# STORAGEDB: Enhancing the Storage Sub-system with DBMS Functionalities

Lin Qiao † Balakrishna R. Iyer § Divyakant Agrawal † Amr El Abbadi † Sandeep Uttamchandani *

§ *IBM Silicon Valley Lab* †*University of California, Santa Barbara* * *IBM Almaden Research Center*
*balaiyer@us.ibm.com* {*lqiao, agrawal, amr*}*@cs.ucsb.edu* *sandeepu@us.ibm.com*

## Abstract

*This paper proposes* STORAGEDB*: a paradigm for implementing storage virtualation using databases. It describes details for storing the logical-to-physical mapping information as tables within the database; handling the incoming I/O requests of the application as database queries; bookkeeping of the I/O operations as database transactions. In addition,* STORAGEDB *uses built-in DBMS features to support storage virtualization functionalities; as an example we describe how online table space migration can be used to support reallocation of logical disks. Finally, we describe our modifications to a traditional RDBMS implementation, in order to make it light-weight. Improving the performance of a traditional DBMS is critical for the acceptance of* STORAGEDB *since the performace overheads are considered a primary challenge in replacing existing storage virtualization engines. Our current lightweight RDBMS has a 19 times shorter invocation path length than the original. In comparision to the open-source virtualization software, namely LVM, the extra cost of* STORAGEDB *is within 20% of LVM in trace-driven tests. (unlike* STORAGEDB*, LVM did not have logging overhead). We consider these initial results as the "stepping stone" in the paradigm of applying databases for storage virtualization.*

## 1. Introduction

With the growing complexity of storage infrastructures in terms of *physical sub-systems, protocols, and users*, it will soon become impossible for the administrators to: 1) manually track the mapping of application data to the storage subsystems, 2) configure and maintain each subsystem, 3) monitor and optimize the usage of the sub-systems at runtime, 4) ensure guarantees of performance, reliability, and security for the application data. There is thus a need for frameworks that hide the complexity from the administrators by providing a logical view of the infrastructure and internally handling the mapping details of the logical disks to the physical subsystem. These frameworks are commonly called *Storage Virtualization* Engines. Existing virtualization engines manifest themselves as: filesystems (such as Lustre, IBM's Storage Tank), intelligent switches (such as SAN Volume Controller), microcode in storage controllers (such as IBM Enterprise Storage System, Hitachi's Lightning). A generic virtualization engine has the following two basic modules: (1) Data-structures for maintaining the mapping of logical disks to the physical storage subsystem. (2) Lookup and caching mechanisms for the logical-to-physical translations as well as the actual data accessed from the storage subsystem. While the above two features support basic storage access, many modern applications [1, 6] require more advanced features, such as: (1) Logging and periodic check-pointing of all changes made to the data and the logical-to-physical mapping. (2) Resource reallocation mechanisms that change the logical-to-physical mapping on the fly, without affecting the user-applications that operate on the logical view. (3) QoS optimizations such as data encryption and data compression.

Existing DBMS implementations already support these advanced functionalities required for storage virtualization. Table 1 enumerates the virtualization requirements and the corresponding database features that can be exploited to implement them. The benefit of using a DBMS for building a storage virtualization engine is that it saves the cost of design, development, and testing of concepts that have been already well-developed and well-tested in DBMSs. Our implementation of proposed prototype took only 0.5 person year. This provides us the opportunity to exploit the DBMS to cheaply and reliably realize these functionalities for storage virtualization. DBMS-based storage virtualization does not invoke much management overhead, thanks to simplicity of storage semantics, although DBMS administration is costly in enterprise environments. Moreover, commercial DBMSs, primarily driven by benchmarking wars (TPC-C, TPC-H, etc. [14]), have significantly improved in performance. Any assumption based on the "conventional wisdom" of database inefficiencies needs to be re-examined.

In this paper, we propose STORAGEDB, as a paradigm

Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)
0-7695-2318-8/05 $20.00 © 2005 IEEE

**Table 1. Correlating Storage Virtualization functions to Database concepts**

| Storage Virtualization Functionality | Database-based Implementation |
|---|---|
| Data-structures for the logical-to-physical mapping | Stored as a table with the table rows pointing to the physical location of the data |
| Lookup and caching mechanisms | Caching handled by the buffer pool; data lookup handled as queries |
| Recoverability | Write-ahead log and periodic checkpoint |
| Resource reallocation | Alter container assignment and rebalance data placement |
| QoS optimizations | Built-in row-level encryption and compression functions |

for leveraging a DBMS for building a storage virtualization engine. The main contributions of this work are: (1) We propose to correlate the concepts between the storage virtualization engines and the DBMS engines, namely, storing the logical-to-physical mapping information as tables and handling the incoming I/O requests of the application as DBMS queries. (2) Based on this basic paradigm, we employ STORAGEDB with various advanced features, such as recovery, online resource reallocation, storage compression and encryption. All these features are of great importance in practice and can be cheaply and reliably implemented in our STORAGEDB. (3) We have implemented STORAGEDB as a prototype based on a commercial DBMS. While our STORAGEDB has significantly more advanced features than existing virtualization engines, the extensive experimental study shows that the extra cost of the STORAGEDB system performance is relatively small (20% in average from trace driven experiments) compared to those existing systems due to our various optimization strategies. As an added benefit, our STORAGEDB is independent of any specific operating systems or hardware platforms.

## 2. Handling I/O requests using DBMS

In a typical storage virtualization system [8, 13], the I/O requests from the application often refer to the addresses of *logical blocks*. The role of the virtualization engine is to translate these logical blocks to the actual *physical blocks*.

### 2.1. Logical-to-physical Mapping

The data stored on the logical disk as well as on the physical disk can be represented as *block vectors*; a block vector is a set of consecutive blocks. As such, the physical or logical devices can be represented as a collection of block vectors. The physical block vector consists of a set of 512 byte consecutive blocks from the same physical device. The entire physical storage, which consists of several physical devices, can thus be modeled as $r$ physical block vectors as shown below. $p^i_j$ is the $j$th physical block of vector $PV^i$.

$$PV^i = p^i_1, p^i_2, p^i_3, \cdots p^i_{q_1}, (1 \le i \le r)$$

Similarly, the logical disks, namely LUNs, are defined in terms of $n$ logical block vectors as follows. $b^{i'}_j$ is the contents of the $j$th block of logical block vector $BV^{i'}$.

$$BV^{i'} = b^{i'}_1, b^{i'}_2, b^{i'}_3, \cdots b^{i'}_{m_1}, (1 \le i' \le n)$$

In STORAGEDB, the $r$ physical block vectors, the $n$ logical block vectors, and their mapping are implemented as follows. First, we propose to instantiate logical block vectors and physical block vectors using DBMS objects. For each logical block vector $BV^i$, we define a *table* $TBV^i$. The corresponding table schema is $(block.number, block.data)$. The field *block.number* has an integer value and it represents the block number for the data stored in the same row. The field *block.data* is 512-Byte binary. Each physical block vector $PV^j$ is represented by a *container* $CPV^j$. DBMS supports containers residing on files or containers on raw devices. Here raw devices are disks or disk partitions exported through raw IO interface. We deploy the raw device containers to have the DBMS manage the storage subsystem directly. A container consists of pages, the smallest IO unit in DBMS, and each page contains records, whose size equals the row size in a table. Hence, a block in $CPV^j$ can be identified by the page ID and the record ID. Finally, *table space* is the DBMS object keeping track of the mapping between logical block vectors and physical block vectors in STORAGEDB. A table space consists of a set of containers exclusively and each table space is dedicated to one $TBV$ table. After the relationship between logical block vectors and physical block vectors is set up by defining a table space, the one-to-one mapping from one logical block to one physical block is maintained by a table space map. When the logical block vector changes its residential physical block vectors, the table space map is modified to reflect the up-to-date status. To convert LUN $BV^i$ to table $TBV^i$, a meta table is created.

### 2.2. I/O Access Mechanism

Without loss of generality, we assume that the application's read and write requests are in SCSI format [11]. The mapping proposed below can also apply to other IO standards, such as ATA/IDE [2] and Fibre Channel Protocol [4]. A block I/O request is described by 5 parameters: ($\mathcal{T}$, $i$, $s$, $n$, $\mathcal{C}$). $\mathcal{T}$ describes whether the access type is a read or

write; $i$ is the logical block vector number; $s$ is the starting block to be accessed within the logical block vector $BV^i$; $n$ represents the number of consecutive blocks to be accessed; $C$ contains the contents of the blocks to be updated for write access. For read access, $C$ returns the blocks being read.

Figure 1 depicts the IO flow through STORAGEDB. A meta table is first accessed to map the LUN ID into a table ID. The protocol converter takes the table ID and other parameters from the SCSI call as input and maps them to a DBMS call. DBMS calls are initiated in SQL [12]. Table 2 shows a simple example of translating a single block ($n = 1$) I/O to a SQL query. I/O calls with more than one block ($n > 1$) are converted into more complex SQL queries. We use UPDATE statements on table $TBV^i$ for write IOs and SELECT statements for read IOs. The functions $f$ and $f^{-1}$ are included in our SQL statements to adopt various DBMS built-in functions which we will describe later. For now, $f$ and $f^{-1}$ are identity functions.

**Table 2. SQL statements for single block I/Os**

| write: | read: |
|---|---|
| UPDATE $TBV^i$ | SELECT $f^{-1}$(block.data) |
| SET block.data=$f(C)$ | FROM $TBV^i$ |
| WHERE block.number = $s$ | WHERE block.number = $s$ |

The task of the DBMS server is to locate the record with $block.number = s$ in the table $TBV^i$, and then perform the read or write operations on the $block.data$ column which holds the data for the block. Unless the database page containing the record is already in the DBMS buffer, the DBMS would fetch the page from physical storage, pin it in the DBMS buffer, locate the record within the page and read or write the data column of the record.

## 3. Exploiting Built-in DBMS Features for Storage Virtualization

DBMS technologies in general are mature and proven. This section briefly describes how DBMS features can be used for storage virtualization.

### 3.1. Supporting Data Reliability

In STORAGEDB, every IO operation is handled as a database transaction. A database transaction satisfies the ACID properties [10], namely, Atomicity, Consistency, Isolation and Durability. From the storage virtualization perspective, atomicity ensures that either the entire IO is completed, or no changes are made; Durability ensures that after an IO is completed, it should be in the system and never lost; Isolation allows concurrent IOs while still having the results serializable; Consistency ensures that certain rules or constraints are not violated in the storage systems. For instance, when a LUN table is to be deleted, the corresponding mapping rule in the meta table needs to be removed too.

In STORAGEDB, every transaction is logged using the write-ahead log (WAL) technique that enforces *atomicity* and *durability* [10]. WAL requires that the data updates be first written to the log before changing the actual data image. When a system crashes, the recovery process applies the delta log sequentially redo the committed IOs, and applies the delta log in the reverse order to undo the uncommitted IOs. Next, by mapping the block IO transactions to the database transactions, the *isolation* property for those block IO transactions is automatically enforced by the DBMSs. Finally, the *consistency* property is also important for block virtualization engine. For example, when a logical block vector (LUN, or table) is deleted, the corresponding entry in the meta table must also be deleted. This can be achieved using referential integrity. If the table is dropped and the corresponding row in the catalog table is deleted, then the referential integrity cascades such deletes to the meta table. As a final conclusion, STORAGEDB can easily utilize the ACID properties of DBMSs to achieve a reliable storage system.

### 3.2. Online Resource Reallocation

With the changing access patterns associated with the application data, the mapping of data to the storage resources needs to change to support the performance requirements of the data and also provide optimal utilization of the available storage resources. Hence, it is crucial for a storage virtualization engine to provide an *online* data migration function, which moves data from one or one set of physical disk(s) to another without pausing application IOs.

Recall that in STORAGEDB, the logical disks are represented by DBMS tables, and the tables are further associated with containers through table spaces. It is possible to implement online LUN migration, if DBMSs are able to move data across containers. STORAGEDB allows reassignment of containers to a table space. Hence, a data migration function can be implemented by adding new containers to the table space, and then removing old containers from the table space. STORAGEDB will automatically move data from the source disks to the target disks. Note that the data movement path does not involve DBMS buffer activity and runs at the storage level directly. STORAGEDB uses a feedback-based throttling mechanism to control the impact of data migration. This mechanism is in-built in several commercial databases such as DB2.

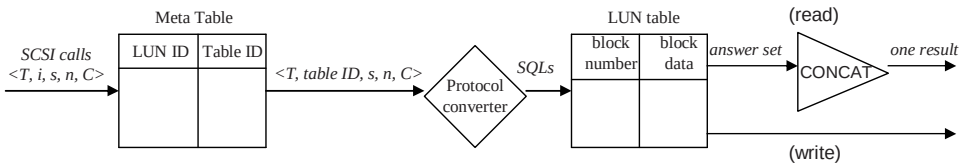


**Figure 1. IO Flow in** STORAGEDB

### 3.3. Block Level Compression and Encryption

Compression trades off CPU for storage efficiency. According to the "80/20" rule, most of the stored data is less often accessed. Hence infrequently accessed storage is an ideal candidate for compression [5, 3]. Compression also saves storage bandwidth and hence improves performance. The implementation of storage compression is achieved by simply adding the *compression* verb in the SQL statement as shown in Table 2. The $f()$ and $f^{-1}()$ functions are respectively `COMPRESS()` and `UNCOMPRESS()`.

With emerging regulations such as HIPAA [7], data security is becoming a legal requirement for storage applications. In STORAGEDB, encryption is achieved by using the verb *encrypt* in the SQL statements (shown in Table 2). The function $f()$ is simply `ENCRYPT()` and $f^{-1}()$ is `DECRYPT()`. STORAGEDB uses the RC2 block cipher with padding encryption algorithm.

In both cases, every row in the LUN table stores compressed/encrypted data, and the data is compressed/encrypted and uncompressed/decrypted on the fly for read and write IOs.

## 4. Optimizations for STORAGEDB

Using a DBMS in a traditional way to serve the purpose of block virtualization incurs large performance overheads. This becomes a major concern because I/O subsystems have real time requirements. Hence, we optimized several critical components to build an embedded DBMS which exhibits much less overheads. STORAGEDB requires extra space to store the metadata. Some of the proposed optimization techniques also reduce space overhead.

**Table Direct Access** Beyond the traditional optimization strategies, we found that STORAGEDB has some unique properties such that special optimizations may be applicable. First, in STORAGEDB, the rows are fixed-size with one integer column, *block.number* and one 512-byte binary column *block.data*. Second, the updates will never move the rows since the keys *block.number* remain unchanged. Based on these two properties, we can lay out the records in such a way that their addresses can be computed by a mathematical formula, which is simply $block.number \times sizeof(row) + start\ offset$. With this approach, record access becomes much less expensive since index access can be completely avoided. This in fact also reduces the space cost for the indices. We call this access method *Table Direct Access* or TDA for short. The DBMS we used, was capable of supporting TDA.

**Static Query Optimization** Also we found, through experiments, that the query optimizer always produces the same access method for all types of queries in our STORAGEDB scenarios. One reason is due to the simple structures of the table as well as the queries. Hence optimizing once and reusing the same optimized plan for subsequent DBMS calls dramatically reduces the optimizer's cost. This can be achieved by exploiting the Static SQL [9] feature in a DBMS.

Our preliminary experiments show that by employing both TDA and Static SQL, the CPU consumption is reduced by more than 60% for the queries in Table 2. Hence our STORAGEDB employs both TDA and Static SQL features. In the rest of this paper, we will consider this system as a basic implementation and explore various strategies to further optimize the performance.

**Optimization of Cursor Processing** An I/O request generally translates to multiple data blocks being accessed. Thus, the DBMS server will produce a query result-set with multiple records and send them to the client. In order for the STORAGEDB client to obtain such results, the Cursor process needs to be invoked for *each* result row to fetch and copy this row to the client host variable.

The idea is simply to reduce the number of records. Since the query results have to remain correct, the reduction of the number of records implies an increase in the size of each record. In practice, the output record length often has some limitation $L_{max}$. Hence, we can reduce the number of records, or the number of Cursor invocations, to $\lceil n/L_{max} \rceil$. In order to return a record that concatenates several table rows on the server side, we need to introduce a special *User Defined Function (UDF)*. UDFs can extend the functionalities of existing database engine. It can run in the DBMS engine space by specifying the fenced mode, and therefore be executed almost as efficient as native built-in functions. Our experimental study shows more than 60% reduction of the Cursor cost by this technique.

While the above technique reduces the cursor invocation times to 1 or $\lceil n/L_{max} \rceil$, further optimization is possible to completely remove such cursor processing. That is, if

the DBMS server can directly communicate with the client, then we can completely avoid Cursor processing at all. This is possible when the DBMS server and client reside on the same host machine, and the data passing is through the IPC shared memory. Therefore, the cursor processing overhead can be completely removed. While IPC shared memory is superior to the cursor approach, the cursor approach is generic for both local and remote server accesses.

**Optimization of Query Processing** The UDF approach described in the previous subsection successfully reduces computational cost as well as latency. However, the embedded DBMS server will still incur a significant per-record computational overhead, i.e., invoking the UDF per block. To further optimize for performance, we re-design the table schema, so that multiple consecutive blocks are stored together within a single record, referred to as *block accumulation*. We omit the analysis on the saving of computational cost in this extended abstract, and defer it to the full paper.

Assume that the number of blocks in a row is $k$ and a IO size is $n$ blocks. We now analyze how to decide the value of $k$. With block accumulation, a tuple in $TBV^i$ now becomes $\langle j,\ b^i_j b^i_{j+1} ... b^i_{j+k-1} \rangle$ by combining $\langle j,\ b^i_j \rangle$, $\langle j+1,\ b^i_{j+1} \rangle, ..., \langle j+k-1,\ b^i_{j+k-1} \rangle$. DBMSs read or write rows in data page units, which can be defined as a multiple of 4KBs. A data page consists of a page header, pointers to the rows in the page, and rows. Rows may not span pages. We call the former two data structures *page metadata*. To maximize data page utilization, we put as many blocks as possible into one page and merge them into one row. Therefore, the value of $k$ is given by the following formula.

$$k = \lfloor \frac{page\ size - page\ metadata\ size}{block\ size} \rfloor \qquad (1)$$

For instance, if the page size is 8K bytes, page metadata is 400 bytes and block size is 512 bytes, then $k$ equals 15.

In summary, the number of rows processed by the UDF function for the block range $[s,\ s+n-1]$ reduces from $n$ to $\lceil n/k \rceil$ or $\lceil n/k \rceil + 1$. Our UDF function was appropriately changed to be aware of block accumulation, and was capable of starting with a block that is not necessarily aligned at the beginning of the column containing the blocks.

### 4.1. Summary of Experimental Results

We have implemented STORAGEDB as a prototype based on a commercial DBMS, IBM DB2 UDB v8. In this section, we evaluate the performance feasibility of using STORAGEDB as a storage virtualization engine. For our experiments, we compared STORAGEDB with the open-source Linux Logical Volume Manager (LVM) [13]. Even though STORAGEDB is functionally superior to LVM, we use the experimental results as a ballpark estimate for the feasibility of this paradigm. We only show a few results and refer to our full paper for testbed details and extensive evaluation results.

We analyze the effectiveness of the improvements made to eliminate CPU overheads in STORAGEDB, namely, 1) O0: Apply Table Direct Access method; 2) O1: O0 plus static SQL; 3) O2: O1 plus concatenation UDF; 4) O3: O2 plus shared memory; 5) O4: O3 plus block accumulation.

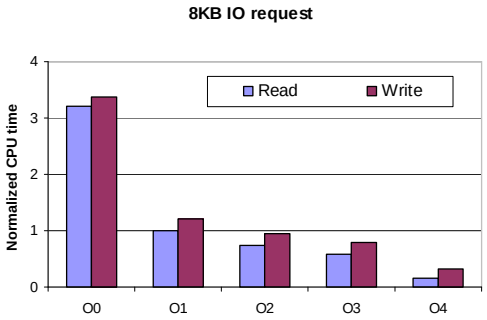


**Figure 2. CPU Time Improvement**

Figure 2 depicts the total CPU time for each case of reads and writes. We normalize the CPU times over the CPU time for O1 read case. The relative scale still remains the same. Compared to O0, we achieve 19 times CPU time speedup for 8KB read IOs, and 10 times CPU time speedup for 8KB write IOs. This establishes the effectiveness of our proposed optimization strategies.

**Table 3. IO Traces**

| Trace | Type | Latency Ratio |
|---|---|---|
| BYU | TPC-C benchmark | 1.21 |
| US2 | File system server | 1.32 |
| WebSearch | Web search engine | 1.02 |
| Financial2 | OLTP application | 1.30 |

To reveal the performance in reality, we drive traces from real-world workloads as listed in Table 3, and we report the latency ratio of STORAGEDB over LVM. The ratio for WebSearch trace shows the smallest performance gap, because WebSearch consists of mostly read IOs. Because STORAGEDB logs changes for write IOs, the overhead of extra seqential log IOs is shown in the results of other traces. On average, the latency of STORAGEDB is 20% more than that of LVM, which is a surprisingly good number, while STORAGEDB provides much more useful functions.

## 5. Conclusion and Future Work

Implementing an OS-independent block virtualization engine using a DBMS is proposed and demonstrated for the first time in this paper. We present our design of a DBMS-based virtualization engine, STORAGEDB, and the corresponding table schema and interface for block virtualization. Techniques, including new User Defined Functions, and block accumulation are developed. System performance, represented by CPU consumption and latency, is improved by over an order of magnitude for the embedded DBMS block virtualization solution. The IO performance from trace-driven experiments of our STORAGEDB solution is within 20% more than LVM [13], an open source block virtualization engine, while it delivers enforced recoverability and concurrency control required by the storage systems. Valuable functionalities, such as online LUN migration, storage compression and encryption, are also implemented easily by exploiting existing DBMS functionalities. In this paper, we address the problem of exploiting external routines provided by DBMS to support storage sub-systems efficiently. To further improve STORAGEDB performance, as our future work, we are exploring the optimization inside the DBMS kernel.

## References


[1] K. Amiri, G. A. Gibson, and R. A. Golding. Highly Concurrent Shared Storage. In *International Conference Distributed Computing Systems*, pages 298–307, 2000.

[2] AT Attachment (ATA) storage interface: `http://www.t13.org`.

[3] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. In *IEEE Transactions on Software Engineering*, pages 1047–1058, 1985.

[4] Fibre Channel Standards: `http://www.t11.org`.

[5] G. Graefe and L. D. Shapiro. Data Compression and Database Performance . In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, April 1991.

[6] R. Grimm, W. Hsieh, and F. Kaashoek. Atomic Recovery Units: Failure Atomicity for Logical Units. In *International Conference Distributed Computing Systems*, pages 26–37, 1996.

[7] HIPAA: National Standards to Protect the Privacy of Personal Health Information `http://www.hhs.gov/ocr/hipaa/finalreg.html`.

[8] IBM Corp. *AIX Logical Volume Manager from A to Z: Introduction and Concepts.* IBM Redbooks, 2000.

[9] IBM Corp. *Application Development Guide: Programming Client Applications*. IBM Redbooks, 2002.

[10] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[11] SCSI block commands(2) working draft 2004: `http://www.t10.org/ftp/t10/drafts/sbc2/sbc2r14.pdf`.

[12] ANSI/ISO SQL92, ISO/IEC 9075:1992(E) Information Technology - Database Languages - SQL, 1992.

[13] D. Teiglan and H. Mauelshagen. Volume Managers in Linux. In *USENIX*, pages 185–197, 2001.

[14] TPC benchmarks: http://www.tpc.org.