Final Project Report

A Survey of techniques used to reduce the Semantic Gap between Database Management Systems and Storage Subsystems

Biplob Kumar Debnath

Nagapramod Mandagere

Nov 28, 2006

A Survey of techniques used to reduce the Semantic Gap Between Database Management Systems and Storage Subsystems

1. Introduction

One of the most important modules of any database management system is the storage manager module. This module essentially controls the way the data is allocated, accessed, and stored on storage devices. Storage subsystems are typically virtualized for the purposes of consolidation, easy of management, reducing interdependence, etc. Due to this virtualization now the database storage managers neither have a strict control over the physical layout of data nor are they aware of the internal characteristics of storage subsystems, and apply some coarse rules of thumb to optimize its access. On the other hand, storage subsystems do not have semantic knowledge of the data that they are storing, again relying on their own rules of thumb to manage such workload-dependent tasks as pre-fetching, caching, and data layout. The end result is both sides are working blindly to optimize their performance without not knowing what other side is doing. Various studies [1][2] emphasize the importance reducing information gap between applications and underlying storage devices. Over the years, various techniques have been developed to reduce the semantic gap between database management systems and storage systems. In this project we plan to survey various approaches used by researchers or implementers in order to reduce this semantic gap.

Storage technology has evolved much in the intervening years and disk-resident processing got attention from the research community again in 1990s. The biggest change was widespread use of disk arrays that use a large number of disks working in parallel. Special-purpose silicon cores in database machines are replaced by general-purpose embedded processing and increased memory cores. Numerous parallel algorithms for database operations, such as joins and sorts have been developed for different architectures, such as shared-nothing, shared-memory, and shared disk since the inception of specialized database machines. Serial communications were able to provide enough bandwidth to disk to overcome the message passing overheads.

2. Classification

As disk arrays have become very popular, storage subsystems have moved more swiftly towards virtualization, in the process, increasing the gap between storage and applications. To cope with this trend, researchers have proposed various mechanisms to reduce the gap between applications and the data store. These efforts can be broadly grouped into four categories, namely – Building Special Purpose Database Machines, Server side changes, Storage side changes and interfaces that facilitate better communication between the two. In Figure 1, various components are marked where various modifications can be made to reduce this semantic gap.

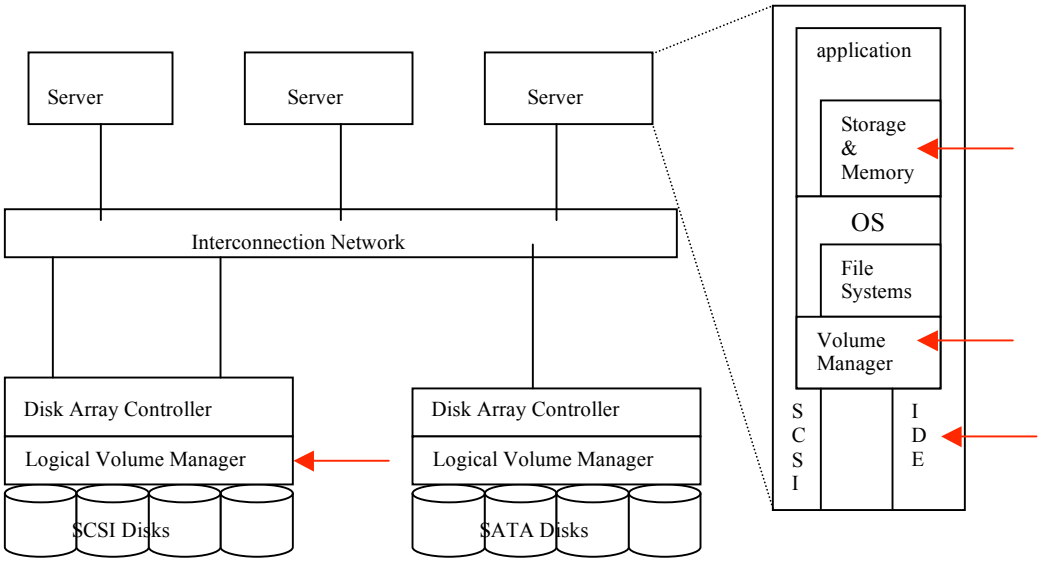

Figure 1. Memory & Storage hierarchy (-> indicate alternatives for improvement)

2.1. Building Special Purpose Database Machines

The idea of placing intelligence in storage systems to help database operation was explored extensively in the context of database machines in late 1970s and late 1980s. Database machines can be classified into four categories depending on disk processing [8]:

1. Processor per head: DBC[14], SURE[15]
2. Processor per track: CAFS[16], RARES[17], RAP[18], CASSM[19]
3. Processor per disk: SURE[15]
4. Multi-processor cache: DIRECT[20], RDBM[21], DBMAC[22], INFOPLEX[23], RAP.2[24]

In all of the architectures, there was a central processor which pushed simple database operations (e.g., scan) closer to disk, and achieved a dramatic performance improvement for these operations. The main counter-arguments are summarized by Boral and Dewitt [11]. First, most database machine use special-purpose hardware, such as associative disks, associative CCD devices, and magnetic bubble memory which increased the design time and cost of these machines. Again, the performance gain was not enough to justify the additional cost incurred by this hardware. Second, although the performance was impressive for scan operations, but for the complex database operations, such as sorts and joins did not provide significant benefits. Third, the performance offered by database machines can be easily achievable by smart indexing techniques. Fourth, CPU processing speed was improving much faster than the disk transfer rates improve, so CPU was sitting idle. Fifth, the communication overhead between processing elements were high. Finally, database vendors did not agree to rewrite their legacy code base to take advantage of features offered by this new hardware.

On the server side multiple schemes for memory and page management have been explored by a number of researchers. Main focus has been on decoupling in-memory page layout form the actual storage layout. Clotho [5] is a buffer pool and storage management architecture that decouples in-memory page layout and the storage layout.

On the storage side multiple alternatives have been explored, but most of them focus on moving some application functionality onto the storage devices. Active disk, semantic disk etc. are techniques that fall under this category. Another option is to build intelligent Logical Volume Managers that allow the applications to exploit characteristics of underlying group of disks. Atropos [3] is one such disk array Logical Volume Manager that exploits storage characteristics.

Little effort has gone into the interface itself. SCSI has existed since early 1980s.
Object based Storage interface is new paradigm that has gained some attention recently.

2.2. Server Side Modifications

Relational database systems store data in fixed size pages. The pages size can range from 2KB to 4KB. To access individual records of a table requested by a query, a scan operator of a database system access main memory. But, not all pages can be accommodated in the main memory and hence non-volatile disks are used for permanent storage. An access of data involves fetching a page from disk and copying it into memory. It's the job of the database storage manager to perform the task of translation and retrieval of these pages. Storage model is basically all the layout information needed to interpret the information in a page. Pages contain header information that describes what records are contained within and how they are laid out. The choice of the storage model/page layout has a drastic impact on the performance of the database system [13]. In commercial system different page layout/storage models have been used, namely N-array storage model (NSM), Decomposition Storage Model (DSM) and Partition Attributes across (PAX).

2.2.1. N-array Storage Model (NSM)

In NSM[13], all attributes of a relation are stored in a single page and full records are stored within a page one after the other. This approach provides maximum benefit for queries with full record access, which is commonly seen in online transaction processing workloads. In this model an implicit assumption is made about the storage system layout. A page is mapped to consecutive LBAs considering the assumption that accessing consecutive LBAs is faster than accessing random LBAs. Usage of Logical Block Addressing mode hides the actual disk layout characteristics.

2.2.2. Decomposition Storage Model (DSM)

In DSM [13], only one attribute is stored per page. This approach provides maximum benefit for queries that access a small number of attributes of a table, which is often the case in decision support systems. Here, DSM pages with consecutive records containing

the same attribute can be mapped into extents of contiguous LBNs. A single large IO access can bring in multiple pages within an extent thus increasing IO efficiency and also assists in pre-fetching. Again an implicit assumption is that access to consecutive LBAs is faster than accessing random LBAs.

2.2.3. Partition Attributes Across (PAX)

PAX [13] is a page layout optimized for processor cache performance. In PAX, page is partitioned into separate minipages. A single minipage contains data of only one attribute and occupies consecutive memory locations. A single page contains all attributes or minipages for a given set of records. A scan of individual attributes in this system accesses consecutive memory locations there by taking advantage of cache line pre-fetch logic implemented for the processor. By carefully designing the cache boundaries, one can in effect cause a page fault to pre-fetch data for several records. Again, it does not address the actual disk geometrics.

The above approaches do not address all level of memory hierarchy. They try to use a static predetermined approach based on prior knowledge across various levels of memory hierarchy. This leads to page layouts that are highly optimized for only a specific type of workload. Moreover, none of these methods exploit the disk geometrics.

2.2.4. Decoupling in memory page layout and storage layout - Clotho

The main motivation behind the design of Clotho [5] is the fact that it is very difficult to design a static scheme for data placement in memory and on disk that performs uniformly across different types of workloads and different types of disk subsystems. The basic idea behind this work is that since performance characteristics of each level of memory hierarchy differ vastly, different data organization needs to be used at different levels. The authors propose to decouple the in-memory data layout from that of the disk/storage data layout. This facilitates implementation of data layouts tailored to different levels without compromising performance at other levels.

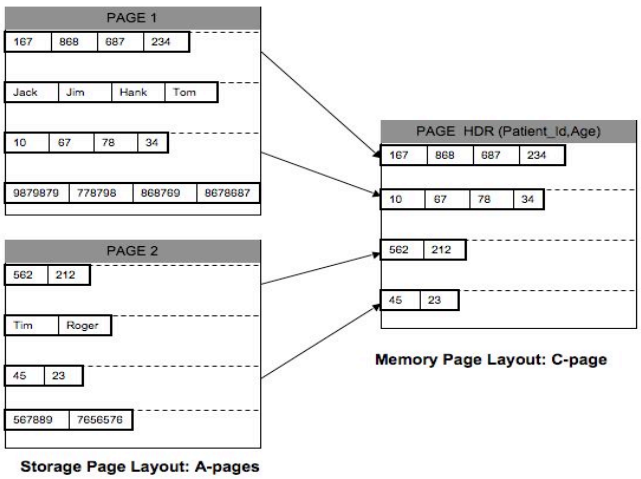

Figure 2. Decouple of in-memory page layout and storage page layout

The basic idea is illustrated in Figure 2. Here, relation R shown in Figure 3 consists of four attributes: Patient_ID, Name, Age and Phone.

Relation R

Patient _ID	Name	Age	Phone
167	Jack	10	9879879
868	Jim	67	7787987
687	Hank	78	868769
234	Tom	34	8678687
562	Tim	45	567889
212	Roger	23	7656576

Figure 3. Relation R

The following select query is applied to the relation R.

SELECT Patient_ID
FROM R
WHERE Age > 50

At the storage level, the data is organized into A-pages. An A-page contains all attributes of the records there by only one A-page needs to be fetched to retrieve a full record. 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. There by this approach fully exploits sequential scan for evaluating predicates, and also allows or facilitates careful placement A-pages on the device to ensure close to sequential perfromance when reconstructing a record.

C-page is the in-memory representation of a page. A C-page 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, a C-page only contains values for the attributes the query accesses. Since, the query in the example only uses the Patient_ID and Age, the C-page only includes these two attributes, maximizing memory utilization. C-page can now make room for data from two A-pages to fill up the space saved from omitting unwanted attributes. For a detailed discussion on the C-pages and A-pages and the detailed working of Clotho, the reader is advised to refer [5].

2.3. Storage Side Modifications

In this section we first describe the working of Logical volume managers (LVM). An LVM is a piece of software that is responsible for allocating space on storage devices in a way that is more flexible than conventional partitioning schemes. A volume manager can concatenate, stripe together or otherwise combine partitions into larger virtual ones that can be resized or moved. This process can be thought of as a form of virtualization as it turns storage into a more easily manageable and transparent resource.

Current disk array LVMs try to exploit the unique performance characteristics of their individual disk drives. Since an LVM sits below the host's storage interface, it could

internally exploit disk specific features transparent to the host. Most LVMs use data distribution schemes designed and configured independently of the underlying devices. This again assumes that access to sequential LBAs is faster than random LBAs. They stripe data across their disks, assigning fixed-sized sets of blocks to their disks in a round-robin fashion. Disk striping can provide effective load balancing of small I/Os and parallel transfers for large I/Os if a good stripe size is chosen. The stripe sizes have remained relatively constant within the range of 2 – 32 KB. However, the track sizes have changed significantly as memory density or aureal density has changed. Typical track sizes are anywhere between 500 KB to 2MB. Another factor that has lead to increased track size is use of perpendicular recording. But, disk arrays currently do not seem account for growing track size over time. As a consequence of this mismatch, medium to large sized requests to the disk arrays result in suboptimal performance due to small inefficient disk accesses.

2.3.1. Atropos – LVM Built to Exploit and Expose Disk Characteristics

Atropos is a disk array LVM [3] built with the sole purpose of exploiting characteristics of its underlying collection of disks that it virtualizes. The basic idea behind Atropos is illustrated in Figure 4. Consider an application that requires efficient access to two-dimensional structures in both dimensions. A relation in database qualifies as a perfect use case.

	Col 1	Col 2	Col 3	Col 4
Row a	A1	A2	A3	A4
Row b	B1	B2	B3	B4
Row c	C1	C2	C3	C4
Row d	D1	D2	D3	D4
.	.	.	.	.
Row z	Z1	Z2	Z3	Z4

Figure 4. A 2-dimensional Data Set

The example in Figure 2.3.1.1 depicts a two-dimensional data structure consisting of four columns 1,2,3,4 and many rows A-Z. To map this two-dimensional structure into a linear space of *LBN*s, conventional systems decide a priori if row major or column major order is likely to be accessed most frequently. In the figure column major order has been chosen for illustration with disk stripping across two disks.

The mapping of each element to the *LBN*s of the individual disks is depicted in Figure 2.3.1.2(a) in a traditional layout. Accessing data in the row major order, however, results in disk I/Os to disjoint *LBN*s. For the example in Figure 5(a), an access to row A1,A2,A3,A4 requires four I/Os, each of which includes the high positioning cost for a small random request. The inefficiency of this access pattern stems from the lack of information in conventional systems; one column is blindly allocated after another within the *LBN* address space. *Atropos* supports efficient access in both orders with a new data organization as shown in Figure 5(b). This layout maps columns such that their respective first row elements start on the same disk and enable efficient row order access. This layout still achieves sequential, and hence efficient, column-major access, just like the

traditional layout. Accessing the row A1,A2,A3,A4 however, is much more efficient than with traditional approach. Instead of small random accesses, the row is now accessed semi-sequentially in one disk revolution (worst case), incurring much smaller positioning cost as a result of the first seeks only and no rotational latency.

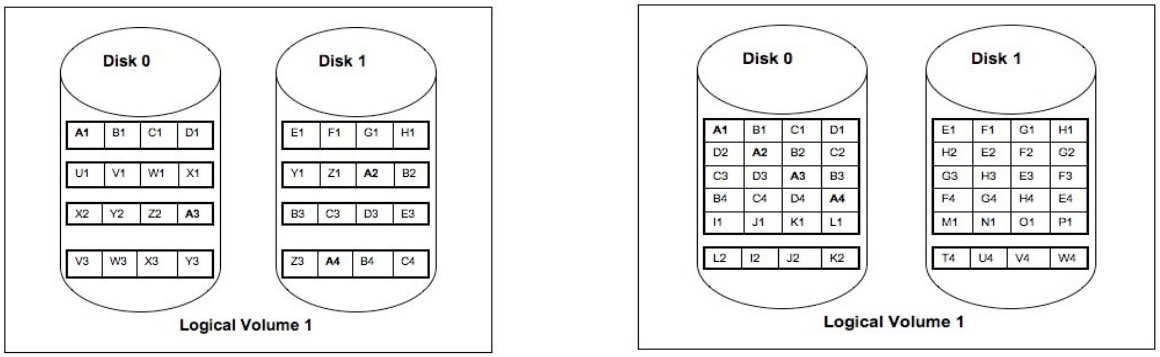

Figure 5. (a): Traditional Mapping **(b): Atropos Mapping**

In this work the authors demonstrate that a logical volume manager can be built to exploit and expose the underlying disk characteristics. For a detailed discussion on the how the volume manager interfaces with the host applications and the detailed working of Atropos, the reader is advised to refer [3].

2.3.2. Active Disks – Offloads Simple Processing to Disks

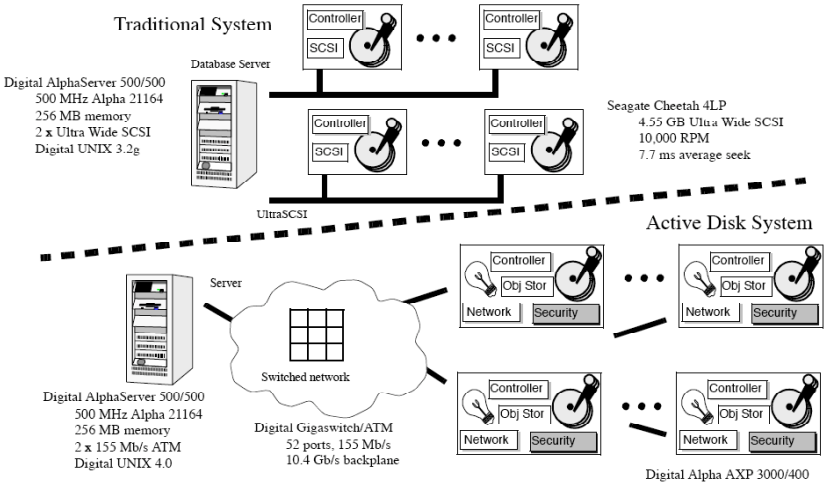

Figure 6. Disk Subsystem using Active Disks [6]

Active disks [6] leverage storage technology advancements and parallel algorithms improvement for database operations over the intervening years. Commodity disks are no longer mere spinning magnetic storages for data. Disks are embedded with a processor, memory, cache, and network connection. It can perform some computation in addition to

storing data. [6] proposes the idea of leveraging disk processing power to execute selective portion of database operation in parallel close to data. It discuses how one can map all basic database operations (e.g., select, join, and project) on Active Disk system with proper low-level primitives. The architecture is shown in Figure 6.

For *select* operation *(e.g., select l_quantity, l_price, form lineitem where l_shipdate >= '2006-09-07')*, the *where* clause in SQL query is passed to individual disk. Each disk searches its local data and returned the qualified records which satisfied the where condition. Host parses the query and sends select condition to individual disk.

For sorting a replacement selection algorithm is used as it shows good adaptive behavior with changing memory conditions. Each disk-drive works on its local data and sends the information back to host, which finally produce the final sorted order. For aggregation operation (*e.g., select sum (l_quantity), sum (l_price *(l-l_discount)) from lineitem group by l_return),* each disk-drive sends the local sum and count (for sum). The host produces the final aggregate result by combining these local summary results passed by all local drives. After receiving the query which involves sort or aggregation operation, host sends modified query for each disk drive which tells them to send the local sum and counts.

Join operation (*e.g., select l_price, l_quantity, p_partkey from part, lineitem whre p_name like '%white%' and l_partkey = p_partkey*) is implemented using bloom-filter strategy. This strategy is used as it does not depend on the size of the relations. A bit vector *b[1…n]* is initialized with '0's is created. A hash function is applied to the join attribute of R, and corresponding bit in b is set to '1'. Then this bit vector is passed to all the disk drive, which applies the same hash function to join attribute of S and if it maps to a bit in b which is '0' that tuple is discarded. In this way it reduces the number tuples of S sent to the host for joining.

To support database operation, some primitives are added the disk drive. Arithmetic operations are added for aggregation and comparison operators are added for scan and sort operation. Replacement selection sort primitive is added in the disk-drive to support sort and aggregate operation, and a semi-join primitive which uses bloom-filter is added to support join operation.

By moving portion of database processing directly at disk one can exploit parallelism in large storage system, and early discard unwanted data which can reduce the network traffic dramatically. [6] Shows how Active Disk system can use disk processing power to overcome the I/O bottleneck. Active Disk does not consider geometry aware data layout for efficient database operations.

2.3.3. Database Aware Semantic-Disk Approach

Database Machines require specialized hardware and Active Disks requires sophisticated programming environments. In contrast, semantic-disk [4] approach increases its functionality by placing high-level semantic knowledge about DBMS within the storage systems. Embedded with high level of knowledge DBMS and using its own low level control, semantic disks can improve performance with better layout and caching, can

provide improved reliability and security guarantees. The main motivation behind semantic-disk approach is that current block based interface storage systems interface will not changed in near future as developers like block-based interface for its ease of use and to support legacy applications. So, semantic disk approach attempts to adapt storage systems to learn about the applications behavior running above it and use this knowledge to improve performance.

The main mechanism behind semantic disk is to find correlations among the blocks. It tries to track which relational table a particular block has been allocated to. One way to find this information is to consult the write-ahead log (WAL) entries. DBMS tracks all operations that change the on-disk contents, by writing it in WAL. To support ACID properties, WAL entries will be written on disk before the actual change is being performed. Due to this property, inferring knowledge WAL is quite simple. Semantic disk takes advantage of this property to find the relationship between a block and table, which can be used to derive co-relations among blocks. Now, DBMS needs not to aware of the underlying storage system properties, storage systems gather information transparently using its high level knowledge about the application behavior. This approach is called *log snooping.*

[4] shows that performance can improved more if DBMS explicitly collects three additional types of statistics and periodically writes them to storage devices. As these statistics will be used for performance enhancement, it does not need transactional semantics; as a result for logging these statistics the overhead will be very low. The *access time* of a particular block or table is the most important statistics which DBMS can communicate with the semantic disk. It can be used to derive other useful information. The *access correlation* for entries such as tables and indices is another useful statistics DBMS can record for each query. This information can help storage device to group related tables and to find correlation among related tables. Third useful statistics is the *access counts*, number of queries that accessed a particular table over a fixed period of time. In addition to log snooping, using these extra information semantic disks can improve performance with better layout and caching, can improve reliability, and can provide extra security guarantees.

2.4. Interface Modifications

.

2.4.1. Object-Based Storage Device (OSD)

[7] examines the approach of passing semantic information from a database system to the storage subsystem to bridge the information gap between these two levels. Recently standardized Object-Based Storage Device (OSD) interface moves low level storage functionalities close to data and provides an object interface to access the data. The OSD model is shown in Figure 7. [7] leverages OSD interface for communicating semantic information database to the storage device. It discusses how we can map relation of a database to an OSD object; and how we can read and write database relation

efficiently taking advantage of geometry aware data-layout through additional OSD interface.

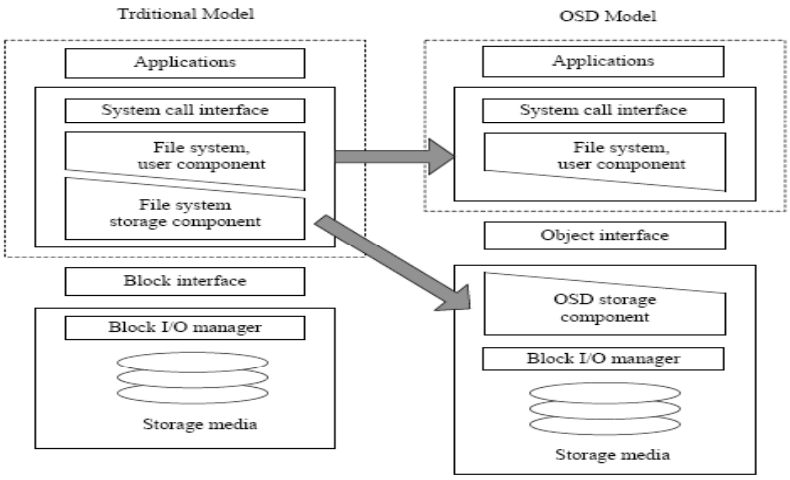

Figure 7. Object Based Storage Model [7]

An Object is a logical unit of storage. It has interface like file-access (e.g. read, write), securities policies to authorize access, and has attributes which describe the characteristics of the data it is storing. Object size can dynamically grow and shrink. Objects can store any types of data (e.g., text, video, audio etc.). It can even store an entire database. The device which stores objects is called OSD. OSD differs from block-based storage device in terms of interface, but not in terms of physical media. OSD puts a wrapper around traditional block-based storage device, provides a clean interface, and hide the view of block-based storage device. OSD can be a single disk, tape drive, optical media, or storage controller with an array of drives.

OSD provides the opportunity of push more intelligence on the storage devices. When data is stored as an object, various characteristics of data, for example data type, access pattern, access frequency, reliability requirements are stored as object attributes. Storage device can use these attributes to lay-out data efficiently and serve data to upper application layer which can help various database operations.

We can offload the space management component of a DBMS to OSD which can makes data sharing easier and flexible. Traditionally, each file system lays data on disk in its own manner and maintains the block-level metadata. As different file systems lay data differently on disk and maintain different meta-data, so applications need to be aware file system specific meta-data information which makes cross platform data sharing very difficult. With OSD as space-management component is offloaded to objects so dependency between file system specific metadata and storage system is completely removed, this greatly improves the scalability of clusters.

Another handy feature of OSD is the security. In OSD every access is authorized, and as data path and control path is totally different, and authorization is done without accessing the central authority the performance is greatly improved. OSD provides security in finer granular in low cost compared to block-based devices.

Using OSD interface, database can inform the characteristics of a relation to the storage system. A relation is stored in a single object using an existing storage model (e.g., NSM or DSM or CSM). Data pages can be mapped to fixed-sized ranges (e.g., 8KB) within objects, as if the OSD were a standard block device. When an object is created to store a relation, DBMS can inform OSD of the schema being stored. The length of each field in a single record will stored as a shared attribute. Armed with the schema and the characteristics of the underlying disks, OSD can generate a geometry-aware data layout to store the relation.

Currently, OSD standard provides read and write interface which reads from and write to object data in linear fashion using byte-offset. As, database is considered two-dimensional, OSD interface need to extended to support efficient database operation embedded with semantic information. The following new commands are suggested in [7]:

- *CREATEDB(schema)*
- *READDB(Record Offset, Length, Field Bitmap)*
- *WRITEDB(Record Offset, Length, Field Bitmap)*

CREATEDB command creates a new objects and the schema of the relation is passed to OSD which stored it as a shared attribute. READDB and WRITEDB command access database using record offset and operates only on the attributes specified by field-bitmap and applies operation to only length number of records. Now, database can offload address translation, low level data layout, command processing functionalities to OSD. Database make request to the OSD specifying the required data and memory address where these data need to be placed, OSD handles rest of the task.

3. Comparison of Various Approaches

NSM, DSM, and PAX model [13] improves performance by layout relational pages in different ways. NSM is good for OLTP workload, DSM is good for DSS workload, and PAX is a trade off between two of them. A comparison among them is shown in the Table 1 below:

Page Layout	Cache-memory performance		Disk-memory performance	
	Full-record access	Partial-record access	Full-Record access	Partial-Record Access
NSM	√	x	√	x
DSM	x	√	X	√
PAX	√	√	√	x

Table 1. Comparison of NSM, DSM, and PAX

Fractured mirror approach [12], maintains two copies of the database. One copy uses NSM page layout, another one uses DSM layout. Depending on the query either is

forwarded to either NSM or DSM copy. The problem this approach is it increases space requirement by 100%.

Apropos logical volume manager [3] exposes the underlying storage system layout to the applications running above it. Now, DBMS use this information to write data, rows are written in semi-sequential manner, and columns are written in sequential manner so that for column wise access it performance will be close to DSM performance, but for row wise access performance will be better than the random access. But the problem, as application are tight coupled with underlying storage devices characteristics, if storage devices layout changes, we have to rewrite some part of the DBMS storage manager.

Semantic disk approach [4] can work transparently without DBMS knowing that storage system is performing log snooping. Storage systems have a very high level knowledge of the DBMS working behavior. It uses WAL entries to find correlations among blocks, uses this knowledge for better caching, layout, reliability, and security guarantees. With some help from DBMS side it can improve performance even better. In this case, the change needed in DBMS is less and overhead for record logs of access time, access counts, access correlation is very less. As technology trends show that block based interface will be changed in near future, semantic disk seems be very promising.

In contrast to semantic disk, database machines require specialized hardware and active disks required a sophisticated programming environment. Active disks [6] leverage the processing power of storage device to process application level code near data. It mainly focuses on how to partition data among disk and host processors so that data transferred will be minimized. But the problems of Active disks approach is that we have change legacy DBMS code, and we have to write disk firmware code aware database operations.

Object-based storage device [7] moves low-level storage functions to the storage device and provides an object interface to the applications. The underlying storage device is still block-based; OSD just put an object wrapper above it. It also provides a clean interface thorough which application can communicate QoS requirements, characteristics of data, and access patterns to the storage device. Now, storage device can either ignore it, or with it low control use this information for geometry aware data layout, caching, and improving reliability and security guarantees. The benefit of OSD is as application communicate with storage through predefines storage so as long as interface remains same, it does not what change is happening in the underlying storage devices. But using OSD we have change existing DBMS code. We have change current block-based storage functions to OSD interface compatible. Another problem, currently OSD has no special interface for efficient DB operations, as we add extend the current OSD standard suitable for database operations. It is a speculation that may in terms performance OSD will not that much promising, but if the concerns is scalability, ease of management, and clean interface, OSD can be a one of the viable candidate solutions. Table 2 is the summary of all of our findings.

	Data Layout Awareness on disk	DB operation processing On Disk	Changes needed in DBMS code	User interface provided	Page Layout awareness	Early Discard	Specialized Hardware needed	Disk Firmware Update Needed
Database Machines	x	√	x	x	x	x	√	√
NSM	x	x	x	x	√	x	x	x
DSM	x	x	x	x	√	x	x	x
PAX	√	x	x	x	√	x	x	x
Fractured Mirrors	√	x	x	x	√	x	RAID 1	x
Atropos LVM	√	x	√	x	x	x	x	√
Active Disk	x	√	√	√	x	√	Disk processing power need	√
Semantic Disk	x	x	√	x	x	x	x	√
OSD	√	√	√	√	√	√	x	√

Table 2. Summary of all findings

4. Conclusion

In this paper, we explored different techniques to reduce the semantic gap between storage device and DBMS running above it. Various approaches such as building special machines, extracting information from the storage devices, sending information about the application behavior to storage devices, and changing interface of storage devices are used to solve the problem. Every solution has some pros and cons. As current block based interfaced well accepted by the industry, and programmers likes the liner abstraction of storage device, so semantic disk approach [4] seems to be very promising. As it requires very less change on DBMS part and assume very little high level knowledge about DBMS in disk firmware. For long run, OSD approach [7] seems to be a good candidate solution. As OSD standard is still a ongoing process, there are ample scopes for researchers to explore flowing issues: how database tables can be mapped into OSD objects, what additional interface we need for efficient database operation, how to define geometry aware layout, what minimum information DBMS will supply to OSD devices, and what we need in current DBMS stack to support OSD devices.

5. References

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

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

[3] Jiri Schindler, Steven W Schlosser, et al, Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks, *In 3rd USENIX Conference on File and Storage Technologies FAST 04, CA. March 2004*

[4] Muthian Sivathanu, Lakshmi N Bairavasundaram, et al, *Database Aware Sematicaly Smart Storage*, FAST 2005.

[5]Minglong Shao, et al, *Clotho: Decoupling Memory Page Layout from Storage Organization*, VLDB 2004.

[6] E. Riedel, C. Faloutsos, and D. Nagle. *Active Disk Archittecture for Database*s. Technical Report CMU-CS-00-145, Carniegie Mellon University, April 2000.

[7] Steve Schlosser, Sami Iren. *Database Storage Management with Object-Based Storage Devices,* DAMON 2005.

[8] K. Keeton. *Computer Architecture Support for Database Applications*, PhD thesis, University of California at Berkeley, 1999.

[9] A. Acharya, M. Uysal, and J. Saltz. *Active Disk: Programming Model, Algorithms, and Evaluation*, ASPLOS VIII, 1998.

[10] E. Riedel, G. Gibson, and C. Faloutsos. *Active Storage for Large Scale Data Mining and Multimedia*, VLDB 1998.

[11] H. Boral and D. J. Dewitt. *Database Machines: An Idea whose time has passed?,* In 3rd Workshop on Database Machines, 1983.

[12] R. Ramamurthy, D. J. Dewitt, and Q. Su. A *Case for Fractured Mirrors*, In Proc. VLDB, 2002

[13] A. Ailamaki, D. J. Dewitt, M. D. Hill, and M. Skounakis. *Weaving Relations for Cache Performance*. In Proc. VLDB, 2001.

[14] J. Banerjee, et al. *DBC - a database computer for very large data bases*, IEEE Trans. on Computers, June 1979.

[15] H. O. Leilich, G. Stiege, and H. C. Zeidler. *A search processor for data base management systems*", Proc. of the 4th VLDB, 1978.

[16] D. Bitton and J. Gray. *The rebirth of database machine research*, invited talk at VLDB '98, August 1998.

[17] S. C. Lin, D. C. P. Smith, and J. M. Smith. *The design of a rotating associative memory for relational database applications,* Transactions on Database Systems, 1(1):53-75, March 1976.

[18] E. A. Ozkarahan, S. A Schuster, and K. C. Smith. *RAP - associative processor for database management*, AFIPS Conference Proc., Vol. 44, pages 379 - 388, 1975.

[19] S. Y. W. Su and G. J. Lipovski. *CASSM: a cellular system for very large data bases,* Proc. of theVLDB Conference, pages 456-472, 1975.

[20] D. J. DeWitt. *DIRECT - A multiprocessor organization for supporting relational database management systems*, IEEE Transactions on Computers, pages 395-406, June 1979.

[21] W. Hell. *RDBM - A relational data base machine: architecture and hardware design*" Proc. 6th Workshop on Computer Architecture for Non-Numeric Processing, June 1981.

[22] M. Missikoff. *An overview of the project DBMAC for a relational machine*, Proc. of the 6th Workshop on Computer Architecture for Non-Numeric Processing, June 1981.

[23] S. E. Madnick. *The Infoplex database computer: concepts and directions*, Proc. IEEE Computer Conf., February 1979.

[24] S. Schuster, et al. *RAP.2 - an associative processor for databases and its applications,* IEEE Trans. on Computers, June 1979.