C-pages for a table with four attributes. In our design, C-pages only contain fixed-size attributes. Variable-size attributes are stored separately in other page layouts (see Section 6.2). A C-page contains a page header and a set of minipages, each containing data for one attribute and collectively holding all attributes needed by queries. In a minipage, a single attribute's values are stored in consecutive memory locations, to maximize processor cache performance. The current number of records and presence bits are distributed across the minipages. Because the C-page only handles fixed-size attributes, the size of each minipage is determined at the time of table creation.

The page header stores the following information: page id of the first A-page, the number of partial A-pages contained, the starting address of each A-page, a bit vector indicating the schema of the C-page's contents, and the maximal number of records that can fit in an A-page.

Figure 3(a) and Figure 3(b) depict C-pages with complete and partial records, respectively. The leftmost C-page is created for queries that access full records, whereas the rightmost C-page is customized for queries touching only the first two attributes. The space for minipages 3 and 4 on the left are used to store more partial records from additional A-pages on the right. In this example, a single C-page can hold the requested attributes from three A-pages, increasing memory utilization by a factor of three.

On the right side of the C-page we list the number of storage device blocks each minipage occupies. In our example each block is 512 bytes. Depending on the relative attribute sizes, as we fill up the C-page using data from more A-pages there may be some unused space. Instead of performing costly operations to fill up that space, we choose to leave it unused. Our experiments show that, with the right page size and aggressive prefetching, this unused space does not cause a detectable performance deterioration (details about space utilization are in Section 7.6).

4.2 Data sharing in buffer pool

Concurrent queries do not necessarily access the same sets of attributes; concurrent sets of accessed attributes may be disjoint, inclusive, or otherwise overlapping. The *Clotho* buffer pool manager must (a) maximize sharing, ensuring memory space efficiency, (b) minimize book-keeping to keep the buffer pool operations light-weight, and (c) maintain consistency in the presence of updates.

As an example, query Q1 asks for attributes a_1 and a_2 while query Q2 asks for attributes a_2 and a_3. Using a simple approach, the buffer manager could create two separate C-pages tailored to each query. This approach ignores the sharing possibilities in case these queries scan the table concurrently. To achieve better memory utilization, the buffer manager can instead dynamically reorganize the minipages of a_1 and a_2 inside the two C-pages, fetching only the needed values and keeping track of the progress of each query, dynamically creating new C-pages for Q1 and Q2. However, this approach incurs too much book-keeping overhead, and is inefficient in practice.

The *Clotho* buffer pool manager balances memory utilization and management complexity. Each frame in the buffer pool stores a C-page which conforms to a *page schema*, a bitvector that describes which attributes the C-page holds. For each active table, we keep a list of the different page schemas for C-pages that belong to the table and are currently in the buffer pool. Finally, each active query keeps a *query schema*, a bitvector that describes which attributes the query needs for each accessed table. Whenever a query starts executing, the buffer pool manager notes the query schema and inspects the other, already active, page schemas. If the new query schema accesses a disjoint set of attributes from the over active queries, if any, the buffer pool manager creates a new C-page. Otherwise, it merges the new schema with the most-efficient overlapping one already in memory. The algorithm in Figure 4 modifies the page schema list (p_sch), which is initially empty, based on the query schema (q_sch). Once the query is complete, the system removes the corresponding query schema from the list and adjusts the page schema list accordingly using the currently active query schemas.

During query execution the page schema list dynamically adapts to changing workloads depending on the concurrency degree and the overlaps among attribute sets accessed by queries. This list ensures that queries having common attributes can share data in the buffer pool while queries

if read-only query **then**
  if $\exists$p_sch $\supseteq$ q_sch **then**
    Do nothing
  else if q_sch $\cap$ allp_sch $= \varnothing$ **then**
    Add q_sch to the schema list
  else
    New p_sch = $\cup$(q_sch, $\{$p_sch $\mid$ p_sch $\cap$ q_sch $\neq \varnothing\}$)
    Add the new p_sch to the list
  end if
else if it is a write query (update/delete/insert) **then**
  Use full schema as the q_sch
  Modify the list: only one full p_sch now
end if

Figure 4: **Buffer pool manager algorithm.**

with disjoint attributes will not affect each other. In the above example, Q1 first comes along, the buffer pool manager creates C-pages with a_1 and a_2. When Q2 arrives, the buffer pool manager will create a C-page with a_1, a_2, and a_3 for these two queries. After Q1 finishes, C-pages with only a_2 and a_3 will be created for Q2.

4.3 Maintaining data consistency

With the algorithm in Figure 4, the buffer pool may still have multiple copies of the same minipage. To ensure data consistency, if a transaction modifies a C-page, other queries should be able to get the latest copy if it is still resident in memory.

When looking for a record, a traditional database buffer manager looks for the corresponding page id in the page table, and determines whether the record is in memory. To support record lookup in the query-specific C-page in *Clotho*, the page table of the buffer pool manager contains the page ids of all the A-pages used to construct the active C-pages, and is augmented with the page schema bitvectors. To perform a record lookup, we use a key consisting of the page id and the page schema requested. A hit means that the page id matches one of the A-page page ids, and the schema of the C-page subsumes the schema of the requested record as described in the key.

For writes (insert, delete, and update operations), we use full-schema C-pages for two reasons: (a) inserts and deletes need full-record access and modify all respective minipages, whereas (b) for updates, full schema is convenient for maintaining the data consistency in the buffer pool with nearly no additional cost. Queries asking for modified records can automatically obtain the correct dirty page from the buffer pool.

When a write query is looking up a C-page, it invalidates all of the other buffered C-pages that contain the parts of the same A-page. Thus, there is only one valid copy of the modified data. Since the C-page with updated data has a full schema, the updated page subsumes all other queries asking for records in this page, which will use it until it is flushed to the disk.

5 Logical volume manager

This section briefly describes the storage device-specific data organization and the mechanisms exploited by the LVM in creating logical volumes that consist of either disk drives or a single MEMStore. Much of this work builds upon our previous work on *Atropos* disk array logical volume manager [17] and MEMStore [19]. Due to space constraints, this section describes the high-level points for each device type.

5.1 Atropos disk array LVM

The standard interface of disk drives and disk arrays uses a simple linear abstraction, meaning that any two dimensional data structure that is to be stored on disk needs to be serialized. For example, *NSM* serializes along full records (row-major) and *DSM* serializes along single attributes (column-major). Once the table is stored, access along the dimension of serialization is sequential and efficient. However, access along the other dimension is random and inefficient. *Atropos* uses the same linear abstraction as before, but solves this problem by using a new internal data organization and exposing a few key parameters to the higher-level software.

By exposing enough information about its data organization, the database's storage manager can achieve efficient access along either dimension. *Atropos* exploits the request scheduler built into the disk's firmware and automatically-extracted knowledge of track switch delays to support semi-sequential access: diagonal access to ranges of blocks (one range per track) across multiple adjacent disk tracks. This second dimension of access enables two dimensional data structures to be accessed efficiently. To improve sequential access, *Atropos* exploits automatically-extracted knowledge of disk track boundaries, using them as its stripe unit boundaries for achieving efficient sequential access. By also exposing these boundaries explicitly, it allows a storage manager like *Lachesis* [15] to use previously proposed "track-aligned extents" (*traxtents*), which provide substantial benefits for streaming patterns interleaved with other I/O activity [16]. Finally, as other striped logical volume managers, it delivers aggregate bandwidth of all of the disks in the volume and offers the same reliability/performance tradeoffs of traditional RAID schemes [12].

5.2 Semi-sequential access

To understand semi-sequential access, imagine sending two requests to a disk: one request for the first *LBN* on the first track, and one for the second *LBN* on the second track. These two adjacent tracks are in the same cylinder, but different heads are used to access them. First, the disk heads will seek to the cylinder, then there will be some initial rotational latency before the first *LBN* is read. Next, the disk will switch heads, which takes some fixed amount of time (typically around 1 ms), and access the second *LBN*. With properly chosen *LBN*s, the *Atropos* layout guarantees that the second *LBN* is accessible after only a head switch, with no additional seek or rotational latency incurred. Requesting more *LBN*s on successive tracks laid out in this fashion allows further semi-sequential access, with only a single head switch between each access.

Atropos limits the number of *LBN*s that are semi-sequentially mapped to a single disk before mapping them to the next disk. Requests to the semi-sequential *LBN*s on a single disk are all issued in a batch. The disk's internal scheduler then chooses a request that will incur the smallest positioning cost and service it first. Servicing all other requests will incur only a head switch

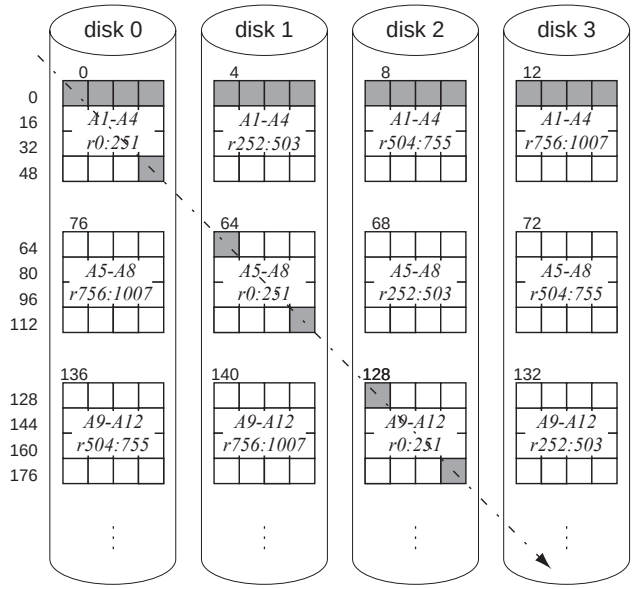


Figure 5: **Mapping of a database table with 12 attributes onto *Atropos* logical volume with 4 disks.** The numbers to the left of disk 0 are the *LBN*s mapped to the gray disk locations connected by the arrow and not the first block of each row. The arrow illustrates efficient semi-sequential access fetching single A-page with 63 records. A single sequential I/O for 16 *LBN*s can efficiently fetch one attribute from 1008 records striped across four disks.

to the adjacent track, while bounding the response time to a single rotation. When more *LBN*s are requested, they can be accessed in parallel from multiple disks, each disk accessing semi-sequentially a subset of the requested *LBN*s. Naturally, the sustained bandwidth of semi-sequential access is less than that of sequential access. However, semi-sequential access is more efficient than reading randomly chosen *LBN*s spread across adjacent tracks, as would be the case when accessing data along the secondary dimension of a table stored in a normal striped disk array. Accessing random *LBN*s would incur an additional rotational latency equal to half a revolution, on average.

5.3 Efficient database organization

Atropos allows database tables to be laid out onto the disks such that access to one dimension of the table is sequential, and access to the other dimension is semi-sequential. Figure 5 shows a simple table consisting of 1008 records, each with 12 attributes stored in an *Atropos* logical volume comprised of four disks. In the figure, the primary dimension is along the columns of the table. Accessing one attribute of all records in the table is done with four track-sized, track-aligned reads. For example, a sequential scan of attribute A1 for all records in the table results in a read starting at *LBN* 0 through *LBN* 15. Accessing all attributes of a single record results in three semi-sequential accesses, one to each disk. For example, accessing attributes A1 through A12 of record 0 requires three semi-sequential reads, each proceeding in parallel on different disks, starting at *LBN*s 0, 64, and 128. In this case, access along the columns of the table is sequential and access along the rows of the table is semi-sequential. A single A-page is mapped to a set of semi-sequential *LBN*s to allow efficient access.

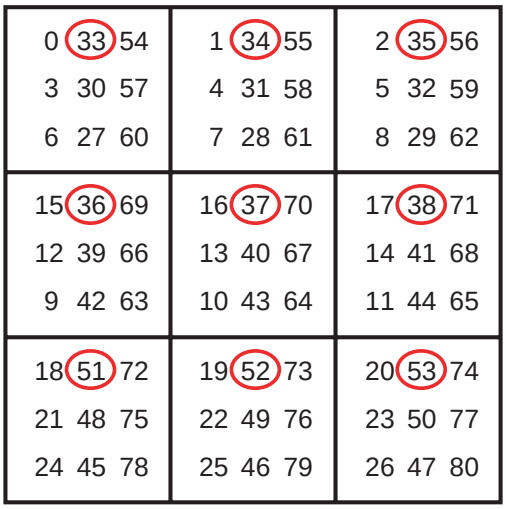


Figure 6: **Data layout with parallel-accessible *LBN*s highlighted.** The *LBN*s marked with ovals are at the same location within each square and, thus, comprise an equivalence class. That is, they can potentially be accessed in parallel.

5.4 MEMS-based storage devices

Two groups have evaluated the use of internal access parallelism in MEMStores to efficiently access database tables [19, 22]. *Clotho* shows the benefit of using these techniques in the context of a complete database management system. As these devices are not yet available, we simulate their behavior using the DiskSim simulator combined with the *Atropos* logical volume manager.

Most MEMStore designs [5, 21] consist of a media sled and an array of several thousand probe tips. Actuators position the spring-mounted media sled in the X-Y plane, and the stationary probe tips access data as the sled is moved in the Y dimension. Each read/write tip accesses its own small portion of the media, which naturally divides the media into *squares* and reduces the range of motion required of the media sled.

When a seek occurs, the media is positioned to a specific offset relative to the entire read/write tip array. As a result, at any point in time, all of the tips access the same locations within their squares. An example of this is shown in Figure 6 in which *LBN*s at the same location within each square are identified with ovals. Realistic MEMStores are expected to have enough read/write tips to potentially access 100 *LBN*s in parallel. However, because of power and shared-component constraints, only about 10 to 20 of those *LBN*s could be *actually* accessed in parallel.

Given the simple device in Figure 6, if one third of the read/write tips can be active in parallel, a system could access together up to 3 *LBN*s out of the 9 shown with ovals. The three *LBN*s chosen could be sequential (e.g., 33, 34, and 35), or could be disjoint (e.g., 33, 36, and 51). In each case, all of those *LBN*s would be transferred to or from the media in parallel with the same efficiency. A-pages are arranged onto the rows and columns of read/write tips, much as they are across sequential and semi-sequential *LBN*s on a disk drive. By activating the appropriate read/write tips, parallel access to either the rows or columns of the table is possible. In contrast to a disk drive, access along the rows access along the columns are equally efficient. This differs from disk drives in which access to one dimension is less efficient than access to the other (semi-sequential vs. sequential).

6 *Clotho* implementation

Clotho is implemented within the Shore database storage manager [4]. This section describes the implementation of C-pages and scan operators, as well as variable-sized attributes and the LVM.

6.1 Creating and scanning C-pages

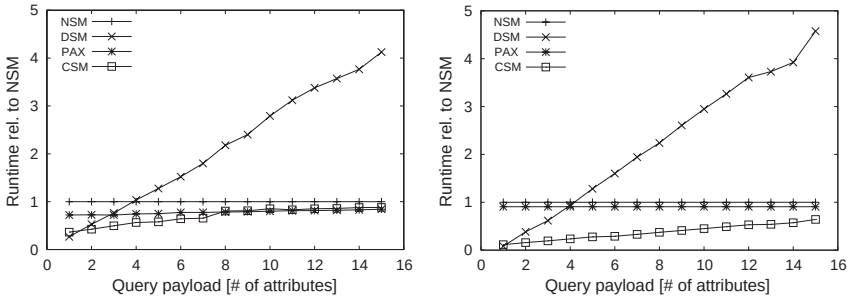
We implemented *CSM* as a new page layout in Shore, according to the format described in Section 4.1. The only significant change in the internal Shore page structure is that the page header is aligned to occupy one block (512 B in our experiments). As described in Section 4, the original buffer pool manager is augmented with schema management information to control and reuse C-page contents. These modifications were minor and limited to the buffer pool module. To access a set of records, a scan operator issues a request to the buffer pool manager to return a pointer to the the C-page with the (first of the) records requested. This pointer consists of the first A-page id in the C-page plus the page schema id.

If there is no appropriate C-page in the buffer pool to serve the request, the buffer pool manager allocates a new frame that will hold the requested page. It then fills the page header with schema information that allows the storage manager to determine which data is needed and issue the appropriate commands to the LVM. The partially-filled frame is called a *skeleton*. The storage manager completes the creation of the skeleton by determining how many and which minipages will be requested as well as their boundaries. This decision depends on the number of attributes in the payload and on their relative sizes. Based on the skeleton, the storage manager constructs a batch of I/O requests for the individual minipages and, upon completion, the requested blocks are "scattered" to their appropriate locations.

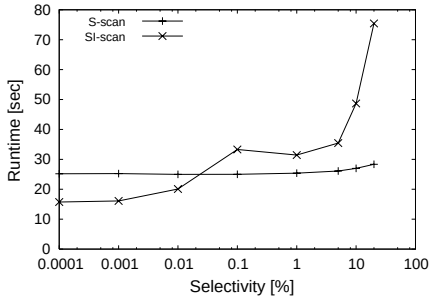
Two scan operators were implemented: S-scan is similar to a scan operator on *NSM* pages, with the only difference that it only scans the attributes accessed by the query. (in the predicate and in the payload). *Clotho* invokes S-scan to read tuples containing the attributes in the predicate and those in the payload, reads the predicate attributes, and if the condition is true returns the payload. The second scan operator, SI-scan, works similarly to an index scan. SI-scan first fetches and evaluates only the attributes in the predicates, then makes a list of the qualifying record ids, and finally retrieves the projected attribute values directly. Section 7.2.1 evaluates these two operators. To implement the above changes, we wrote about 2000 lines of C++ code.

6.2 Storing variable-sized attributes

Our current implementation stores fixed-sized and variable-sized attributes in separate A-pages. Fixed-sized attributes are stored in A-pages as described in Section 3.1. Each variable-sized attribute is stored in a separate A-page whose format is similar to a *DSM* page. To fetch the full record of a table with variable-sized attributes, the storage manager must issue one A-page-sized I/O to fetch the A-page containing all of the fixed-size attributes, and additional ones for each variable-sized attribute in the table. As future work, we plan to design storage of variable-sized attributes in the same A-pages as fixed-sized attributes using attribute size estimations [1] and overflow pages whenever needed.



(a) **Atropos disk LVM.** (b) **MEMS device.**



(c) **Scan operators performance.**

Figure 7: **Microbenchmark performance for different layouts.** The graphs show the total microbenchmark query run time relative to *NSM*. The performance of S-scan and SI-scan is shown for CSM layout running on *Atropos* disk array.

6.3 Logical volume manager

The *Atropos* logical volume manager is implemented as a standalone C++ application that communicates with Shore through a socket (control path) and shared memory (data path) to avoid data copies. *Atropos* determines how I/O requests are broken into individual disk I/Os. 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 with minimal impact on Shore execution.

Since real MEMStores do not exist yet, the MEMStore LVM implementation relies on simulation. It uses an existing model of MEMS-based storage devices [18] integrated into the DiskSim storage subsystem simulator [8]. The LVM process runs the I/O timings through DiskSim and uses main memory for storing data.

7 Evaluation

This section evaluates the benefits of decoupling in-memory data layout from storage device organization using our *Clotho* prototype. The evaluation is presented in two parts. The first part uses representative microbenchmarks [20] to perform a sensitivity analysis by varying several parame-

ters such as the query payload (projectivity) and the selectivity in the predicate. The microbenchmarks include queries (sequential and random access), point updates, and bulk insert operations. Microbenchmarks are useful to understand the behavior of the system and evaluate worst- and best-case scenarios. The second part of the section presents experimental results from running DSS and OLTP workloads, demonstrating the efficiency of *Clotho* when running these workloads with only one common storage organization.

7.1 Experimental setup

The experiments are conducted on a two-way 1.7 GHz Pentium 4 Xeon workstation running Linux kernel v. 2.4.24 and RedHat 7.1 distribution. The machine for the disk array experiment has 1024 MB memory and is equipped with two Adaptec Ultra160 Wide SCSI adapters, each controlling two 36 GB Seagate Cheetah 36ES disks (ST336706LC). The *Atropos* LVM exports a single 35 GB logical volume created from the four disks in the experimental setup and maps it to the blocks on the disks' outermost zone.

An identical machine configuration is used for the MEMStore experiments; it has 2 GB of memory, with half used as data store. The emulated MEMStore parameters are based on the G2 MEMStore [18] that includes 6400 probe tips that can simultaneously access 16 *LBN*s, each of size 512 bytes; the total capacity is 3.46 GB.

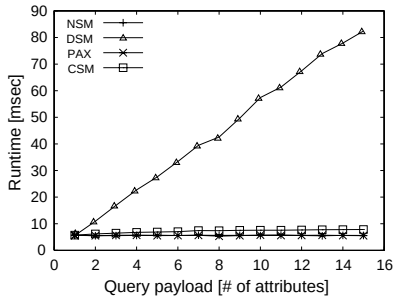
All experiments compare *CSM* to the *NSM*, *DSM*, and *PAX* implementations in Shore. *NSM* and *PAX* are implemented as described in [1], whereas *DSM* is implemented in a tight, space-efficient form as described in [14]. For *CSM*, the *Atropos* LVM uses its default configuration [17]. The *NSM*, *DSM*, or *PAX* page layouts don't take advantage of the semi-sequential access that *Atropos* provides. However, they still run over the logical volume which is effectively a conventional striped logical volume with the stripe unit size equal to individual disks' track size to ensure efficient sequential access. Unless otherwise stated, the buffer pool size in all experiments is set to 128 MB and page sizes for *NSM*, *PAX* and *DSM* are 8 KB. For *CSM*, both the A-page and C-page sizes are also set to 8 KB.

7.2 Microbenchmark performance

To establish *Clotho* baseline performance, we first run a range query of the form `SELECT AVG(a1), AVG(a2), ... FROM R WHERE Lo < a2 < Hi`. *R* has 15 attributes of type `FLOAT`, and is populated with 8 million records (roughly 1 GB of data). All attribute values are uniformly distributed. We show the results of varying the query's payload by increasing the number of attributes in the select clause from one up to the entire record, and the selectivity by changing the values of *Lo* and *Hi*. We first run the query using sequential scan, and then using a non-clustered index to simulate random access.

7.2.1 Queries using sequential scan

Varying query payload. Figure 7 shows the performance of the microbenchmark query with varying projectivity for four data layouts. The data are shown for a query with 10% selectivity; using 100% selectivity exhibits the same trends. For the projectivity analysis, we use S-scan operator on *CSM* pages.



(a) **Point queries.**

Runtime	
NSM	58 s
DSM	236 s
PAX	63 s
CSM	57 s

(b) **Full scan.**

Figure 8: **Microbenchmark performance for *Atropos* LVM.**

Clotho shows the best performance at both low and high projectivities. At low projectivity, *CSM* achieves comparable performance to *DSM*, which is the best page layout when accessing a small fraction of the record. The slightly lower runtime of *DSM* for the one attribute value in Figure 7(a) is caused by a limitation of the Linux operating system that prevents us from using DMA-supported scatter/gather I/O for large transfers[1]. As a result, it must read all data into a contiguous memory region and do an extra memory copy to "scatter" data to their final destinations. *DSM* does not experience this extra memory copy; its pages can be put verbatim to the proper memory frames. Like *DSM*, *CSM* effectively pushes the project to the I/O level. Attributes not involved in the query will not be fetched from the storage, saving I/O bandwidth, memory space, and accelerating query execution.

With increasing projectivity, *CSM* performance is better than or equal to the best case at the other end of the spectrum, i.e., *NSM* and *PAX*, when selecting the full record. *DSM*'s suboptimal performance at high projectivities is due to the additional joins needed between the table fragments spread out across the logical volume. *Clotho*, on the other hand, fetches the requested data in lock-step from the disk and places it in memory using *CSM*, maximizing spatial locality and eliminating the need for a join. *Clotho* performs a full-record scan over $3\times$ faster when compared to *DSM*. As shown in Figure 7(b), the MEMStore performance shows the same results.

Comparison of S-scan and SI-scan. Figure 7(c) compares the performance of the above query for the S-scan and SI-scan operators described in Section 6.1. We vary selectivity from 0.0001% to 20% with the above query using a payload of four attributes (the trend continues for higher selectivities). As expected, SI-scan exhibits better performance at low selectivities, whereas S-scan wins as the selectivity increases. The performance gain comes from the fact that only pages containing qualified records are processed. The performance deterioration of SI-scan with increasing selectivity is due to two factors. First, SI-scan must process a higher number of pages than S-scan. Because of the uniform distribution, at selectivity equal to 1.6%, all pages will have qualifying records. Second, for each qualifying record, SI-scan must first locate the page, then calculate the record address, while S-scan uses a much simpler same-page record locator. The optimizer can use SI-scan or S-scan depending on which one will perform best given the estimated selectivity.

[1]The size of an I/O vector for scatter/gather I/O in Linux is limited to 16 elements, while commercial UNIX-es support up to 1024 elements.

7.2.2 Point queries using random access

The worst-case scenario for *Clotho* data placement schemes is random point tuple access (access to a single record in the relation through a non-clustered index). As only a single record is accessed, sequential scan is never used; on the contrary, as the payload increases *Clotho* is penalized more by the semi-sequential scan through the disk to obtain all the attributes in the record. Figure 8(a) shows that, when the payload is only a few attributes, *Clotho* performs closely to *NSM* and *PAX*. As the payload increases the performance of *Clotho* becomes slightly worse (although it deteriorates much less that the performance of *DSM*).

7.2.3 Updates

Bulk updates (i.e., updates to multiple records using sequential scan) exhibit similar performance to queries using sequential scan, when varying either selectivity or payload. Similarly, point updates (i.e., updates to a single record) exhibit comparable performance across all data placement methods as point queries. *Clotho* updates single records using full-schema C-pages, therefore its performance is always 22% worse than *NSM*, regardless of payload. To alleviate this behavior, we are currently investigating efficient ways to use partial–record C-pages for updates as we do for queries. As with point queries, the performance of *DSM* deteriorates much faster.

7.2.4 Full table scans and bulk inserts

When scanning the full table (full-record, 100% selectivity) or when populating tables through bulk insertions, *Clotho* exhibits comparable performance to *NSM* and *PAX*, whereas *DSM* performance is much worse, which corroborates previous results [1]. Figure 8(b) shows the total runtime when scanning table *R* and accessing full records. The results are similar when doing bulk inserts. Our optimized algorithm issues track-aligned I/O requests and uses aggressive prefetching for all data placement methods. Because bulk loading is an I/O intensive operation, space efficiency is the only factor that will affect the relative bulk-loading performance across different layouts. The experiment is designed so that each layout is as space-efficient as possible (i.e., table occupies the minimum number of pages possible). *CSM* exhibits similar space efficiency and the same performance as *NSM* and *PAX*.

7.3 DSS workload performance

To quantify the benefits of decoupled layout for database workloads, we run the TPC-H decision support benchmark on our Shore prototype. The TPC-H dataset is 1 GB and the buffer pool size is 128 MB.

Figure 9 shows execution times relative to *NSM* for four representative TPC-H queries (two sequential scans and two joins). The leftmost group of bars represents TPC-H execution on *Atropos*, whereas the rightmost group represents queries run on a simulated MEMStore. *NSM* and *PAX* perform the worst by a factor of $1.24\times - 2.0\times$ (except for *DSM* in Q1) because they must access all attributes. The performance of *DSM* is better for all queries except Q1 because of the benchmark's projectivity. *CSM* performs best because it benefits from projectivity and avoids the cost of the joins that *DSM* must do to reconstruct records. Again, results on MEMStore exhibit the same trends.

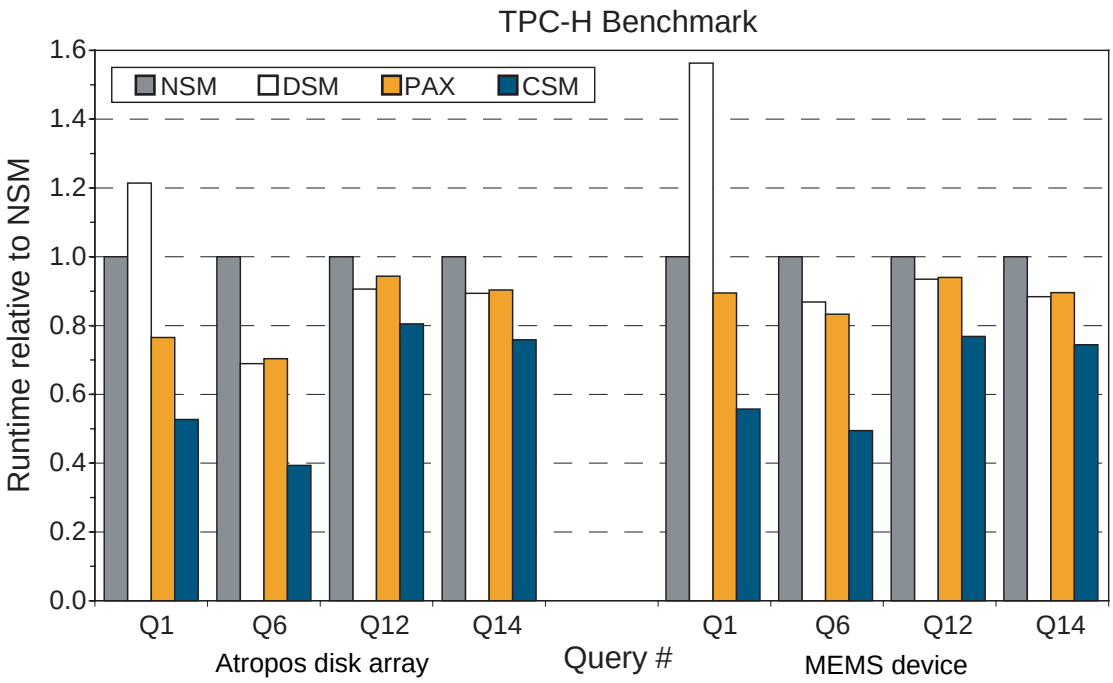


Figure 9: **TPC-H performance for different layouts.** The performance is shown relative to *NSM*.

7.4 OLTP workload performance

The queries in a typical OLTP workload access a small number of records spread across the entire database. In addition, OLTP applications have several insert and delete statements as well as point updates. With *NSM* or *PAX* page layouts, the entire record can be retrieved by a single-page random I/O, because these layouts map a single page to consecutive *LBN*s. *Clotho* spreads a single A-page across non-consecutive *LBN*s of the logical volume, enabling efficient sequential access when scanning a single attribute across multiple records and less efficient semi-sequential scan when accessing full records.

The TPC-C benchmark approximates an OLTP workload on our Shore prototype with all four data layouts using 8 KB page size. TPC-C is configured with 10 warehouses, 100 users, no think time, and 60 seconds warm-up time. The buffer pool size if 128 KB, so it only caches 10% of the database. The completed transactions per minute (TpmC) throughput is repeatedly measured over a period of 120 seconds.

Table 1 shows the results of running the TPC-C benchmark. As expected, *NSM* and *PAX* have comparable performance, while *DSM* yields much lower throughput. Despite the less efficient semi-sequential access, *CSM* observes only 6% lower throughput than *NSM* and *PAX*. *CSM* achieves this performance by taking advantage of the decoupled layouts to construct C-pages that are shared by the queries accessing only partial records. On the other hand, the frequent point updates penalize *CSM*'s performance: the semi-sequential access to retrieve full records. This penalty is in part compensated by the buffer pool manager's ability to create and share pages containing only the needed data. **Note to reviewers:** In the next version (still being implemented) the updates will also use partial C-pages and *Clotho* should be able to recover this penalty.

Layout	*NSM*	*DSM*	*PAX*	*CSM*
TpmC	1063	140	1090	1002

Table 1: **TPC-C benchmark results with *Atropos* disk array LVM.**

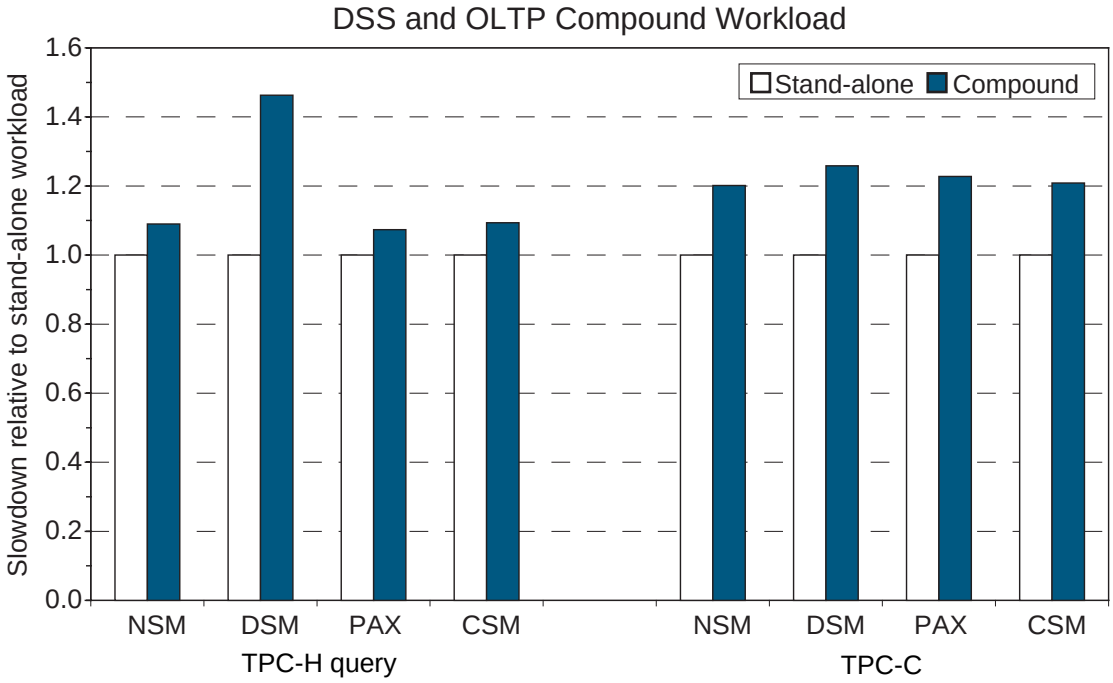


Figure 10: **Compound workload performance for different layouts.** This figure shows the slowdown off TPC-H query 1 runtime when run with TPC-C benchmark relative to the case when runs in isolation and the impact on TPC-C performance.

7.5 Compound OLTP/DSS workload

Benchmarks involving compound workloads are important in order to measure the impact on performance when different queries access the same logical volume concurrently. With *Clotho*, the performance degradation may be potentially worse than in other page layouts. The originally efficient semi-sequential access to disjoint *LBN*s (i.e., for OLTP queries) could be disrupted by competing I/Os from the other workload creating inefficient access. This problem does not occur for other layouts that map the entire page to consecutive *LBN*s that can be fetched in a single media access.

We simulate a compound workload with a single-user DSS (TPC-H) workload running concurrently with a multi-user OLTP workload (TPC-C) against our *Atropos* disk LVM and measure the differences in performance relative to the isolated workloads. The respective TPC workloads are configured as described earlier. In previous work [15], we demonstrated the effectiveness of track-aligned disk accesses on compound workloads; here, we compare all of the page layouts using these efficient I/Os to achieve comparable results for TPC-H.

As shown in Figure 10, undue performance degradation does not occur: *CSM* exhibits the same or lesser relative performance degradation than the other three layouts. The figure shows indicative performance results for TPC-H query 1 (others exhibit similar behavior) and for TPC-

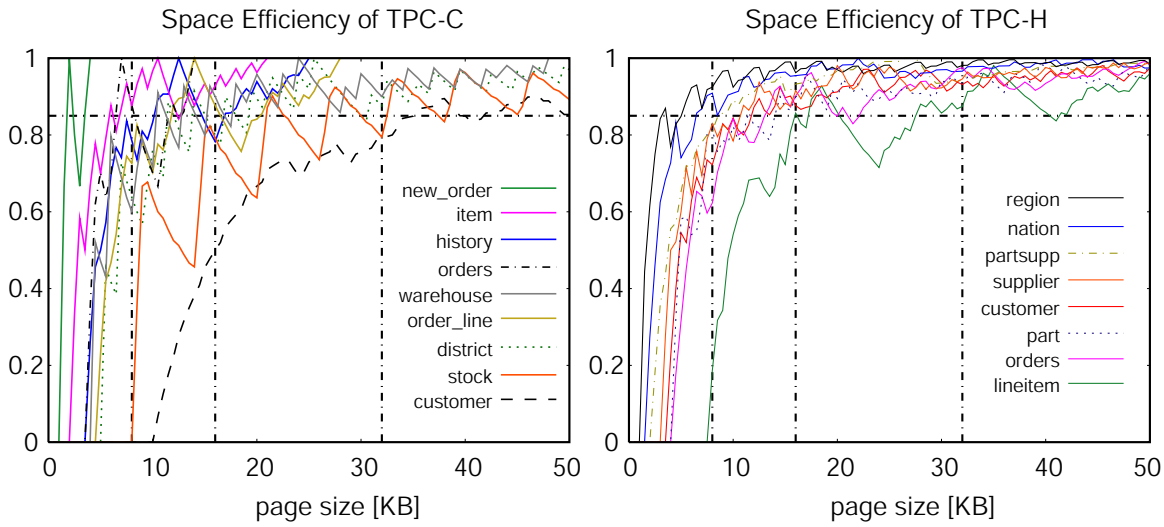


Figure 11: **Space efficiencies with *CSM* page layout.**

C, relative to the base case when OLTP and DSS queries run separately. The larger performance impact of compound workloads on DSS with *DSM* shows that small random I/O traffic aggravates the impact of seeks necessary to reconstruct a *DSM* page. Comparing *CSM* and *PAX*, the 1% lesser impact of *PAX* on TPC-H query is offset by 2% bigger impact on the TPC-C benchmark performance.

7.6 Space utilization

Since the *CSM* A-page partitions attributes into minipages whose minimal size is equal to the size of a single *LBN*, *CSM* is more susceptible to the negative effects of internal fragmentation than *NSM* or *PAX*. Consequently, a significant amount of space may potentially be wasted, resulting in diminished access efficiency. When using *PAX*, minipage boundaries can be aligned on word boundaries (i.e., 32 or 64 bits) to easily accommodate schemas with high variance in attribute sizes. *Clotho*, on the other hand, restricts minipage size to a multiple of the *LBN* size. In this case, *CSM* must use large A-page sizes to accommodate all the attributes without undue loss in access efficiency due to fragmented space.

To measure the space efficiency of the *CSM* A-page, we compare the space efficiency of *NSM* and *CSM* layouts for the TPC-C and TPC-H schemas. *NSM* exhibits the best possible efficiency among all four page layouts. Figure 11 shows the space efficiency of *CSM* relative to *NSM* for all tables of TPC-C and TPC-H as a function of total page size. Space efficiency is defined as the ratio between the maximum number of records that can be packed into a *CSM* page and the number of records that fit into an *NSM* page.

A 16 KB A-page suffices to achieve over 90% space utilization for all but the customer and stock tables of the TPC-C benchmark. A 32 KB A-page size achieves over 90% space efficiency for the remaining two tables. Both customer and stock tables include an attribute that is much larger than all other attributes. The customer table includes a 500 byte long `C_DATA` attribute containing "miscellaneous information", while the next largest attribute has a size of 20 bytes. The stock table includes a 50 byte `S_DATA` attribute, while the next largest attribute is 24 bytes. Both of these

attributes are rarely used in the TPC-C benchmark.

8 Conclusions

Clotho decouples in-memory page layout from in-storage data organization, enabling independent data layout design at each level of the storage hierarchy. Doing so allows *Clotho* to optimize I/O performance and memory utilization by only fetching the data desired for queries that access partial records, while mitigating the trade-off between *NSM* and *DSM*. Experiments with our *Clotho* implementation show substantial performance improvements across a spectrum of query types, for both a real disk array and future MEMS-based storage devices.

References

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

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: where does time go? *International Conference on Very Large Databases* (Edinburgh, UK, 7–10 September 1999), pages 266–277. Morgan Kaufmann Publishing, Inc., 1999.

[3] 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] L. Richard Carley, James A. Bain, Gary K. Fedder, David W. Greve, David F. Guillou, Michael S. C. Lu, Tamal Mukherjee, Suresh Santhanam, Leon Abelmann, and Seungook Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.

[6] Xin Chen and Xiaodong Zhang. Coordinated data prefetching by utilizing reference information at both proxy and web servers. *Performance Evaluation Review*, **29**(2):32–38. ACM, September 2001.

[7] George P. Copeland and Setrag 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.

[8] The DiskSim Simulation Environment (Version 3.0). http://www.pdl.cmu.edu/DiskSim/index.html.

[9] George G. Gorbatenko and David 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] Richard A. Hankins and Jignesh M. Patel. Data morphing: an adaptive, cache–conscious storage technique. *International Conference on Very Large Databases* (Berlin, Germany, 09–12 September 2003), pages 1–12. VLDB, 2003.

[11] 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.

[12] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data* (Chicago, IL), pages 109–116, 1–3 June 1988.

[13] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*, number 3rd edition. McGraw-Hill, 2003.

[14] Ravishankar Ramamurthy, David J. DeWitt, and Qi 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.

[15] Jiri Schindler, Anastassia Ailamaki, and Gregory 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.

[16] 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.

[17] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004). USENIX Association, 2004.

[18] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):1–12, 2000.

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

[20] Minglong Shao and Anastassia Ailamaki. *DBMbench: Microbenchmarking database systems in a small, yet real world*. Technical Report CMU–CS–03–161. Carnegie-Mellon University, Pittsburgh, PA, October 2003.

[21] P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The "Millipede" – more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, **44**(3):323–340, 2000.

[22] Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Tabular placement of relational data on MEMS-based storage devices. *International Conference on Very Large Databases* (Berlin, Germany, 09–12 September 2003), pages 680–693, 2003.