

Attribute Storage Design for Object-based Storage Devices

Ananth Devulapalli
Ohio Supercomputer Center
ananth@osc.edu

Dennis Dalessandro
Ohio Supercomputer Center
dennis@osc.edu

Nawab Ali
The Ohio State University
alin@cse.ohio-state.edu

Pete Wyckoff
Ohio Supercomputer Center
pw@osc.edu

Abstract

As storage systems grow larger and more complex, the traditional block-based design of current disks can no longer satisfy workloads that are increasingly metadata intensive. A new approach is offered by object-based storage devices (OSDs). By moving more of the responsibility of storage management onto each OSD, improvements in performance, scalability and manageability are possible.

Since this technology is new, no physical object-based storage device is currently available. In this work we describe a software emulator that enables a storage server to behave as an object-based disk. We focus on the design of the attribute storage, which is used to hold metadata associated with particular data objects. Alternative designs are discussed, and performance results for an SQL implementation are presented.

1. Introduction

With storage systems becoming increasingly more complicated, and storing an ever growing amount of data, the traditional block-based concept of storage has become inadequate. In a time when individual hardware components are becoming more aware of their role in the system, it makes sense to consider similar improvements to storage. The ANSI standard for an object-based interface to storage devices [21] aims to do just this. An object-based storage device (OSD) offers the file system an object-based view of the data. OSDs manage the data layout and keep track of various attributes about data objects on the disk. The concept of an object is a powerful one, allowing for a number of attributes about a particular piece of data to be stored along with the data. Unlike a block-based disk, an OSD is aware of the logical organization of data as defined by its users. By moving functionality that is traditionally the responsibility of the host OS to the disk, it is possible to

improve the overall performance and simplify the management of a storage system.

Since the OSD standard [21] is still relatively new, there are no readily available production hardware disks. To enable research into issues related to using object storage in high-performance file systems, we have created a software OSD emulator. In this work we present our standards-compliant emulator and aim to identify the trade-offs associated with using SQL queries for metadata operations, in particular, for the fast indexing operations supported by OSDs.

An object-based approach to storing data is not new; in fact, many parallel file systems [1, 10, 23] use an object representation of data. However, despite the object abstraction at the file system level, the disks themselves are simple block-based devices. Though our OSD target uses ordinary block-based disks at the lowest level, the interface exported to file systems is an object interface.

The rest of this paper is organized as follows. Section 2 discusses the different storage architectures and the OSD usage environment. Section 3 introduces the design and implementation details of the OSD target emulator. Section 4 evaluates the SQL-based design and raises issues that are important to good metadata performance. Section 5 summarizes related work, and we conclude with ideas for future work in Section 6.

2. Background

The basic feature that sets OSDs apart from traditional block-based disks is an object view of data. An object is defined as an ordered set of bytes that is associated with a unique identifier. Four types of objects are defined in the OSD specification [21]: root, partitions, collections, and user objects. There is always exactly one root object on an OSD, but it can contain multiple partitions. A partition is a way of defining a namespace for collections and user objects. User objects are the entities that actually contain

file data and metadata, while collections are groupings of user objects, used for fast indexing based on attributes.

One of the powerful features of OSD is that each object has a number of attributes associated with it. These attributes can be thought of as the metadata of the object. There are both device-defined and user-defined attributes. Attributes provide an enriched interface, and offer higher semantic control over the data compared to a traditional block-based device. The attributes for each object are organized in pages for identification and reference, and attributes within a page have similar sources or uses. Within each attributes page, attributes are identified by an attribute number. Figure 1 illustrates an example of the user object time stamp attributes page. In the figure, the attribute numbers 0, 1 and 2 represent object creation, attribute access and attribute modification times respectively. The attribute value column shows the corresponding times.

Attribute Page	Number	Value
3	0	01:00:00
3	1	22:00:00
3	2	12:22:22
...

Figure 1. Structure of User Object Time-stamps attributes page.

Another difference between object- and block-based storage is the role that the host file system plays in data storage. In block-based disks, the file system is responsible for telling the disk where to store data. It is up to the file system to know which logical block addresses to read and write; the disk simply does as it is told. With object-based storage, the task of managing data layout is up to the device, not the file system. The file system supplies the OSD with data to store, and the OSD stores it according to its own internal policies, returning an object identifier, and optionally the attributes about that object.

OSDs also offer a comprehensive security model that allows direct device access by multiple clients in an untrusted environment. Previous distributed storage implementations required imposing servers in the data path between clients and storage devices to add security and to offer higher-level services to clients. They also provide other servers to store the metadata of a file system, usually on separate storage devices. With OSDs, it is feasible to store both data and metadata securely and to offer a high level of semantic interaction to clients.

While it will be beneficial to employ OSDs in a local file system configuration, the target of our work is to evaluate how parallel and distributed file systems can take advantage of the new devices. This goes beyond simply delegating the

storage of data containers to devices, and entails designing metadata layouts to improve look-up times and search capabilities for clients. Metadata-heavy workloads are fast becoming the norm [20, 14], and OSDs may offer a workable alternative to what is currently a major bottleneck for distributed file systems.

3. Design

Our OSD target emulator is a SCSI device, and as such, commands and responses are communicated by a SCSI transport protocol. Examples of SCSI transport protocols include fibre channel, SCSI parallel interface, or in our case, iSCSI. Basic SCSI command processing is performed by a generic SCSI layer, and OSD-specific commands are handled by our emulator. We utilize the existing software *tgt* [3] to handle transport and iSCSI command handling. *tgt* is a user-space iSCSI target implementation for block-based devices. In spite of being implemented in user-space, its performance is equivalent to the in-kernel iSCSI target alternatives. Moreover, since *tgt* is user-space software, no kernel modifications are needed.

There are other OSD target emulators available. Despite this, we could not adopt existing codes due to our overall goals of applying OSDs in parallel file systems. Some existing work relies on old or incompatible iSCSI stacks [17, 7], others do not make their source code available [2, 6]. Also, given the rapid rate of change in the specification, none of these alternatives was current with respect to the latest version of the T10 proposed standard [21]. Our goal for the target emulator is to achieve conformity with the OSD specification, mimic the expected behavior of the device as closely as possible and attain reasonable performance.

OSD commands can be classified in the following categories: object manipulation, input/output, attribute manipulation, security, and device management. Our OSD target currently implements all mandatory object manipulation, input/output, and attribute manipulation commands. It also supports collection functions such as QUERY and SET MEMBER ATTRIBUTES. Work on the other commands including security and device management is in progress.

3.1. Data storage

In designing a storage emulator, there are a number of issues concerning how to store data. Essentially, an OSD implements a simplified file system. An OSD must manage data placement and on-disk structures to describe object data (called “inodes” in BSD parlance). Standard techniques such as journaling, logging, log cleaning, group placement, defragmentation, and more apply to this problem and are the responsibility of the OSD. These problems are well studied in the context of local file systems [9, 15, 19]. It is not our goal to apply any of these

existing solutions to the OSD case. Although, there are interesting challenges when objects are generated by striping file systems, as studied in the EBOFS work [22].

Our emulator stores object data in files. The files are provided by an underlying file system, using the VFS interface in Linux. We use `pread` and `pwrite` to move data to and from the disk, relying on the kernel-resident file system and disk scheduler to store the bytes. The rest of this paper concerns itself with the storage of metadata.

3.2. Attribute storage

Attributes hold “metadata” that is associated with the object. In the traditional POSIX sense, metadata includes information such as ownership, creation, access, and modification times, size, and so on. Local OS-resident file systems store this information in a fixed-sized inode that stores the necessary information. Some file systems support the use of “extended attributes” [4, 16] that hold arbitrary metadata created by user applications.

An OSD must store extensible metadata, but also has further requirements. Certain operations generate object lists based on metadata contents. This is a very powerful operation that may revolutionize the way in which we interact with storage. For instance, consider a large computational experiment that is parametrized by multiple variables, such as temperature, pressure, concentration of oxygen, and so on. Or equivalently, in the domain of customized graphics for entertainment, variables would include frame number and camera angle. Regardless of the user application, a frequent question arises, such as “Find all experimental results where the temperature was between 350 and 400 degrees Kelvin, and the oxygen pressure was above 1500 millibars.” On traditional block-based systems, this operation may proceed in one of two manners. First, if the user has carefully encoded the parameters of the experiment in the file names, a search for matching file names may occur. This method has serious drawbacks if the number of variables is more than two or three. A second way to do it is to use a structured file format such as HDF5 [11] or NetCDF [18], which explicitly support named variables. But to search in this space requires opening each file, reading and parsing the header, and evaluating the parameters in each file individually.

OSDs offer a method to transcend existing behavior. A single QUERY operation will provide object identifiers (and attributes) of objects that match the requested bounds, including the examples mentioned above. There is also support to list objects and their attributes, selectively; and to group objects in logical collections to allow for fast indexing over a subset of objects. To implement this QUERY operation, and related list and collection operations, effectively, requires a comprehensive attribute storage model. The following sections describe multiple approaches to this

problem.

File-based implementation: A simplified approach to implementing attributes would involve encoding the attribute page and number into the file name, with the attribute value residing in the file. This is the approach used by previous OSD emulators [7, 2]. This approach is inefficient when handling complex queries. For example, the query operation described above requires that each attribute file be opened, read, and closed to see if the values are within bounds. The overhead of performing this work through the operating system interface is enormous. This approach is slow, inflexible for complex operations, and would limit scaling in the number of stored objects and attributes.

Simple database: The file-based approach lacks an indexing mechanism which can make look-ups and inserts very fast. A simpler database like *gdbm* [12] or *db4* [13] can solve this problem. Such databases use extensible hashing or B-trees to store the data into two-column tables: one for the key which is used for indexing and another for the value. Such a database is sufficient for implementing the basic commands of the OSD specification [21]. But implementing multi-object commands like QUERY and SET MEMBER ATTRIBUTES, or complex commands like LIST with attributes and getting directory pages, increases the complexity of the implementation due to the restriction of a two-column format.

SQL database: The drawbacks of the previous approaches have motivated us to look at SQL databases for attribute organization and manipulation. For our implementation we were interested in rich SQL semantics of the database but without the overheads of inter-process communication or connection setup that traditional client-server SQL databases. Nor would we need a persistent database that lives outside of the context of the OSD emulator. An alternative embedded database, SQLite [5], simplifies these management issues. It is an open-source database and implements most of the SQL92 standard [8]. Moreover the code footprint is about 160 kB making it viable to be implemented on an embedded processor of an actual disk. SQLite supports all the necessary features for expressing attribute manipulation commands in clean and concise SQL statements.

Customized data structure: The optimal solution to attribute handling is a customized data structure designed purely to support OSD attribute structure. Such a data structure would likely outperform a generic SQL-based attribute manipulation, but it is not clear if the payoff from such a scheme will be commensurate with the effort. Further, designing such a data structure requires an intimate understanding of the relationships between the different components of the OSD attributes.

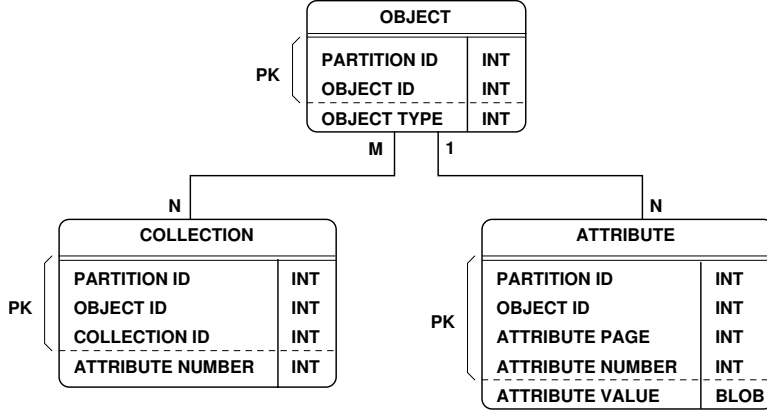


Figure 2. Attribute schema. Primary keys in each table are indicated with “PK.”

Considering the above choices, we traded off some performance for ease of use and selected the SQLite-based design for attribute management. Next we describe the schema used for storing attributes.

3.2.1. Attribute schema

Figure 2 shows the object and attribute schema organization used by our OSD target. The OBJECT table stores the information about the objects currently present on the target. The partition and object identifiers uniquely specify one object. These fields are thus used as the primary key in the OBJECT table. The *object type* column stores one of the four possible types of objects: user object, collection, partition or root.

The ATTRIBUTE table stores information on all defined attributes for all objects. The attributes of each object are uniquely determined by an attribute page and attribute number. Therefore the tuple (Partition ID, Object ID, Attribute Page, Attribute Number) serves as the primary key for this table. It is possible to store attributes of a given object separately in their own table rather than lumping all the attributes of all the objects in a single table, but it increases the complexity of the SQL queries, and common operations such as object creation and removal would result in slow table creation operations in the database. It would be possible to eliminate the OBJECT table by including an Object Type field in the ATTRIBUTE table, and always requiring at least one attribute for every object; however, this would also complicate some otherwise simple queries.

Finally, the COLLECTION table stores the many-to-many relationships between user objects and collections. As discussed in Section 2, collection objects are used for fast indexing of user objects. One collection can have many user objects and one user object can belong to zero or more collections. A user object is made a member of a collection by setting the attribute value in the user object’s Collections Attributes Page to the identifier of the collection

object [21]. Therefore, the tuple (Partition ID, Collection ID, Object ID) uniquely identifies the membership of a user object in a collection. The “Attribute Number” field points back to the attribute in the object’s Collections Attributes Page that was used to make it a member of the collection.

4. Metadata experiments

This section introduces the basic performance of the emulator, then describes four interesting metadata operations that OSDs can perform, along with performance results from our SQL-based emulator. The need to perform these types of queries, and to perform them efficiently, is what motivates the need for a database-like design.

The OSD target described in the paper implements the iSCSI protocol and is capable of communicating with iSCSI initiators. Since the communication overhead might mask interesting interactions within the target, we evaluate the target on its own by injecting OSD commands directly at the target’s OSD command processing layer.

Our experimental platform is a Linux cluster where each node has two AMD Opteron 250 processors, 2 GB of RAM and an 80 GB SATA disk. The cluster runs the Linux operating system, version 2.6.20. The *x*-axis of most plots will be the number of objects, either in the device or in the collection, to show how attribute storage scales with increasing database size. The *x*- and *y*-axes frequently use logarithmic scaling to help visualize the entire range. Experimental sizes were kept small enough to remain within the memory cache of the machine, to examine algorithmic impacts. All plots include error bars showing the standard deviations, but frequently those are so small that they are not visible.

4.1. Primary metadata operations

The first experiment characterizes the scalability of the basic metadata operations. The CREATE operation creates an object, SET ATTRIBUTE sets an attribute on an object,

and GET ATTRIBUTE retrieves an attribute from an object.

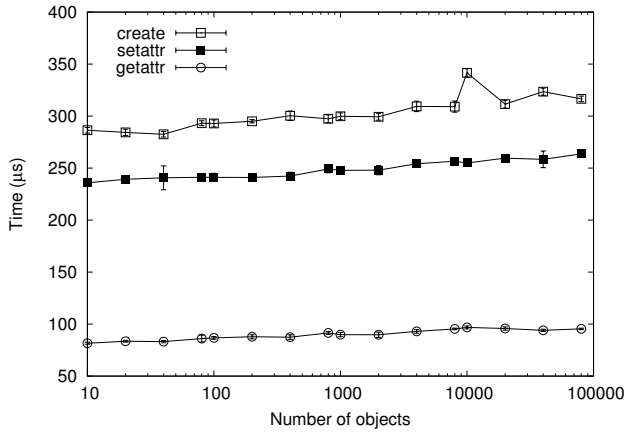


Figure 3. Time to perform CREATE, GET ATTRIBUTE and SET ATTRIBUTE operations.

Figure 3 shows the response times of all three operations as a function of the number of objects in the device. In all cases, a pre-determined number of objects were created with attributes set appropriately. Next, for the CREATE, the time for creation of an additional object was measured, whereas for SET ATTRIBUTE and GET ATTRIBUTE, the time to set and get an attribute on that additional object was measured. The plot shows that latencies of all the operations increase only very slowly as a function of the number of objects, as one would expect from any reasonable database.

The GET ATTRIBUTE operation is the fastest among the three since it simply involves two table look-ups. The first is to determine the type of the object and the other is to look-up the attribute of the object. The SET ATTRIBUTE operation involves table look-ups to test the presence and type of the object. Then a record is inserted into the ATTRIBUTE table, which requires an exclusive lock on the database. The create operation is the slowest since it involves table look-ups for the presence and the type of the object, followed by insertion of object information in OBJECT table and insertion of default attribute information in ATTRIBUTE table.

4.2. List

The LIST operation can be used in two ways. One is to return a list of the partitions in a device. The other is to return a list of objects in a partition. We focus only on the latter, as they are similarly implemented, and listing objects would be expected to be the more common operation.

The SQL statement to perform a LIST operation is:

```
SELECT oid FROM obj
```

```
WHERE pid = PID AND type = USEROBJECT;
```

where “oid” is the object identifier to be selected from a given partition “PID”, constrained to return only objects, not collections, as they share the same namespace [21].

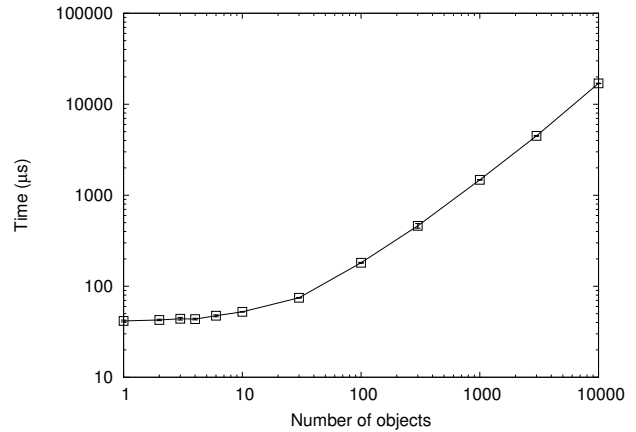


Figure 4. Time to perform a LIST operation.

Figure 4 shows the time to perform a list operation as a function of the number of objects in the partition. In our case, there is only one partition in the device. Beyond an asymptotic minimum, the time taken scales linearly, as each object must be visited and returned by the SQL query.

4.3. List with attributes

A variation to the LIST operation can request a set of attributes for each object returned. The client specifies a list of desired attributes by page and number. The target then generates a list of objects, as in the previous case, but also supplies the requested attribute values for each of the objects it returns.

The SQL statement for this operation is:

```
SELECT obj.oid, attr.page, attr.num, attr.val
FROM obj, attr
WHERE obj.pid = attr.pid AND obj.oid = attr.oid
AND obj.pid = PID AND obj.type = USEROBJECT
AND attr.page = page1 AND attr.num = num1
AND obj.oid >= OID
UNION ALL
SELECT obj.oid, attr.page, attr.num, attr.val
FROM obj, attr
WHERE obj.pid = attr.pid AND obj.oid = attr.oid
AND obj.pid = PID AND obj.type = USEROBJECT
AND attr.page = page2 AND attr.num = num2
AND obj.oid >= OID
ORDER BY obj.oid;
```

where both the OBJECT and ATTRIBUTE tables are consulted to locate all objects. The tuples (page1, num1) and (page2, num2) specify the attribute page and number of the

attributes to be retrieved. The statement constrains the selection to objects belonging to partition PID and of user object type. In case the client does not provide sufficient buffer space for all the results, continuation is supported by the final test on object identifier and by returning sorted results.

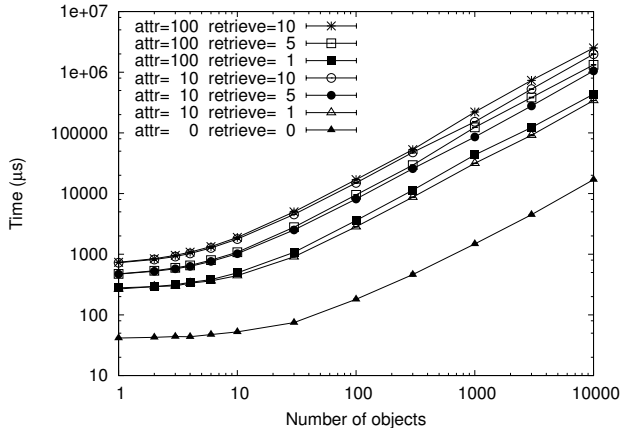


Figure 5. Time to perform a LIST operation, with attributes, for various values of total attributes and retrieved attributes. The bottom curve is the same result for LIST with no attributes as in Figure 4.

Figure 5 shows the time to perform the LIST operation as a function of the number of objects, but this time adds two more parameters: number of attributes per object, and number of attributes retrieved by the client. For example, each object may have 10 attributes associated with it, but the client is only interested in receiving one of those attributes. As expected, the time for the operation is proportional to the number of objects. The time increases linearly with the number of attributes to retrieve, as expected, due to the cost of gathering and marshalling each of the attributes. For any particular value of the number of attributes retrieved, the curves for both 10 and 100 total attributes appear on top of each other. The effect of increasing total number of attributes has only a slight impact on the total time; it increases logarithmically due to the enlarged search space.

4.4. Query

The QUERY operation is a multi-object command that allows selection of objects based on certain criteria, with either a *union* or *intersection* constraint. This command is an example of off-loading computation to the disk. The power of this command is further highlighted in the case of network-attached storage environments, where rather than bringing the data across the network for clients to process,

the computation can be performed directly on the disk, returning only matching entries to the client.

The SQL statement for this operation is:

```
SELECT attr.oid FROM coll, attr
WHERE coll.pid = attr.pid AND coll.oid = attr.oid
AND coll.pid = PID AND coll.cid = CID
AND attr.page = page1 AND attr.num = num1
AND attr.val BETWEEN min1 AND max1
UNION
SELECT attr.oid FROM coll, attr
WHERE coll.pid = attr.pid AND coll.oid = attr.oid
AND coll.pid = PID AND coll.cid = CID
AND attr.page = page2 AND attr.num = num2
AND attr.val BETWEEN min2 AND max2;
```

The QUERY command shown here tries to retrieve objects belonging to the collection CID within the partition PID that match two query criteria specified by the tuples (page1, number1, min1, max1) and (page2, number2, min2, max2). The “min” and “max” values are the constraints on the attribute value (“val”) that serve to restrict the range of selection. While this example shows the union of the two criteria, the intersection form is similar. Since SQLite’s query optimizer is not very sophisticated, we were careful to order the tables in the FROM clause properly, and to order the terms in the WHERE clause to get the desired query plan.

The above translation is optimized for the case when the number of objects within a collection is relatively small compared to the total number of objects within the device. If, however, the sizes are comparable, then a query that directly selects the objects matching the criteria and finally constrains the selection to the collection membership would be more efficient. But this alternate form does not scale well if many objects match the criteria. Based on some function of number of objects, collection size and expected matches, either of the queries can be selected.

There are five variables that impact the performance of QUERY: the number of objects in the database, the number of objects in the collection, the number of attributes per object, the number of query criteria, and the number of objects that match the criteria. The QUERY command is flat with respect to the total number of objects in the device, as one would expect, but is affected by the collection size, match size and number of criteria. We ran the experiment with the INTERSECTION of query criteria for various combinations of number of attributes per object, collection size, and number of query criteria.

In Figure 6, we show the effect of the number of matches on the latency of the QUERY operation. Figure 7 shows the effect of the number of query criteria, and Figure 8 shows the effect of the total number of attributes already present on each object. All three graphs plot latency of the QUERY operation as a function of the number of objects in the col-

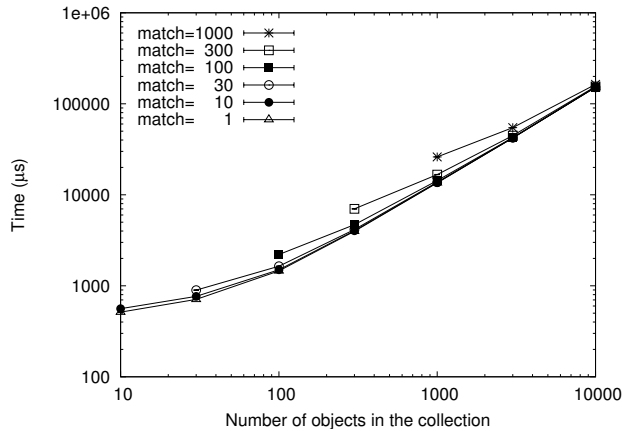


Figure 6. Time to perform a QUERY operation, for various values of the number of matching objects. The number of criteria is fixed at 2, and the number of attributes per object is 10.

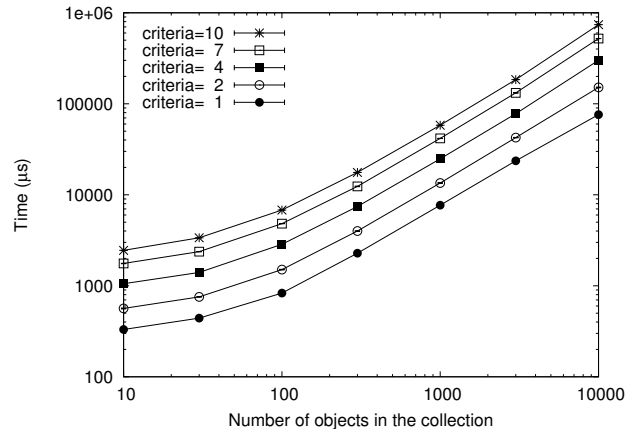


Figure 7. Time to perform a QUERY operation, for various values of the number of selection criteria. The number of matching objects is fixed at 10, and the number of attributes per object is 10.

lection.

All the figures show a linear relationship between the latency and the number of objects within the collection, since the query tests the criteria on each member of the collection. As shown in Figure 6, there is very minimal impact resulting from the number of matches, which only affects the time to assemble the results. Also as the number of number of objects increases, the time becomes dominated by scanning the collection for object membership.

Figure 7 shows a linear relationship between the number of query criteria and the execution time. Each query criterion adds a *select* sub-query to the SQL statement, thereby increasing the time proportional to the number of sub-queries. In the case of a *union* of query criteria, one can collapse the multiple *select* statements to one with the query criteria specified in the *where* clause. But for *intersection* that optimization does not work. We are currently working on proper table organization and SQL design to further optimize all types of QUERY operations.

Finally Figure 8 shows the logarithmic affect of the number of attributes per object on the operation latency. The number of attributes and number of objects within the device have only an indirect effect on the latency by increasing the size of the search space.

4.5. Set member attributes

In many scenarios it is often efficient to combine multiple operations into a single command by amortizing fixed costs like network round-trips and disk seek times. For example, take the case of increasing the version number of all files within a project. One can solve the problem by including all the objects belonging to the project in a collection

and using the SET MEMBER ATTRIBUTES command to set the version number attribute on all the objects at once.

The SQL statement for SET MEMBER ATTRIBUTES is:

```
INSERT OR REPLACE INTO attr
SELECT PID, coll.oid, page1, num1, val1
FROM coll WHERE coll.cid = CID
UNION ALL
SELECT PID, coll.oid, page2, num2, val2
FROM coll WHERE coll.cid = CID;
```

where two attributes are set on each member of the collection CID. The two *select* statements generate values to be used to update the attribute tables. The tuples (page1, number1, val1) and (page2, number2, val2) were specified by the user, along with the PID and CID in question. The *select* statements look up the objects in the collection, and the *insert or replace* statement updates the attributes of those objects.

There are four variables that affect the performance of SET MEMBER ATTRIBUTES: number of objects in the collection, number of attributes being set, total number of objects in the database and total number of existing attributes per object. The last two variables have an indirect impact on performance through the change of the size of the tables. However, as seen from the SQL statement, the size of the collection and the number of attributes to set will have a direct impact on performance.

Figure 9 shows the time to perform a SET MEMBER ATTRIBUTES operation as a function of the number of objects in the collection. Only one attribute is set. As ex-

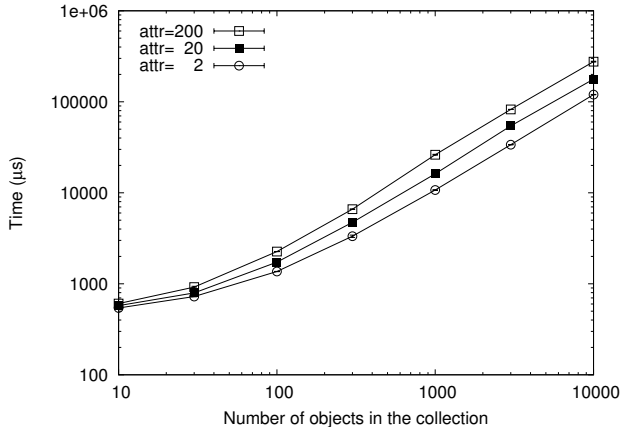


Figure 8. Time to perform a QUERY operation, for various values of the number of attributes per object. The number of matching objects is fixed at 10, and the number of criteria is 2.

pected, the graph shows a linear relationship between the latency and number of objects in the collection. Changing the total number of attributes already existing on each object has only a small effect. The transpose of this data is shown in Figure 10, where a small slope on the curves indicates a slight logarithmic effect. This effect is due to the increased search space caused by the higher number of pre-existing attributes on objects in the collection.

Next we examine the impact of the number of attributes that are set per collection object. Figure 11 shows the results, with the total number of attributes per object fixed at 10. There is a linear relationship between the number of attributes to be set and the execution time. This is shown more clearly in Figure 12.

5. Related work

In addition to our work, various implementations of OSD targets are also available. IBM has developed a prototype of an object-based controller called ObjectStone [6], which uses the iSCSI transport like our emulator. The target is only available in binary form. It uses *gdbm* to store attributes, but the key/value arrangements are unknown.

Intel [7] also has a target OSD emulator for an older version of the protocol. It implements attributes as separate files in a local file system. Work from Du *et al.* [2] builds on this implementation by adding security, but leaves the attribute implementation unchanged.

EBOFS [23] is an object file system that manages the low-level storage of object-based disks. It uses B-trees to perform object look-ups on disk, index collections, and manage block allocation. By directly interacting with the raw block device, EBOFS avoids the local file system in-

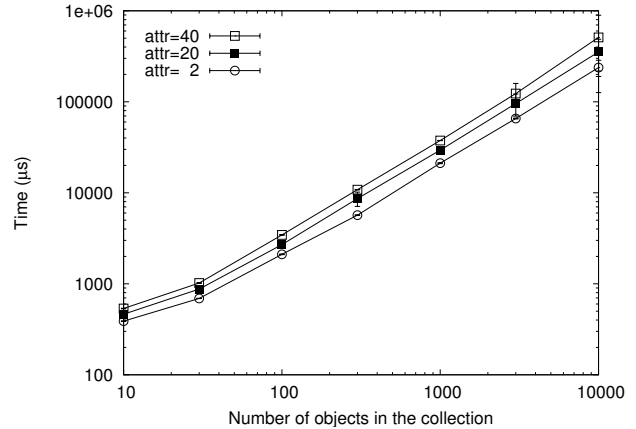


Figure 9. Time to set 1 attribute using SET MEMBER ATTRIBUTES operation, for various values of total number of attributes.

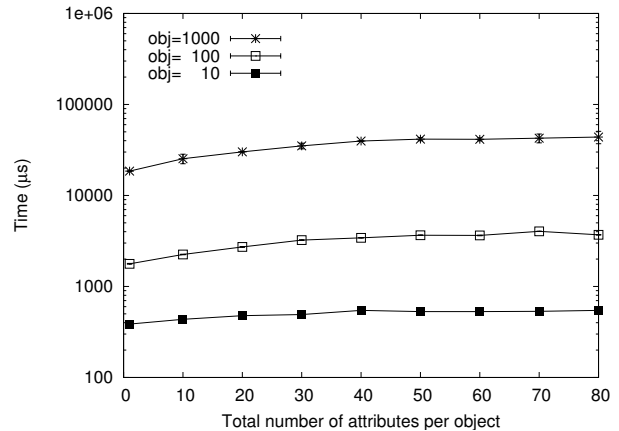


Figure 10. Transpose of data in Figure 9, with more samples, showing slight logarithmic impact of the total number of attributes.

terface. However, EBOFS does not implement attributes.

6. Conclusions and future work

In this paper we have presented a comprehensive design of attribute storage for object-based storage devices. This work describes an implementation for general purpose processors and operating systems using an embedded SQL database, but the concepts and trade-offs are likely to apply to hardware solutions as well. Using a database for attributes rather than storing them as unrelated files permits efficient implementation of fast indexing operations supported by OSDs.

Future work with our OSD target emulator will focus on improving database performance, by adding indexes to

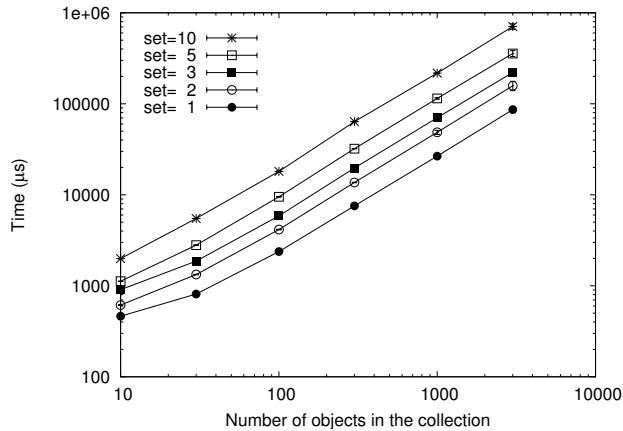


Figure 11. Time to set various numbers of attributes using SET MEMBER ATTRIBUTES operation. The total number of attributes per object is fixed at 10.

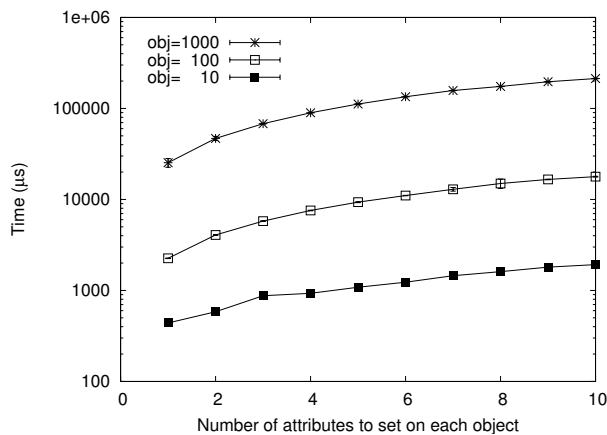


Figure 12. Transpose of data in Figure 11, with more samples, showing pronounced logarithmic impact of the number of attributes to set.

the tables where they will be useful. We will also reconsider the entire design in light of real-world usage models of OSDs in parallel file systems. Concurrent operation of metadata commands to service multiple clients will require a locking strategy for rows or tables so that multiple threads can proceed independently.

References

[1] Cluster File Systems, Inc. Lustre: a scalable high-performance file system. Technical report, Cluster File Systems, Nov. 2002. <http://www.lustre.org/docs/whitepaper.pdf>.

[2] D. Du, D. He, C. Hong, J. Jeong, et al. Experiences in building an object-based storage system based on the OSD T-10 standard. In *Proceedings of MSST'06*, College Park, MD, May 2006.

[3] T. Fujita and M. Christie. tgt: framework for storage target drivers. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[4] A. Grünbacher et al. Linux Extended Attributes and ACLs. <http://acl.bestbits.at/>, 2007.

[5] D. R. Hipp et al. SQLite. <http://www.sqlite.org/>, 2007.

[6] IBM Research. ObjectStone. <http://www.haifa.il.ibm.com/projects/storage/objectstore/objectstone.html>.

[7] Intel Inc. et al. Intel open storage toolkit. <http://sourceforge.net/projects/intel-iscsi/>, 2007.

[8] ISO/IEC. *Database Language SQL*, July 1992.

[9] M. K. M. Margo Seltzer, Gregory Ganger et al. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of USENIX*, June 2000.

[10] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, Pittsburgh, PA, Nov. 2004.

[11] NCSA. HDF5. <http://hdf.ncsa.uiuc.edu/HDF5/>.

[12] P. Nelson et al. gdbm. <http://www.gnu.org/software/gdbm/>, 2007.

[13] Oracle Inc. et al. Oracle Berkeley DB. <http://www.oracle.com/database/berkeley-db/>, 2007.

[14] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, June 2000.

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

[16] SGI Inc. et al. XFS: A high-performance journaling filesystem. <http://oss.sgi.com/projects/xfs/>, 2007.

[17] Sun Inc. et al. Solaris object storage device. <http://www.opensolaris.org/os/project/osd/>, 2007.

[18] Unidata. netCDF. <http://www.unidata.ucar.edu/software/netcdf/>.

[19] N. A. V. Prabhakaran, L. Bairavasundaram et al. IRON file systems. In *Proceedings of SOSP'05*, Oct. 2005.

[20] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the Twentieth IEEE/Eleventh NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004.

[21] R. O. Weber. Information technology—SCSI object-based storage device commands -2 (OSD-2), revision 1. Technical report, INCITS Technical Committee T10/1729-D, Jan. 2007.

[22] S. A. Weil. Leveraging intra-object locality with EBOFS. Technical Report CMPS-290S, UCSC, May 2004.

[23] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of OSDI'06*, pages 307–320, Seattle, WA, Nov. 2006.