Enabling database-aware storage with OSD

Aravindan Raghuveer
University of Minnesota
aravind@cs.umn.edu

Steven W. Schlosser
Intel Research Pittsburgh
steven.w.schlosser@intel.com

Sami Iren
Seagate Research
sami.iren@seagate.com

Abstract

The ANSI Object-based Storage Device (OSD) standard is a major step toward enabling explicit *application-awareness* in storage systems behind a standard, fully-interoperable interface [3]. In this paper, we explore a particular flavor of application-awareness, that of database applications. We describe the design and implementation of a database-aware storage system that uses the OSD interface not only as a means to access data, but also to permit explicit communication between the application and the storage system. This communication is significant, as it enables our storage system to transparently optimize data placement and request scheduling. We demonstrate that OSD makes it practical to improve storage performance in these ways without exposing proprietary disk drive parameters to application code, and without labor-intensive, fragile parameter measurement.

1 Introduction

Storage system researchers, designers, and users have long bemoaned the limitations of standard block-based storage interfaces [5]. While providing good performance for most applications and concealing unnecessary complexity of the storage subsystem from applications, block-based interfaces keep parties on both sides of the interface ignorant of the other. There have been suggestions that interfaces should be more expressive [5, 6, 12, 16, 17, 19, 22, 28], that storage should intuit the details of the data they store and the applications that access it [24, 23], or that applications should run inside the storage subsystem itself [1, 8, 16]. In each case, impressive benefits were demonstrated "if only" the interface could be extended, or benefits were limited because the interface could not be changed.

These systems demonstrate that combining knowledge of storage system parameters and application details can improve overall performance for a variety of workloads. At a high level, the goal of combining storage and application parameters to improve performance leads to two alternative approaches: *storage-aware applications* and *application-aware storage*. While each could achieve the same ends, there are real concerns of practicality arising from the lack of a communication channel between storage and applications, and of depending on parameters which can often be hard to measure, fragile, and proprietary.

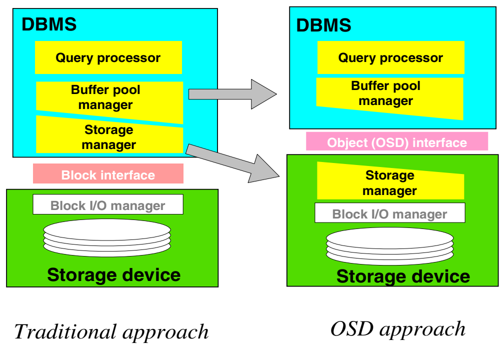

Figure 1: OSD moves the low-level storage management functions down to the storage device and provides an object interface.

The industry has taken a significant step forward in addressing practicality concerns by ratifying the ANSI Object-based Storage Device (OSD) standard [3]. The OSD interface includes a standard and extensible *shared attributes* mechanism which allows, among other things, explicit communication of parameters between applications and storage systems. While such a mechanism can enable both storage-aware applications and application-aware storage, we believe that only application-aware storage is practical because space management is handled by the object-based storage device.

Dealing with parameters is difficult and is usually storage- and application-dependent. Useful storage system parameters can be low-level, such as cache sizes and configuration, disk geometry, and prefetching and replacement policies. Some parameters, especially those regarding physical characteristics, are difficult to measure externally and are fragile even across different drives of the same model. On the other hand, some parameters are general statements about hardware (number of disks) or data (type of a particular file), and some are proprietary to particular hardware (the low-level geometry of a drive) or applications (the internal data structures of documents). Successful storage-aware applications or application-aware storage systems will need to strike the right balance between specificity and generality across a range of hardware and applications.

In this paper, we make the case for application-aware storage, and we investigate a particular flavor of application-aware storage that is enabled by OSD: that of database applications. We make the case that database-aware storage is more viable and can lead to better results than storage-aware databases, as have been proposed in the past [17, 20, 22]. In our system, database applications store each relation (table) in a single object and annotate that object with an OSD shared attribute specifying that relation's schema. The database-aware storage system uses the schema and its own knowledge of drive parameters to optimize low-level placement of the data independent of the database itself. In this way, the storage system is tasked with all low-level storage management tasks previously handled by the database application, but can do a better job by leveraging its own knowledge of low-level storage parameters.Unlike low-level storage parameters, schema information about a particular relation is not proprietary to the database vendor, but is a general statement providing semantic information about the data being stored.

The rest of the paper is organized as follows. Section 2 compares and contrasts storage-aware applications and application-aware storage, and describes related work. Section 3 describes how database-aware object-based storage devices can improve database performance. Section 4 describes our proposed database-aware OSD, including the proposed shared attributes. Section 5 describes how database software can be adapted to use OSD. Section 6 details our implementation. Section 7 presents an evaluation of the prototype's performance, and Section 8 concludes.

2 Storage-aware applications or application-aware storage?

Performance of storage-bound applications can be improved by combining detailed knowledge of both the application and the storage system. The question is whether the applications should be made aware of storage characteristics (i.e., storage-aware applications) or should storage systems be made more aware of application characteristics (i.e., application-aware storage)? As with so many things, the answer depends on which characteristics are to be used, how readily available those characteristics are, and where they are most easily and effectively put to use.

Fortunately, a major barrier to enabling both storage-aware applications and application-aware storage, the lack of a standard communication mechanism between applications and storage, has been broken down by the adoption of the industry-standard OSD interface. The OSD interface opens the door for new research in practical, interoperable systems that leverage application and storage characteristics to improve performance.

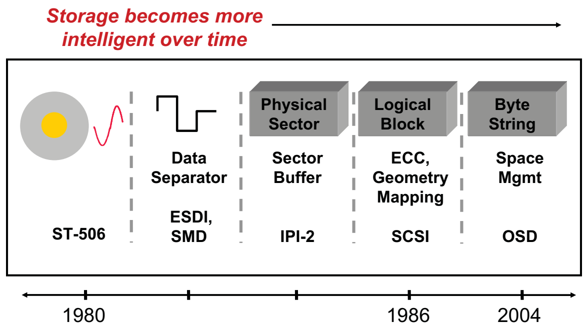

Figure 2: Progression of storage interfaces. Each new interface moves more intelligence to storage.

2.1 Object-based storage

Object-based storage has its roots in the network-attached storage research of the 1990s [7], and now it is an accepted industry standard [3]. It is the next logical step in the progression of storage interfaces. As shown in Figure 2, storage interfaces have progressed steadily over the last several decades, and each change has moved more intelligence from hosts to storage devices (disks). OSD follows this tradition.

The basic unit of storage in OSD is an object. An object is a logical unit (i.e., sequence of bytes) of storage with well-known, file-like access methods (including `READ`, `WRITE`, `CREATE`, and `REMOVE`), object attributes describing the characteristics of the object, and security policies that authorize access [13]. An object is a variable-size entity and can store any type of data including text, images, audio/video, and database relations. The storage application decides what should go into an object. An object grows or shrinks dynamically as data is written or deleted.

The difference between OSDs and block-based devices is the interface, not the physical media the objects are stored on. Hence, magnetic or optical, read-only or writable, random access or sequential access, all storage devices can store objects and can be considered an OSD [13]. Early examples of OSD are single disk drives (e.g., Seagate OSD disks), smart disks (e.g., Panasas storage blades), and disk array/server subsystems (e.g., LLNL units with Lustre). Tape drives and optical media may also be used as OSDs in the future.

One significant advantage of OSD is that the space management is delegated to storage devices. Thus, the storage system has complete knowledge of how each disk block is used and is related to other blocks. This is a dramatic change from today's block-based storage devices, which cannot even detect which blocks are free and which are used.

The OSD standard also provides for two types of attributes that can be associated with data objects, as illustrated in

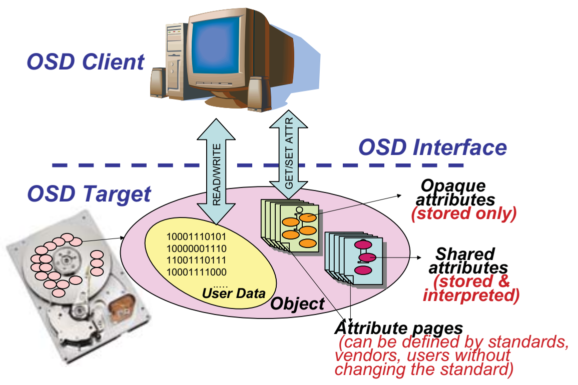
OSD Client
OSD Interface
READ/WRITE
GET/SET ATTR
OSD Target
Opaque attributes (stored only)
10001110101
100000001110
11001110111
10001111000
User Data
Object
Shared attributes (stored & interpreted)
Attribute pages (can be defined by standards, vendors, users without changing the standard)

Figure 3: The basic unit of storage in OSD is an object: a logical unit of storage with file-like access methods (i.e.,`READ` and `WRITE`). Applications can attach two types of attributes to objects: *opaque* and *shared*. Opaque attributes are not interpreted by the OSD, just stored on behalf of applications. Shared attributes are both stored and interpreted by the OSD, and can be used as a communication mechanism between the application and the storage system.

Figure 3. The first type, *opaque attributes*, are not interpreted by the OSD itself, but are simply maintained on behalf of applications for their own use. The second type, *shared attributes*, can be interpreted by both the application and the OSD and provide a standard, yet extensible, communication channel. For example, shared attributes can relay information such as an application's requirements for quality of service, reliability, and security. While attributes are generally described as being attached by applications to objects to be read by the storage system, attributes could also be attached to objects by the storage system to be read by the application.

Related attributes are grouped together in attribute pages, the format of which is defined in the standard for interoperability. Therefore, any OSD application can access these pages even if these pages are not known to the application ahead of time. There is a discovery mechanism (called object directory pages) that enables applications to find out every attribute page defined on a particular OSD, either by standards, device manufacturers, or other applications.

The OSD standard defines a small set of attributes that are deemed necessary for all applications. A significant portion of the attribute name space is intentionally left undefined by the standards body, to be used by (1) future standards, (2) device manufacturers, (3) vendors, and (4) applications (created on the fly). This division of the attribute name space makes OSD *extensible* with shared attributes defined by device manufacturers while still complying with the specification. If there is demand, these shared attributes can be included in future versions of the specification.

Since the format of the attribute pages is defined in the standard, there is no interoperability problem between applications and devices that do or do not support these shared attributes. Those applications that are optimized to take advantage of the shared attributes can do so by accessing and interpreting these attributes, while standard applications can use the OSD as if these shared attributes do not exist. Since attributes are tied to objects, as objects move around (e.g., migrate from one device to another) attributes move with them. Hence, an object with a shared attribute can live on devices that do or do not support that particular shared attribute with no interoperability problems.

The benefits of a higher level of abstraction for storage has been described elsewhere, and much of that value comes from the use of objects rather than blocks as the primary unit of access [7, 13]. OSD's shared attribute mechanism has been explored less in the literature, however. Shared attributes specifying quality of service levels have been proposed and evaluated [11]. We believe that the shared attribute mechanism can be used much more widely and can enable both storage-aware applications and application-aware storage.

2.2 Storage-aware applications

Many have argued that applications should know more about the details of storage and can effectively use this information to improve performance. These efforts demonstrate the potential for storage-aware applications, but they are often deemed impractical since there is no standard means for an application to receive parameters from storage. OSD shared attributes could solve this *parameter passing problem* by allowing the storage system to attach shared attributes to individual objects, collections of objects, or to the entire drive. Applications could then query that attribute to read the required parameters and make their own optimizations on a global or per-object basis.

While OSD could provide a means for solving the parameter passing problem, there remain several shortcomings of using OSD attributes to enable storage-aware applications. First, storage vendors will likely remain reluctant to expose proprietary parameters. There is little reason to believe that vendors will become more willing to expose more parameters just because there is a standard mechanism to do so. Second, storage parameters can change more often than application parameters, often asynchronously due to data migration or disk failure, meaning that storage-aware applications will have to be built to tolerate parameter changes at any time. Third, and most significant, OSD provides a virtualized view of data via the object abstraction, taking full control over data placement and free space management. Even if an application has the parameters that describe low-level details of the storage system (e.g., disk characteristics that can be used to improve data placement), the application cannot use them since it has no control over how data is placed on disk. As a result, a large class of storage parameters that could be exposed using OSD shared attributes are useless to storage-aware applications.

2.3 Application-aware storage

Application-aware storage devices can use the shared attribute mechanism of OSD to receive parameters of data and applications, and can perform optimizations behind the storage interface. By understanding the data that they store and the applications that are using them, application-aware OSDs can proactively optimize storage and access of data in application-specific ways. OSD can enable a variety of application-aware object-based storage devices, each tailored to one or more applications [9]. End-users will choose OSDs that are optimized for their specific application.

We believe that application-awareness avoids several of the shortcomings of storage-aware applications described above and is, therefore, more practical.

First, many application parameters are more static and generic than physical hardware parameters. Take the type of a file as an example. A file's type can be expected to remain constant over its lifetime, and the various types themselves tend not to change very quickly, Image and document formats like JPEG and Microsoft Word files remain fixed for reasonably long periods of time, much longer than any particular disk layout. While the internal structure of some of these file types is proprietary, many are standardized to promote interoperability.

Second, application-awareness using OSD maintains appropriate abstraction boundaries between applications and storage. At a high level, application-awareness represents the right division of labor between applications and storage systems. More practically, this means that low-level storage optimizations are handled by the storage system, which is armed with the requisite application and storage parameters. Optimizations are made transparently – the application is unaware of the optimization except for, hopefully, improved performance. Applications are not required to handle changes in the storage hardware, nor do they need to re-optimize when moving data from one storage system to another. If the original application parameters follow the data when it is moved between systems, then the new storage system can re-optimize.

Lastly, storage interfaces have been changing over time, becoming increasingly functional and handling higher-level storage tasks. Figure 2 illustrates how early storage interfaces provided only very low-level functionality, requiring applications to directly control formatting, error correction, and even aspects of magnetic recording. Over time, storage systems have moved higher-level functions from the host system behind the storage interface, becoming more application-aware over time. OSD is the next natural step in that progression, moving the next lowest levels of storage functionality into the storage system. This progression is significant in this discussion, as there is momentum in the storage industry to move *more* and *higher-level* functionality into the storage system, and no momentum toward exposing lower-level parameters to applications. Application-aware storage embraces rather than fights this momentum, providing an interesting new avenue of research in improving storage performance for specific applications.

Not all OSDs will be application-aware, and not all applications will have corresponding application-aware OSDs. As well, interoperability between standard and application-aware OSDs is critical to their success. An application-aware OSD should provide baseline performance for all applications that is equivalent to normal (i.e., non-application-aware) OSDs. In other words, an application-aware OSD should not impose a performance penalty to applications for which it is not optimized. Also, a non-application-aware OSD should function correctly for all applications, but will not provide the performance benefits of application-awareness. For example, a database storage manager that has been written for OSD should function correctly on both database-aware and database-unaware OSDs. Of course, that database should perform better when using a database-aware OSD.

2.4 Related work

Perhaps the most direct form of application-aware storage is when a part of the application executes in the storage system itself. Active Disks allow application code to run on storage nodes directly with demonstrated benefits for multimedia and databases [1, 8, 16]. However, practical Active Disks will require more work beyond the standard OSD interface, both on the host side and the target side. Although OSD is the right interface for active disks, it needs to be extended to enable application code to be downloaded to storage nodes. Local runtime systems on the storage nodes must be defined that provide good performance, portability, and isolation.

Semantically-smart disk systems are another form of application-aware storage that take a different approach [23, 24]. Their goal is to imbue the storage system with application-awareness while maintaining the standard block interface. The storage system monitors the on-disk data and request stream to intuit application-specific knowledge of the data and the applications. Semantically-smart disks have been demonstrated to improve performance, reliability, and security, both for filesystems and for databases. While our goals are similar, our technique avoids the guesswork of interpreting on-disk datastructures and sequences of requests by using explicit communication enabled by OSD shared attributes.

OSD provides a practical middle ground that is both effective for enabling application-awareness and is general enough to allow the right level of interoperability among vendors and applications. OSD is similar to active storage in that the application-awareness it enables is explicit rather than implied. That is, the application uses shared attributes to communicate its characteristics directly with the storage system, as opposed to requiring the storage system to try and interpret the data it stores on its own. Being an industry-standard interface that is already designed to be extensible, OSD can support a wide variety of applications and interoperable devices.

The Fates project developed a storage-aware database system that used detailed characteristics disk drives to preserve physical locality of multidimensional datasets [17, 20, 22]. As with many storage-aware applications, the Fates work depends on the availability of these parameters to the database. As we describe in the next section, application-aware storage (in this case, database-aware storage) can address this shortcoming and make these techniques much more practical.

3 Motivation - OSD for databases

In this section, we make the case that databases are a good candidate for application-aware storage. Databases are often very dependent on storage performance, and so improvements in storage subsystems can lead to significant benefits. Studies have shown that database storage performance can be improved by leveraging detailed knowledge of storage subsystem characteristics which are generally unavailable at the application level. Furthermore, the majority of commercial databases share the relational storage model, meaning that a storage system that understands relational databases has the potential to benefit many applications.

3.1 Using storage parameters in databases

Recent research showing the benefit of leveraging storage system parameters is promising, but is generally impractical because those parameters are unavailable at the application level. Database-aware storage turns this problem around by passing the parameters of the database to the storage system, enabling storage to use internal knowledge of storage parameters to improve performance transparently to the application.

3.1.1 Classical database storage management

A database management system (DBMS) typically accesses storage via its own storage manager. The storage manager handles data placement, file access, and scheduling of disk requests. A DBMS usually implements its own in-memory buffer pool management as well. The storage manager reads and writes fixed-sized data pages (typically 8-64 KB each) to and from the disk into the buffer pool.

Most current database systems use the N-ary storage model (NSM) as their low-level data layout, which organizes the table into fixed-size pages (e.g., 8KB) each containing a short range of full records [14]. Pages are stored sequentially on disk. NSM is well-suited to workloads that access full records, such as online transaction processing (OLTP) workloads, but is not well-suited to workloads that access partial records, such as decision support system (DSS) workloads. Full records are always fetched into memory regardless of whether the query actually touches the data, wasting memory capacity and, more importantly, disk bandwidth.

DSS workloads are often better-served by the decomposition storage model (DSM), which organizes a table in column-major order by storing individual fields sequentially on the disk [4]. Because of its column-major organization, DSM gives good performance when accessing just one or a few fields from a relation, but suffers when reconstructing full records, leading to poor OLTP performance. A more recent system, C-Store [26], is a column store that is built for read-optimized DSS workloads.

The shortcomings of these storage models stem from the fact that the storage interface is inherently linear, requiring serialization of the relation along one axis or the other. The serialization problem can be mitigated by maintaining two copies of the relation, one using NSM and one using DSM, as suggested by Ramamurthy et al. [15]. However, this technique doubles the required storage space and must propagate updates to both copies of the relation.

3.1.2 Geometry-aware storage management

A more optimal storage model would allow the DBMS to fetch only the data required by a given query as efficiently as possible, given the characteristics of the underlying storage subsystem. Put differently, a successful storage model should deliver performance comparable to NSM for full-record access, equal to DSM for single-attribute access, and provide a linear tradeoff for partial-record access.

DBMS storage managers typically have little or no information about the underlying storage subsystem, which can lead to inefficient storage performance. Several of these shortcomings have been explored and solved in the

Fates project [17, 20, 22], which investigated three aspects of storage performance inefficiency in databases. Lachesis [17] showed the importance of aligning disk accesses to track boundaries. Atropos [20] demonstrated the use of semi-sequential disk access to enable two dimensions of efficient access on disk. Clotho [22] showed the benefit of building query-specific, rather than fixed-content, pages in the buffer pool, and how advanced data placement techniques such as those enabled by Atropos can make fetching query-specific pages from disk efficient. Clotho introduced a new storage model, CSM, which equaled or exceeded the performance of NSM and DSM for both full-record and single-attribute access, respectively.

As these systems are storage-aware applications, each requires the storage manager or logical volume manager to have detailed knowledge of the characteristics of the underlying disk drives. Lachesis and Atropos require knowledge of each disk's track boundaries. Atropos requires even more detail, including a full logical-to-physical map of the disk blocks, the settle time of the disk head after a seek, and tight control over request scheduling. Unfortunately, these parameters are not likely to be exposed by storage system vendors, which raises questions about the practicality of these techniques.

3.2 Turning the problem around with application-awareness

In order to enable the geometry-aware storage optimizations described above, two sets of parameters must be brought together: the parameters of the disks in the storage system and the schema of the database relations being stored. Detailed disk parameters are notoriously difficult to come by, and are generally viewed as proprietary information by storage vendors and are not exposed. Disk parameters can be extracted empirically using specialized tools [18, 27, 29]. Unfortunately, these techniques are generally time-consuming and fragile, as disks continually change from year to year, vendor to vendor, even disk to disk. Without continual effort, parameter extraction tools are bound to lag behind disk development.

On the other hand, the majority of databases use the relational model: tables of strongly-typed records each with a fixed schema. Database software from different vendors may implement the relational model somewhat differently (e.g., vendors use their own page formats when storing relations to disk). However, since most of these databases share the relational model, we can expect database-aware storage to benefit the majority of databases. As well, a relation's schema is specified by the database user, generally does not change over time, and is not proprietary to any particular database system.

The approach that we advocate in this paper is to enable the database software to expose the relevant characteristics of the relations to a database-aware storage subsystem, enabling the optimizations to be handled at the storage level, where the necessary storage parameters are known. Exposing a relation's schema to the storage system is much more practical than exposing the storage subsystem's parameters to the database software, and can achieve the desired goal of enabling parameter-driven database storage management.

Using object-based storage, a DBMS can inform the storage subsystem of the schema of a relation, thereby passing responsibility for low-level data layout to the storage device, where the requisite disk parameters reside. The DBMS no longer needs information about the storage subsystem, but still can take advantage of geometry-aware data layouts such as CSM [20, 22]. The database-aware object-based storage prototype that we describe in this paper demonstrates this approach, implementing the CSM layout behind the object interface. In the following sections we describe how the shared attribute and session mechanisms of the OSD standard can enable an OSD to implement optimizations that were impossible or impractical with previous storage interfaces.

By moving the storage component of the DBMS to storage devices, object-based storage also removes the biggest obstacle to data sharing. Database management systems (and file systems) have their own ways of placing the data on the disk and maintaining block-level metadata. Therefore, when different hosts access the same data, they need to have *a priori* knowledge of both the metadata and on-disk layout. For a DBMS, this usually means a relation created with one vendor's software cannot be shared with another vendor's software. With objects, since metadata is offloaded to the storage device, the dependency between metadata and the storage system/application is removed, enabling portability and data sharing between different storage applications (i.e., *cross platform data sharing*). This also improves the scalability of clusters since hosts no longer need to coordinate all metadata updates.

For DBMS, the above advantages can further be complemented with on-disk processing either for ongoing maintenance of stored relations or more advanced concepts like execution of portions of queries. Using background tasks transparent to the user, storage devices can handle many ongoing maintenance tasks such as reorganization, sorting, indexing, and free space management. For example, relations that are sorted on a primary key must be kept sorted on disk as records are inserted and deleted. Indices and materialized views must also be maintained as the data is updated. As the data and workload changes, the storage organization can be optimized on disk. These maintenance tasks can be handled synchronously while the workload is running, but at a cost in performance. A database-aware storage system could more effectively exploit idle time and excess capacity to provide many of these ongoing maintenance functions asynchronously or lazily without intervention by the DBMS by employing techniques such as free-block scheduling [12].

With an active disk approach, database-aware storage devices could take part in the execution of the queries. With a minimum amount of data movement and better disk utilization, on-disk query processing may provide overall better performance in many cases. Obviously, there are several issues that require further study. For example, deciding how much of the query can/should be executed at the storage level (e.g., processing power available at the storage) and how the query structures can/should be passed to storage devices.

4 Using the OSD interface

In this section, we examine how to use the OSD interface to enable database-awareness. We seek to answer three questions: (1) How is a relation mapped to data objects? (2) How is semantic information provided to the storage system? (3) How is data accessed most efficiently?

We start with the standard OSD interface [3] and find that it is well-suited for mapping relations, incorporating semantic information from applications, and providing database-awareness. However, one shortcoming remains: data objects are addressed through the interface as linear arrays of bytes, which contrasts with the two-dimensional structure of relational databases. We propose a solution using sessions, a concept that was considered as part of the original OSD specification [2], but was dropped in the last phases of the standardization process for the sake of simplicity. Sessions are currently being investigated by the OSD working group [25] and academia [11] as part of the QoS extension to OSD. In this paper, we provide another compelling use for OSD sessions.

4.1 Mapping relations to objects

We consider several options for mapping database relations to objects: (1) one object per record, (2) one object per field, (3) one object per data page (e.g., 8KB), (4) one object per relation.

Most relations contain too many records (millions or billions) to store each record in its own object. Using a per-record mapping, scans that address large portions of the table would generate too many requests to be practical. Furthermore, the OSD security model requires a credential for every object accessed and there can be only one credential per CDB (command). Therefore, only one object can be accessed with a single command. This type of mapping would significantly reduce the efficiency of the system as each record would require a separate command to be sent to the storage device. Therefore, a row-major decomposition seems infeasible.

Table 1: Shared attribute specifying a relation's schema.

Attribute page	Number	Value
Relation schema (8003h)	1h	Field 1 length
	2h	Field 2 length
	3h	Field 3 length
	...	...
	Nh	Field N length

Decomposing the relation into columns and creating an object per field is a natural approach [26], but reconstructing a full record is often expensive, as in standard DSM layouts [4]. In fact, decomposing the two-dimensional relation in either order (row- or column-major) presupposes that the stored relation will be accessed in that order. Accesses in the chosen order are, therefore, efficient and accesses in the other order are inefficient. Making that decision *a priori* is not always possible, so the storage system should not require it.

Storing the data in fixed-sized data pages (e.g., 8KB NSM or DSM pages) with an object for each page has the advantage that it naturally fits the storage model of current databases. However, geometry-aware layouts do not use pages with fixed contents, since the DBMS must be free to access arbitrary ranges of records and fields within those records.

Therefore, our implementation stores the entire relation in a single object. This organization allows the storage system to optimize placement of the relation under the hood in any way it sees fit, while ensuring that the database system can address that data in a consistent, interoperable way. This interoperability is really the key – if application-aware storage is to be successful, data representations need to be abstract and simple, and impose no restraints on the application. As well, it provides the right division of labor between the application and the storage system. In this case, the database is able to logically read and write data regardless of its physical organization on disk.

4.2 Passing semantic information

When an object is created to store a relation, the schema is expressed as a *shared attribute* that is assigned to that object. We defined a shared attribute page that specifies the schema of a relation, identifying the length of each field in bytes, as shown in Table 1. Note that our attribute specifies the length or an attribute rather than its particular type (e.g., int, float, char). Specifying the length is sufficient

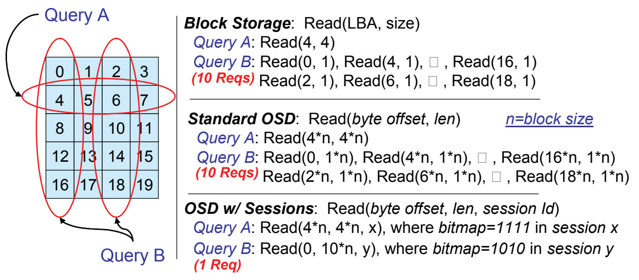
Query A
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Query B
Block Storage: Read(LBA, size)
Query A: Read(4, 4)
Query B: Read(0, 1), Read(4, 1), □ , Read(16, 1)
(10 Reqs) Read(2, 1), Read(6, 1), □ , Read(18, 1)
Standard OSD: Read(byte offset, len)
n=block size
Query A: Read(4*n, 4*n)
Query B: Read(0, 1*n), Read(4*n, 1*n), □ , Read(16*n, 1*n)
(10 Reqs) Read(2*n, 1*n), Read(6*n, 1*n), □ , Read(18*n, 1*n)
OSD w/ Sessions: Read(byte offset, len, session Id)
Query A: Read(4*n, 4*n, x), where bitmap=1111 in session x
Query B: Read(0, 10*n, y), where bitmap=1010 in session y
(1 Req)

Figure 4: Linearization of the relation causes problems when generating requests for projections. Compare the number of requests required for Query A versus Query B. For standard block-based storage, OSD, and OSD with sessions, servicing Query A requires only a single request. However, Query B requires 10 strided requests for both block storage and standard OSD. OSD with sessions satisfies Query B with a single request, since it can be made in terms of a schema bitmap.

for the optimized two-dimensional placement of data that we implement, although future optimizations may benefit from specifying explicit types. A special value of `FFFF FFFFh` is used to indicate a variable length field, which are handled separately from the fixed-length fields. The OSD standard allows up to 2^{32} attributes (fields) to be defined on a page.

4.3 Accessing relations

Once a relation object is created (via a `CREATE` command) and the schema defined (via a `SET ATTRIBUTES` command), the DBMS needs to be able to address ranges of records. Objects in the current OSD specification are addressed as a linear array of bytes, despite the fact that the underlying storage may be optimized for multidimensional access. To address individual records, the DBMS must calculate the byte offsets of those records within the relation. Individual fields within records can also be addressed at a byte granularity. Once byte offsets are calculated, `READ` and `WRITE` commands access the data in a relation object.

Unfortunately, specifying arbitrary ranges of fields and records from the relation using the byte interface is problematic, as illustrated in Figure 4. The linear byte address space imposes a serialization on the relation, even though multidimensional access is efficient due to geometry-aware disk layouts. The illustration shows a simple example of linearizing a table in row-major order. For example, accessing the second row of the table (Query A) requires only a single `READ` request to block 4 of size 4. However, accessing two entire columns of the table (Query B) requires ten requests, each of size 1. Such an interface is unwieldy and expensive, and is only a result of the linear byte abstraction. A preferable approach would be to issue a single request for either case.

The main difficulty with the current OSD specification comes from the fact that objects, and hence their access methods, are defined as linear entities. This is acceptable with current file systems where files are defined as sequence of bytes. However, database systems use two dimensional tables which require a translation from a two dimensional record/attribute structure to a one dimensional byte offset. This not only requires extra effort and bookkeeping at the application level, but also is inefficient when a single attribute of all the records needs to be accessed (i.e., partial record access), as illustrated in Figure 4. The goal should be for the DBMS to generate its requests in terms of the schema of the relation and the desired contents of the resulting pages in memory. Toward this end, we propose a solution that utilizes the session mechanism proposed for OSD.

An OSD session is a set of I/O requests that are to be honored by the OSD device in the same way. That is, applications can use the `SET ATTRIBUTE` command to set up session attributes. In return, OSD devices process the I/O requests per those specifications. A session is started with an `OPEN` or `CREATE` command upon which OSD returns a unique Session Id. Clients then supply this Session Id in each `READ`, `WRITE`, or `APPEND` command. The session is ended with a `CLOSE` command [2].

Sessions were part of the OSD proposal throughout much of the standardization process (up until version 0.8 of the specification). However, they were dropped in the last phase to keep the first version of the OSD specification simple. They are now being discussed in the OSD working group as part of the upcoming OSD QoS proposal, and will likely be part of the future versions of the OSD standard.

We keep the original definition of OSD sessions and introduce a Field Bitmap attribute that describes which fields of a record should be accessed with each I/O request. The Field Bitmap attribute is set for each session that will be used for the database object with the `SET ATTRIBUTE` command. After that, regular `READ` and `WRITE` commands are used to access the database object where a different Session Id is used to indicate a different field bitmap.

Figure 4 shows the `READ` commands issued to an OSD with sessions support to execute Query A and Query B. For both queries, the DBMS specifies the starting byte address, required number of bytes to be transferred, and the Session Id to be used for the transfers. Each of the two sessions (labeled x and y) represents a different bitmap specifying the desired fields. In contrast to the other two approaches, OSD with sessions can deliver data for either query with a single request that is made in terms of the relation's schema.

Figure 5 describes the sequence of events to satisfy Queries A and B illustrated in Figure 4. Two sessions are created to represent two types of access patterns required for queries *A* and *B*. The respective Session Ids are used with each `READ` command to execute either query. Both sessions

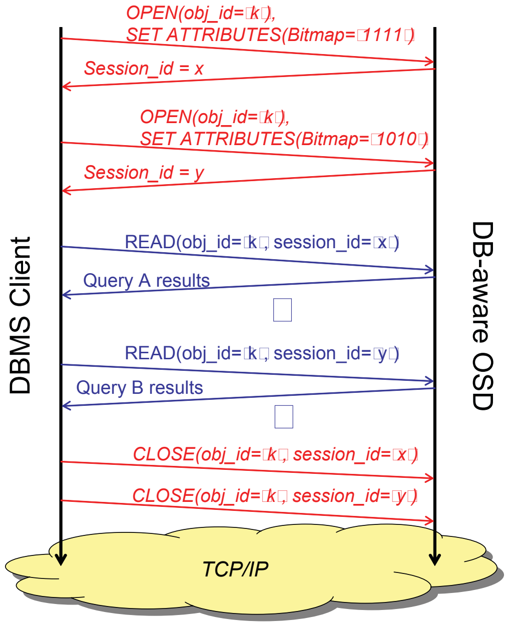
OPEN(obj_id=□k□),
SET ATTRIBUTES(Bitmap=□1111□)
Session_id = x
OPEN(obj_id=□k□),
SET ATTRIBUTES(Bitmap=□1010□)
Session_id = y
READ(obj_id=□k□, session_id=□x□)
Query A results
□
READ(obj_id=□k□, session_id=□y□)
Query B results
□
CLOSE(obj_id=□k□, session_id=□x□)
CLOSE(obj_id=□k□, session_id=□y□)
TCP/IP
DBMS Client
DB-aware OSD

Figure 5: In order to service Queries A and B illustrated in Figure 4, a DBMS client opens two sessions and specifies which fields will be accessed for each session by setting a Field Bitmap attribute. Then, it accesses the database object with regular `READ` and `WRITE` commands each time specifying a Session Id to indicate what fields are to be accessed. When no Session Id is specified, a regular "byte order" access is realized.

are closed when they are no longer needed. Note that sessions are created only once at the beginning and used many times, making their overhead negligible over the length of a query.

5 Adapting database software

As illustrated in Figure 1, database-aware object-based storage devices can take on many of the low-level storage management tasks required of today's database systems. In this section we describe the changes to database software that we envision due to database-aware storage. Fortunately, the DBMS buffer pool provides a strong layer of abstraction between the upper level DBMS software (i.e., query optimizer, planner, etc.) and the lower-level storage manager. No changes are required above the buffer pool manager.

5.1 Database storage manager

Database-aware storage devices have the potential to handle much of the storage management functionality present in today's databases. This is analogous to the object store replacing much of the lowest levels of a filesystem, including inodes, block pointers, and free space management, and encapsulating them below a standard interface. In a sense, the storage manager would disappear as part of the DBMS.

A database-aware OSD can implement various low-level data placement schemes, schedulers, read-ahead and write-back policies, and background maintenance functions. Most importantly, each of these functions can be tailored to and optimized for the underlying storage devices and topologies, transparently to the upper-level database software.

5.2 Buffer pool manager

A traditional DBMS fetches data from relations into an in-memory buffer pool. Relations are statically broken into fixed-sized pages (typically 8-64 KB) which are stored sequentially on disk. When data is to be fetched into memory, the DBMS requests the required page be fetched into the buffer pool by the storage manager. The buffer pool manager's task is to manage the memory pool, fetching pages from disk, evicting pages when memory is needed, and writing dirty pages back to disk when necessary.

However, with the migration of the lower-level storage manager into the database-aware OSD, the buffer pool manager is now the lowest level of the database software, sitting just above the OSD interface. Since the storage system handles data placement, there are no longer statically-defined pages in the system. Access to the relation stored on disk is all done in terms of the desired records and attributes that are needed to satisfy a particular query. The storage system ensures that those records and attributes are fetched efficiently into memory and the buffer pool manager is left to maintain the data once it is fetched.

Query-specific buffer pool management can provide significant benefits to decision support workloads without affecting the performance of other workloads. In particular, sequential scans that fetch a subset of attributes will benefit from increased disk bandwidth and better hit rates in the buffer pool because only those attributes which are required for the query are fetched. Using a static page layout like NSM will always fetch all attributes into the buffer pool even when just one attribute is needed by the query, wasting disk bandwidth and buffer pool capacity. When full records are required by the query, they are fetched either using sequential track-aligned access (for accessing many records) or efficient semi-sequential access (for accessing single records). The policies and tradeoffs in query-specific buffer pool management are beyond the scope of this paper, but are explored in detail by Clotho [22].

Efficient construction of query-specific pages is enabled with the help of geometry-aware data placement, which was first demonstrated by the Atropos logical volume manager [20]. As we show in this paper, database-aware object-based storage can enable the geometry-aware data placement required for efficient query-specific buffer pool management described above. Moving data placement func-

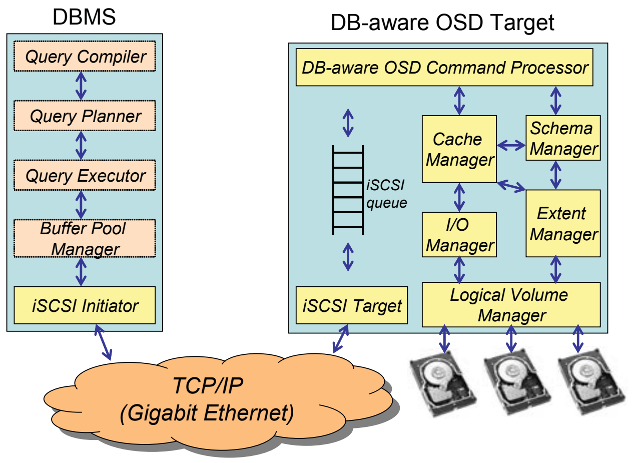

Figure 6: Architecture of our prototype implementation.

tionality into the storage system can provide a significant improvement over implementing it at the database software level, since the required disk parameters are readily available within the storage system.

Many important database features such as indices, locking, and logging are orthogonal to on-disk data placement and will work as usual in a database system using a database-aware storage device. Indices can be built for relations without regard to their on-disk layout and can be stored in their own objects. Locking can occur at the appropriate levels, starting with the entire relation all the way down to individual records. Since database-aware storage eliminates the standard notion of a fixed-size, static, on-disk page, page-level locking would have to be handled in a separate datastructure.

Database-aware storage devices have the flexibility to handle variable-sized attributes in a variety of ways, possibly tailored to a specific relation or workload. Variable-sized attributes can be stored entirely separately from the rest of the relation or can be cast as very large fixed-sized attributes with some provision for overflow. In our design, variable-sized attributes are identified to the database-aware storage device in the schema shared attribute. Most importantly, allocation and placement of variable-sized attributes can be handled by the storage system in a way that is optimal and transparent to the database software.

6 Implementation

In this section, we describe the design and implementation of our database-aware object-based storage device prototype. Figure 6 shows the architecture of the database system built over the object-based storage device. This system consists of three distinct components: The database management system (DBMS), an interconnect infrastructure, and the object-based storage device.

6.1 Database-aware object-based target

Figure 6 shows the components that comprise the database-aware object-based target. Our target currently implements geometry-aware data placement for relational tables and translation of front-end OSD `READ` and `WRITE` requests to efficient back-end disk requests. We explain below each component of the target in further detail.

iSCSI Target: The iSCSI target module is responsible for handling iSCSI specific functions (e.g., discovery, encapsulation of responses to initiators). The basis of our target is an open source OSD/iSCSI toolkit [10]. The CDB encapsulation and decapsulation routines were extended to build / unpack a CDB belonging to a database session.

OSD Command Processor: The OSD command processor fetches OSD commands from the iSCSI queue and hands control over to the appropriate module that implements the command. We have extended the command processor module to handle the `READ` and `WRITE` commands within database sessions.

Schema Manager: This module exposes the schema information of relations to other modules.

Extent Manager: While exposing an object interface, the target internally uses a block-based SCSI device to physically store and retrieve data. The extent manager module is responsible for managing the disk space on the block-based storage device.

The extent manager uses detailed knowledge of both the data and the device to decide the size and position of the extents allocated for an object. The size of extents allocated to a table object is determined by its schema (number and size of attributes) as provided by the schema manager. Extents are sized to align to record boundaries, ensuring that while accessing multiple attributes within a single record, seeks to different extents are avoided. Extents are always positioned on track-aligned boundaries, which eliminates the track-switch overhead and rotational latency for track-sized I/Os [19].

Cache Manager: In the current implementation, we do not implement caching and hence all `READ` and `WRITE` commands are relayed down to the I/O manager.

I/O Manager : The I/O Manager module converts a front end `READ`/ `WRITE` command into actual disk block requests based on the layout of the relation. Our I/O manager supports the NSM, DSM and the CSM layouts. For the CSM layout, the I/O manager chooses between sequential or semi-sequential access based on the number of attributes and records requested. The I/O manager uses efficient scatter-gather I/O to transfer data from the storage device to the network buffers and vice-versa.

The Logical Volume Manager: The basic functionality of a logical volume manager (LVM) is to abstract one or more disks and present them a single logical volume. We use the Atropos LVM [20] to abstract one or more disks and present them as a single volume. The Atropos LVM also finds (through disk characterization) and exposes track boundaries and semi-sequential blocks. The extent manager and the I/O manager use these two parameters to control data placement and to plan I/O.

The database-aware OSD target was implemented using 3000 lines of C++ code. The extent manager and the schema manager required 550 and 250 lines of code, respectively. We added around 500 lines of code to the Intel iSCSI/OSD toolkit to handle database specific `READ` and `WRITE` commands. The bulk of the target code went into the I/O managers for different layout schemes: NSM 450 lines, DSM 650 lines, CSM 800 lines. The Atropos LVM comprises of 2600 lines C++ and C code. The unmodified implementation in the OSD/iSCSI toolkit consists of 22,600 lines of C code. Hence, it is seen that adding database specific functionality does not increase the complexity of the system significantly.

6.2 Database software

In Section 5, we described issues of how DBMS software may change to utilize OSD. Since in this paper we are interested only in evaluating storage performance, we use two simple user-level programs on the initiator that generate I/O requests representative of typical database workloads. The first program, called the *buffer pool emulator*, emulates the I/O traffic generated by a buffer pool manager. The second program is a simple *query engine* to perform TPC-H benchmark queries. The query engine internally uses the buffer pool emulator to generate table scan requests to the target. The query engine implements two-phase merge sort and hash join operators and uses regular objects on the OSD target to store intermediate data. We use a separate user-level program to bulk-load the tables into the database. The buffer pool emulator and the query engine comprise of 1700 lines of C code.

7 Evaluation

In this section we evaluate the performance of the database-aware target. Our goal is to show that our OSD based implementation can provide the benefits of prior storage-aware approaches [17, 20, 22] while providing a more practical route to solving the parameter problem.

In the first set of experiments, we evaluate the performance of the database-aware target for database workloads. The buffer pool emulator generates typical workloads of a buffer pool manager through table scan and random read microbenchmarks [21]. Next, we use the query engine in conjunction with the buffer pool emulator to generate more complete database queries as specified in the TPC-H benchmark. In the second set of experiments, we evaluate the performance of the database-aware target for non-database applications. The buffer pool emulator measures the overall response time of each query while the OSD code is instrumented to provide an operational breakdown on the time spent at the target.

7.1 Experimental setup

The OSD target was set up on a 3.6 GHz Pentium 4 Dell Dimension 8400 workstation running Linux kernel 2.4.27. A 37 GB, 10,000 RPM Seagate Cheetah 10K7 drive (ST373207LC) is directly attached to the target through a Adaptec 29160 Ultra160 SCSI adapter. The initiator was set up on a 3.0 GHz Pentium 4 Dell Dimension 8300 Workstation running Linux kernel 2.4.27. The host and the target are connected through a gigabit Ethernet network.

The buffer pool emulator performs `READ` and `WRITE` operations in batches of at most 20MB data transfers at a time. We are limited by 20MB because of kernel resource limitations that prevent reserving more than 20MB of socket buffer at any time. All `READ` and `WRITE` operations are done in synchronous mode, i.e., the buffer pool emulator does not issue the next command until the previous command has been processed in its entirety. The above two limitations stem from the fact that in our current implementation we do not have a mechanism to split a large request into smaller requests that can be queued up at the target with an appropriate flow-control mechanism in between. In a more complete implementation, this task would be handled by a low-level iSCSI driver. However lack of this feature does not prevent us from evaluating the tradeoffs and benefits of database-aware object-based storage devices.

7.2 Database performance

The set of experiments in this section investigate the effect of disk layout strategies on the performance of representative buffer pool manager workloads. To evaluate the performance of CSM, we have also implemented NSM and DSM layout models. For DSM, we use the chunk-based reconstruction algorithm [15] to reconstruct full records.

7.2.1 Microbenchmarks

Table scan operations: In the first experiment, we compare the performance of the NSM, DSM and

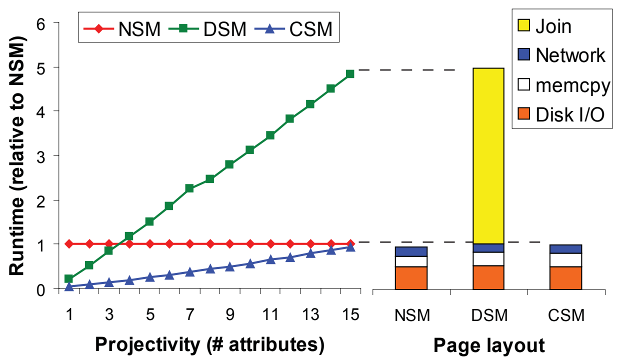

Figure 7: The graph on the left compares the performance of NSM, DSM, and CSM for full table scans, with query payload varying from one to fifteen attributes. CSM achieves best-of-both-worlds performance by equaling or exceeded NSM and DSM for all projectivities. The graph on the right shows the components of runtime for a full table scan of fifteen attributes. The high cost of DSM is due to joining attributes into full records.

CSM layouts through a range query microbenchmark: `SELECT AVG(a1), AVG(a2) ... FROM R WHERE Lo < a1 < Hi`. The relation R has 15 attributes each of size 8 bytes. R is populated with 8 million records amounting to roughly 1GB of data. The buffer pool emulator generates a `READ` request to read a 20MB batch of records and applies the tuple selection criteria (the `WHERE` clause) to every record in the batch. It sends a series of such `READ` commands to scan the entire table. We vary the number of attributes in the `SELECT` clause from 1 through 15 and measure the effect on the overall response time. Since we do not use indices and scan the full table to perform tuple level reconstruction, the values of Lo and Hi do not have a significant effect on performance.

The graph on the left in Figure 7 compares the performance of the query, as observed by the initiator, for the three layouts, while varying projectivity. We use NSM as the baseline for comparison since runtime does not vary with projectivity. The runtime for NSM is 46.34 seconds independent of the projectivity. It is seen that CSM layout performs best at both ends of the projectivity spectrum. At low projectivity, it has comparable performance to DSM. CSM and DSM have good performance at low projectivity since column-major layout enables the DBMS to just fetch those attributes required by the query. A row-major layout like NSM does not take the project operation to the I/O level and fetches all attributes, regardless of how many of them are required. However, the performance of DSM quickly degrades due to the overhead imposed by joining attributes together. On the other end of the projectivity spectrum, where all attributes in the relation are used in the query, NSM performs best by sequentially reading full records from disk. CSM matches the NSM performance by leveraging efficient track-aligned sequential accesses and by scattering the data directly onto the iSCSI buffer. Recent work on column-oriented data stores [26] shows that the cross-over point, seen in Figure 7, between NSM and DSM can be pushed further to the right. However, CSM performance is always better than NSM, providing *best of both worlds* performance of NSM and DSM.

The graph on the right in Figure 7 shows the components that contribute to total runtime of a full table scan that fetches all 15 attributes. The *Disk I/O* component represents the time spent on performing the read operation. *Network* represents the sum of the iSCSI protocol overhead and the time taken to transfer the data over the network. The *memcpy* cost represents the time spent scattering the data read from the disk into the iSCSI buffer. Also, we observe that total time spent in database-specific modules of the the target is never more than 0.6% of the total runtime of a full table scan operation. This shows that database-awareness does not introduce high overhead for database workloads. Lastly, the *Join* cost is the time spent joining individual attributes into the payload defined by the projectivity of the query.

We see that overall runtime of CSM closely matches that of NSM and the *join* cost dominates the runtime of DSM. Since all three techniques fetch the same amount of data from the disk (approximately 1GB), the I/O and the network time components have very similar values across the three layouts. DSM needs to perform join operations because we want to access the data in row major order, one tuple at a time. Hence DSM remains an inefficient layout for workloads that have predominant row-major accesses. In a similar vein, the performance of NSM will suffer when the data needs to be accessed in a column major order (one column at a time). CSM stores each column on a separate track, enabling efficient, sequential columnwise access. For row-major access, CSM avoids the join operation by directly scattering the data read from each column into the target buffers. However when the amount of data required per column is not high, the advantage of sequential access is impaired through positioning cost that is caused due to frequent track switches. In this case, our CSM I/O Manager employs semi-sequential access to provide good performance. Therefore CSM can access the relation in both row-major and column-major orders efficiently.

Point queries: In the second experiment, we evaluate the performance of a point query which generates a `READ` operation to a random tuple. The projectivity is varied as in the previous experiments and the response time is observed for the three layouts. NSM is expected to have the best performance since it has to pay the disk positioning cost only once and then the entire record can be read sequentially. The access time is constant because only a single disk block (512B) needs to be read to obtain all the attributes of the required tuple. For both CSM and DSM, sequential access is not possible due to the column-major storage on disk.

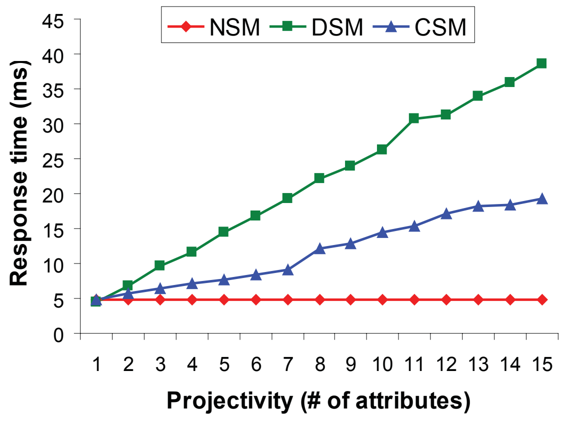

Figure 8: Response time observed at the buffer pool emulator for the point query microbenchmark for NSM, DSM, CSM layouts.

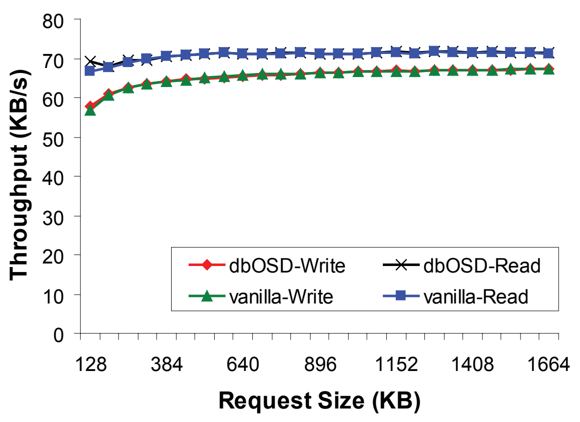

Figure 10: Comparison of read and write performance for the database-aware and vanilla targets.

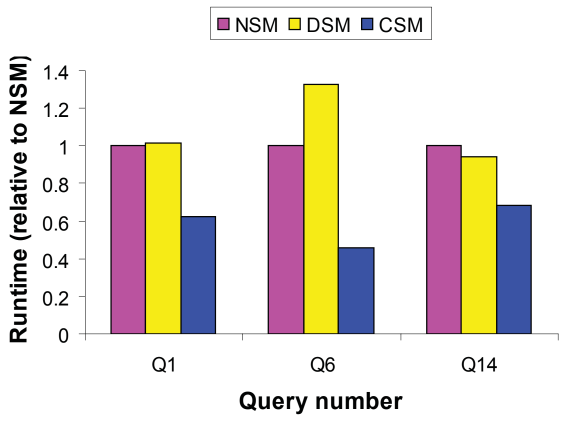

Figure 9: Performance of NSM, DSM and CSM layouts for TPC-H benchmarks.

Since the number of blocks required per attribute is small (1 data block), CSM uses semi-sequential access to read the record. DSM incurs random reads for each attribute, and pay the additional join cost that CSM avoids. Figure 8 shows that our intuition holds with NSM performing best and DSM the worst. CSM performs close to NSM at low projectivity since it pays the position cost only once to access the disk block holding data for the first attribute. From there, it can access other attributes belonging to the same record semi-sequentially. But as the projectivity increases, performance of CSM degrades. However we see that the semi-sequential access of CSM is better than a random access as experienced by DSM, by at least a factor of two.

7.2.2 TPC-H Benchmarks

Next, we measure the performance of our target for a decision support system workload using the TPC-H benchmarks. Figure 9 shows the runtime for three representative queries (relative to NSM) for the three data layout strategies. We observe that CSM performs best across all three queries by taking advantage of the low projectivity of queries and avoiding the join cost. The performance of NSM and DSM depends on the projectivity of the query. For queries 1 and 6, the join cost of DSM dominates leading to higher runtime than NSM. Query 14, on the other hand, uses only 6 attributes out of 20 attributes across two tables (lineItem and Part) that are joined. In this case, the I/O cost expended to read the 14 attributes not required by the query degrades the performance of NSM.

7.3 Standard application performance

In the previous set of experiments we showed that our database-aware target shows good performance benefits for database operations. In this section, we investigate whether database-awareness incurs performance penalty for non-database applications. To observe the overhead of database-awareness, we compare the read/write performance of the database-aware target with a vanilla target that just implements the base T-10 standard. Figure 10 shows throughput for read and write operations for the database-aware target (dbOSD) and the vanilla target. First we see that the throughput initially increases with sequentiality of the read/write operation before reaching a plateau. More importantly, there is almost no performance penalty for non-database applications for both read and write operations across various request sizes. Therefore non-database applications will see very little or no effect on their performance due to database-awareness at the target.

8 Conclusion

We have designed and implemented a prototype database-aware OSD that is aware of the relations that it stores, their schema, and which subsets of a relation are being requested by the database software. Our prototype demonstrates that database-aware storage systems can provide the benefits of detailed knowledge of storage parameters, while avoiding the shortcomings of previous storage-aware approaches. Database-aware storage, and application-aware storage in general, is a more practical approach to exploiting low-level storage characteristics to improve performance.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *Architectural Support for Programming Languages and Operating Systems*, pages 81–91. ACM, 1998.

[2] ANSI. Information Technology - SCSI Object-Based Storage Device Commands (OSD) revision 07. Standard, ANSI, June 2003. www.t10.org/ftp/t10/drafts/osd/osd-r07.pdf.

[3] ANSI. Information Technology - SCSI Object-Based Storage Device Commands (OSD). Standard ANSI/INCITS 400-2004, ANSI, Dec. 2004.

[4] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *ACM SIGMOD International Conference on Management of Data*, pages 268–279. ACM Press, 1985.

[5] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Summer USENIX Technical Conference*, pages 177–190, 2002.

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

[7] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Architectural Support for Programming Languages and Operating Systems*, pages 92–103, 1998.

[8] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Conference on File and Storage Technologies*, pages 73–86. USENIX Association, 2004.

[9] Information Storage Industry Consortium (INSIC). Data Storage Devices and Systems (DS2) Roadmap. www.insic.org/2005_insic_ds2_roadmap.pdf, January 2005.

[10] OSD Reference Implementation, Intel Corporation. http://sourceforge.net/projects/intel-iscsi/.

[11] Y. Lu, D. H. C. Du, and T. Ruwart. Qos provisioning framework for an osd-based storage system. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 28–35, Washington, DC, USA, 2005. IEEE Computer Society.

[12] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Conference on File and Storage Technologies*, pages 275–288. USENIX Association, 2002.

[13] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based Storage. *Communications Magazine*, 41(8):84–90, 2003.

[14] R. Ramakrishnan and J. Gehrke. *Database management systems*. Number 3rd edition. McGraw-Hill, 2003.

[15] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *International Conference on Very Large Databases*, pages 430–441. Morgan Kaufmann Publishers, Inc., 2002.

[16] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. In *ACM SIGMOD International Conference on Management of Data*, pages 13–21. ACM, 2000.

[17] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. In *International Conference on Very Large Databases*. Morgan Kaufmann Publishing, Inc., 2003.

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

[19] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Conference on File and Storage Technologies*, pages 259–274. USENIX Association, 2002.

[20] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. In *Conference on File and Storage Technologies*. USENIX Association, 2004.

[21] M. Shao, A. Ailamaki, and B. Falsafi. Dbmbench: fast and accurate database workload representation on modern microarchitecture. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 254–267. IBM Press, 2005.

[22] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *International Conference on Very Large Databases*, 2004.

[23] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. In *Conference on File and Storage Technologies*, pages 239–252. USENIX Association, 2005.

[24] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically smart disk systems. In *Conference on File and Storage Technologies*, pages 73–88. USENIX Association, 2003.

[25] Storage Networking Industry Association Object-Based Storage Devices Technical Work Group. www.snia.org/tech_activities/workgroups/osd/.

[26] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column oriented DBMS. In *International Conference on Very Large Databases*, 2005.

[27] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based extraction of local and global disk characteristics. Technical Report CSD–99–1063, University of California at Berkeley, 2000.

[28] R. VanMeter. SLEDs: storage latency estimation descriptors. In *IEEE Symposium on Mass Storage Systems*. USENIX, 1998.

[29] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–156, 1995.