Atlas: Baidu's Key-value Storage System for Cloud Data

Chunbo Lai[1], Song Jiang[2], Liqiong Yang[3], Shiding Lin[1], Guangyu Sun[3], Zhenyu Hou[1], Can Cui[1], and Jason Cong[4]

[1]Baidu Inc. , {laichunbo,linshiding,houzhenyu,cuican01}@baidu.com
[2]Wayne State University , sjiang@wayne.edu
[3]Peking University, candiceyoung@pku.edu.cn,gsun@pku.edu.cn
[4]University of California, Los Angeles, cong@cs.ucla.edu

***Abstract*—Users store rapidly increasing amount of data into the cloud. Cloud storage service is often characterized as having a large data set and few deletes. Hosting the service on a conventional system consisting of servers of powerful CPUs and managed by either a key-value (KV) system or a file system is not efficient. First, as demand on storage capacity grows much faster than that on CPU power, existing server configurations can lead to CPU under-utilization and inadequate storage. Second, as data durability is of paramount importance and storage capacity can be limited, a data protection scheme relying on data replication is not space efficient. Third, because of the unique distribution of data object size (mostly a few KBytes), hard disks may suffer from unnecessarily high request rate (when data is stored as KV pairs and need constant re-organization) or too many random writes (when data is stored as relatively small files).**

In Baidu this inefficiency has become an urgent issue as data is uploaded into the storage at an increasingly high rate and both the user population and the system are rapidly expanding. To address this issue, we adopt a customized compact server design based on the ARM processors and replace three-copy replication for data protection with erasure coding to enable low-power and high-density storage. Furthermore, there is a huge number of objects stored in the system, such as those for photos, MP3 music, and documents, but their sizes do not allow efficient operations in the conventional KV systems. To this end we propose an innovative architecture separating metadata and data managements to enable efficient data coding and storage. The resulting production system, called *Atlas*, is a highly scalable, reliable, and cost-effective KV store supporting Baidu's cloud storage service.

I. Introduction

Nowadays almost all Internet-based services, including Google, Amazon, Microsoft, and Baidu, provide cloud storage service. There are also numerous companies, such as Dropbox [1] and Box [2], specialized for service in this rapidly-expanding market. To be attractive and competitive, they often offer large free space and price the service modestly. As an example, in the cloud storage service at Baidu, China's largest Internet search company, there are more than 200 million users and each user has 3TB free space. However, it can be expensive to store a large amount of user data in a data center and to assure an expected service quality. First, the data must be reliably stored with a high availability. Second, though there could be only a small portion of the data set actively accessed in the system, requests for any of its data should be served reasonably fast. Third, more data mean more storage servers. However, unbalanced use of computer resources (high demand on storage space vs. less demand on processor speed) makes conventional servers equipped with power-hungry processors under-utilized and unnecessarily increases the service cost. Below we describe specific challenges we had faced and were motivated to address.

Request Size (Bytes)	Read (%)	Write (%)	#Read / #Write
[0, 4K]	0.6%	1.2%	1.45
(4K, 16K]	0.5%	1.0%	1.41
(16K, 32K]	0.5%	0.8%	1.67
(32K, 64K]	0.8%	1.2%	1.94
(64K, 128K]	1.3%	1.7%	2.08
(128K, 256K]	96.3%	94.1%	2.84
Sum	100.0%	100.0%	2.78

TABLE I: The workload data on a typical day in 2014. The percentages describe distribution of requests across different object size ranges within their respective types (read or write). The last column shows ratios of read and write requests in different object size ranges, and that of total read and write requests, which is 2.78.

A. Challenge on Hardware Efficiency

Our previous cloud storage system ran on servers with Intel x86 processors. Performance study of the system shows that the processors were consistently under-utilized (with 20% or lower utilization rates) even though each server was configured with nine hard disks. We also analyzed its workload's characteristics, including request type and object size distribution. The distribution in a typical day is shown in Table I. A major reason for the processors' low utilization is that the majority of the requests are for large requests (128KB or larger [1]) and most of the data service time is attributed to the storage devices (here hard disks). We also sent a sampled stream of requests from the online system (a workload trace) into a server based on an ARM processor (a 4-core 1.6GHz Cortex A9 processor) and whose storage devices are also nine hard disks of the same type. We found that system's throughput is little changed. Even when we accelerated the trace to its highest possible rate, during most of the runtime the ARM processor utilization remains lower than 50%. This suggests

[1]To prevent service of individual requests from holding excessive amount of resources, including memory and I/O bandwidth, we impose a limit on request sizes, which is 256KB, for our workload.

that CPU cycles are over-supplied in the x86 system and ARM processors are sufficient to support quality cloud storage service, where requests are dominated by requests larger than 100KB, and hard disks, rather than SSDs, are used. Note that installation of more hard disks on a server to increase processor utilization is not an effective solution, as it would allow individual servers to hold an excessive amount of data, compromising data availability and increasing data recovery time upon server failures. In addition, considering that price and energy consumption of ARM processors are usually at most one tenth of the counterparts of the x86 processors, there is a strong incentive to introduce ARM-based servers into cloud storage service.

Fig. 1: Photo of Baidu's ARM-based servers.

In response to this technology trend on workload characteristics and system cost-effectiveness, Baidu deploys its customized ARM-based servers in its production system to support *Baidu Cloud*, its cloud storage service [5]. As illustrated in Figure 1, in each 2U chassis there are six 4-core Cortex A9 processors, each with 4GB memory and four 3TB SATA disks. The storage density is greatly increased (each chassis can hold up to 72TB data). However, the 32-bit ARM processor supports only up to 4GB memory and the entire chassis has at most 24GB memory to support accessing a data set that can be as large as 72TB. This poses a significant challenge on how to organize data on the disks for efficient access.

B. Challenge on Using File System to Manage Cloud Data

We had used Linux Ext3 file system at Baidu to manage user data when x86 servers with lower storage density and more memory were deployed. Our system workload profiling shows that average data object size is only about 128KB. With a very small memory-storage ratio (2GB vs. 16TB) in the ARM-based servers, it is unlikely to accommodate all file system metadata in the memory if the data objects are stored as files. Furthermore, because CDNs (Content Delivery Network) are deployed as a cache of backend storage servers, requests reaching the servers have little locality and the chance of hitting file system metadata cached in the memory is small. This would lead to multiple random disk accesses of metadata and data for serving each request, or a significant I/O performance degradation [10], [15]. To address the issue, Facebook's photo storage, Haystack, reduces metadata size by storing multiple photos in one file [10]. However, because the ARM processor has a very small memory size, even the Haystack approach is not sufficient to keep all metadata in memory.

C. Challenge on Using an LSM Tree to Manage Cloud Data

For efficient metadata access, we adopt the Log-Structured Merge (LSM) tree [17] to manage data objects in the form of key-value (KV) pairs. The LSM tree is introduced to manage a large number of small tuples on the storage, especially on hard disks that are faster with sequential accesses. A data set organized in an LSM tree allows quick inserts as new data items are always appended sequentially at the end of a log file. In addition, updates and deletions are also efficiently supported because the system sequentially records the corresponding operations into the log, rather than conducting in-place data modifications. To facilitate efficient data read and materialize the data modifications, the data in the tree needs to be constantly sorted. With the sorted data, metadata facilitating quick search of requested data can be kept much smaller than that in a file system, including Facebook's Haystack file system, and can easily fit in memory. As an example, if we use Haystack each data object needs about 26 bytes as its metadata, including 16B for key, 4B for data size, 4B for offset, and 2B for hashtable entry. If we assume the average data size is 128KB, the metadata would be of 3.3GB for 16TB of data that can be held in the four 4TB disks installed with one ARM processor. While each processor can have up to 4GB memory, the system software needs about 1GB and some additional space must be reserved for buffer cache. In this case the memory is highly constrained. In contrast, using Baidu's LSM-tree-based data system, we need only around 320MB memory for the metadata. In the LSM-tree, KV pairs are sorted, and each 1MB-block, rather than each KV pair, is associated with a piece of metadata. So there are about 16 million blocks, each with a piece of metadata of about 20 Bytes.

The major cost of using an LSM tree to manage the KV store comes from constant data sorting operations, or data compaction [20]. Usually new data is accumulated into a buffer called *memtable* [11]. When a memtable is filled, it is flushed to the disk with its KV pairs sorted. These sorted KV pairs along with an index and Bloom filters are stored on the storage as an *SSTable* [11]. While the KV pairs within each SSTable have been sorted once it is created, the key ranges of KV pairs from different SSTables can overlap. To approach the status of having a fully sorted list, the system constantly carries out the compactions, in which multiple SSTables are read into the memory, merge-sorted, and written back to the storage as one or multiple fully sorted SSTables.

For a given number of KV pairs written into the store, the amount of data that is accessed for the compactions quantifies the internal workload, which is additional to the write workload from the users. As compaction is the most expensive operation in the LSM-tree-based data management, the amount of internal workload is critical to the entire system's efficiency. The system's efficiency can be quantified by measuring the write amplification, which is defined as the ratio of amount of data accessed as a result of serving users' writes and the amount of data contained in the users' write requests. While the purpose of compaction is to sort KV pairs on the storage, its cost can be unnecessarily high if the cloud data is all managed in a KV store. As shown in Table I, in our workload only less than 2% of objects are smaller than 4KB. As the key has a small fixed size (a couple of bytes), the value size determines the amount of I/O access for sorting a given number of KV pairs. While keys have to be sorted to facilitate quick location of values, values do not need to be sorted together with keys. This is especially the case for the cloud storage of Baidu, in which range query and scan operations are rarely required. Accordingly, we propose to write the values onto a different set of servers and replace the values originally in the KV pairs with references (or pointers) to their respective locations in those servers. In the context of general data management, the keys and values can be considered as metadata and data, respectively. We name the existing LSM tree managing combined metadata and data pairs as fat LSM tree, or *f-LSM* in short, and name the LSM tree with only metadata as slim LSM tree, or *s-LSM* in short.

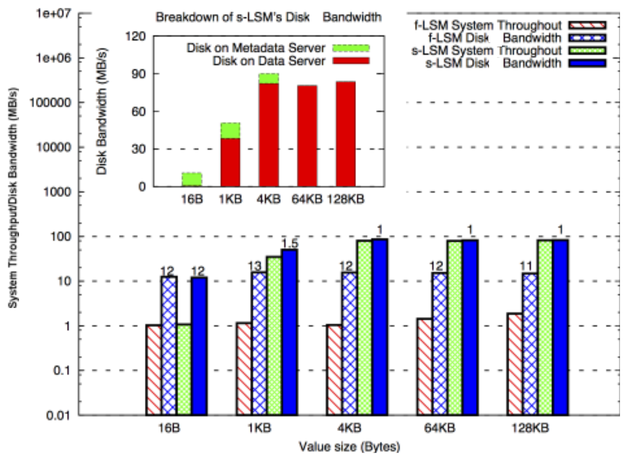


Fig. 2: Comparison of the system throughput observed by the client and the disk bandwidth when random write requests of different value sizes are issued to f-LSM and s-LSM. The Y axis is in the logarithmic scale. As s-LSM uses two disks, one on the metadata server managed by LSM tree and the other on the data server, respective bandwidths are shown in the embedded graph. Write amplifications are shown at the top of the respective disk-bandwidth bars. Keys have 16B size.

To see how write amplification is affected by the proposal, we inserted random KV pairs into LevelDB [6], an open-source LSM-tree-managed KV store [11]. In the experiment, the key size is 16B, and the value size varies from 16B, 1KB, 4KB, 64KB, to 128KB. We used one client to synchronously send KV write requests to a server hosting LevelDB back to back. For s-LSM, a daemon at the server, acting as a metadata server, intercepts the requests and transforms each request into two requests, one containing the value sent to a data server, which sequentially writes the value to its disk, and the other one containing the key and the value's location on the data server sent to the LevelDB on the same local (metadata) server. The LevelDB requests have fixed size (32B). The servers have the same configuration (each with one Intel Xeon L5410 8-core CPU and one Western Digital WDC WD5000AAKS-00V1A0 hard disk) and are connected with the 1Gbps network. Figure 2 shows the system write throughput observed by the client, the disk bandwidth (one disk in f-LSM and two disks in s-LSM), as well as the write amplification calculated as a ratio of the bandwidth and the throughput. As shown, for f-LSM the amplification is mostly in the range between 10 and 15. This suggests that more than 90% of disk bandwidth is spent on internal data re-organization. Only workloads with small values are metadata-intensive, and allocation of a large share of disk bandwidth for sorting key is justified. By removing data out of the metadata sorting operations and redirecting it to the data server, s-LSM essentially allows only keys to be involved in the compaction. With 1KB values, the amplification is only 1.4. With values of 4KB or larger, the amplification approaches 1 as the performance bottleneck moves to the data server and the LSM tree is lightly loaded. As data objects in Baidu's cloud storage is sufficiently large, the proposal of the slim LSM tree architecture takes LSM-tree's advantage of quick data updates but avoids its weakness of high write amplification.

Note that for each key-value pair in our system, the value is user's data object and the key is a signature of the object. We use SHA-1 to generate a 128-bit hash value of the object's data as the signature. This key naming method allows us to efficiently identify duplicates in de-duplication operations. However, it also makes the range search supported in an LSM-tree-based store not useful. Fortunately, because each request involves a sufficiently large amount of data, individually accessing KV pairs does not pose an efficiency issue in the system. In addition, the storage separation of keys from values has an implication on garbage collection. Garbage is produced by update and deletion requests, and needs to be removed. In an f-LSM system, the garbage, or the invalidated KV pairs, is on-line collected in compaction operations. However, in our s-LSM system, only metadata, which contains keys, is on-line collected. Data is off-line collected in a lazy and batched manner for efficiency (see Section II-E for more detail).

D. Design goals and Features

In this paper, we describe the design and implementation of the Baidu's cloud storage, called *Atlas*. In the design, there are several goals to achieve.

- **High availability and durability** This requires redundancy in almost all system components and data storage. The system availability and data safety should be minimally affected by system failures.
- **High write throughput** Because of strong requirement on data safety during the write operations, write amplification is unavoidable. Efforts must be taken to minimize random disk access and the amplification to achieve high write throughput.
- **Strong consistency** As a storage system, Atlas only needs to guarantee that any read request is always served with the latest written data for strong consistency.
- **Fault tolerance** The system must monitor all its functionalities and self-repair any detected system failures and data losses in all of its subsystems.
- **Low cost** Resources, including processors, disks, network, and energy consumption, should be economically provisioned to accommodate workload needs and

to save system deployment and operation costs. This goal is of particular importance as huge volume of data is expected to be uploaded into the system and the system must be able to scale to a very large size with acceptable cost.

To ensure that these goals are successfully achieved, Atlas has provided a number of features in its design and implementation.

- Atlas represents a hardware and software co-design to achieve a high resource utilization with customized low-power servers and efficient data management.
- Taking into account sizes of data objects in the cloud storage, Atlas chooses to use optimized LSM tree to store them as KV pairs, where data (the values) and metadata (the keys) are separated to minimize system I/O workload.
- For the assurance of data safety, Atlas adopts a cost-effective approach where metadata and data are protected differently. As data is moved out of LSM-tree managed KV storage system, it can be reorganized and protected with the Reed-Solomon code [3]. With only 50% redundancy Atlas achieves a data safety as strong as that with the three-replica method. As metadata is small and is still individually managed by a KV system, it is protected by storing three replicas. However, because it has small size, the space overhead due to the high redundancy (200%) is not a concern. In addition, having three replicas allows quick recovery of service after metadata loss.

In the next section we describe the design and implementation of Atlas. The evaluation of the system is described in Section III. Section IV describes additional related works, and Section V concludes.

II. The Design of Atlas

The Atlas cloud storage is designed as a key-value (KV) store to accommodate and leverage its workload's characteristics for achieving three goals. First, because values are much larger than keys in the system, Atlas provides efficient LSM-tree-based data management through separating data (the values) and metadata (the keys and value's location) in their storage and processing. Second, because data volume is much larger than metadata volume and most of the data in the cloud storage is rarely requested, Atlas adopts a hybrid data protection scheme in which metadata is protected by three-copy replication and data is protected by Reed-Solomon coding. The design provides both high space efficiency and strong data reliability. Third, to handle the unbalanced use of server resources between metadata service and data service (the former is CPU/Memory/network intensive while the latter is disk I/O intensive), Atlas co-locates the two services on the same set of servers to complement each other's resource demand for the highest resource utilization.

A. The Architecture of Atlas

Atlas has two major subsystems. One is PIS (Patch and Index System) for managing metadata and preparing data for their storage. The other one is RBS (RAID-like Block System) for data storage. The architecture of Atlas is illustrated in Figure 3. As a general-purpose KV store, Atlas provides a concise API interface to its clients (applications), as shown in Table II. In the commands, key is a 128-bit GUID (Globally Unique Identifier) and value is considered a character string up to 256KB. Users can produce keys by computing SHA-1 signatures of their corresponding values. As we have mentioned, we cap the value size at 256KB to limit amount of resource needed for serving one request.

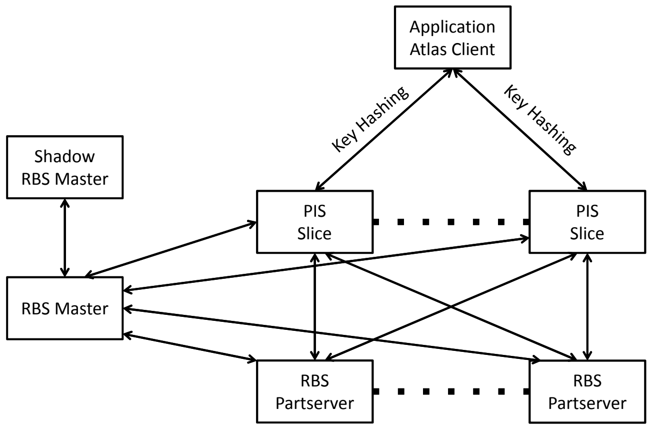


Fig. 3: Architecture of Atlas

Command	Format
Read	Get (UINT128 key, BYTE* value)
Write	Put (UINT128 key, BYTE *value)
Delete	Del (UINT128 key)

TABLE II: Atlas's interface to applications

Rather than relying on a centralized metadata management for locating KV pairs, Atlas distributes its metadata service into a number of PIS slices, which will be described in Section II.B. Facilitated with Atlas's client-side library, an application uses a hash function to determine a PIS slice to receive its request. Choosing a hash function, such as $MD5(key)\ module\ n$ where n is the number of PIS slices, can ensure that keys are uniformly distributed among all slices. Distribution of PIS into many slices avoids a potential performance bottleneck associated with the metadata service, which can be heavily loaded when a large number of requests for relatively small KV pairs arrive. Note that the PIS subsystem does have its metadata server (not shown in Figure 3) for slice management, including mapping slices to physical servers, migrating slices, and recovering slices after failures. As clients cache the mapping between slices and physical servers, in most cases they do not need to consult the PIS master.

Being a metadata service provider, PIS retrieves a collection of values from KV pairs it receives from clients and sends them to the RBS subsystem for storage. Note that PIS does not send individual values for storage. Instead, it accumulates them into a patch until a block of 64MB is formed. This large block size facilitates convenient data coding and efficient data storage. This 64MB block is evenly partitioned into eight parts; each part is stored on a different RBS server, called **partserver** as shown in Figure 3. RBS uses a large access unit, or 64MB block, at the interface for data writing and deletion at each partserver. Specifically, every write or deletion operation on RBS must be of 64MB, although reads are served in the actual number of requested bytes. This asymmetry between write/deletion and read units keeps the metadata describing data locations on individual partservers small and keeps garbage collection and data recovery efficient. Section II.D will discuss this further. Furthermore, using a large write unit enables cross-block coding to generate parity blocks for data recovery.

As shown in Figure 3, the RBS subsystem has a master server, which is a centralized server maintaining metadata about on which partservers a block is stored. RBS serves write, read, and deletion requests received via the PIS slices. However, partservers are not directly involved in the deletion operations. Instead, deletions are conducted via garbage collections at individual partservers in a lazy and batched manner for efficiency. Section II.E will explain this further. In the next two subsections, we will describe the PIS and RBS subsystems in the context of serving write, read, and deletion requests.

B. The PIS subsystem

The PIS subsystem provides metadata service for the entire system. In addition to data indexing, PIS also conducts data replication and accumulation to allow the data to be reliably and efficiently stored into the RBS subsystem. The structure of a PIS slice is illustrated in Figure 4. Each PIS slice has three units that are almost identical in terms of their constituent components and functionalities except that one of them (the primary unit) can serve write/deletion requests and the other two units (the secondary units) are used to provide redundancy preventing loss of medadata/data and service.

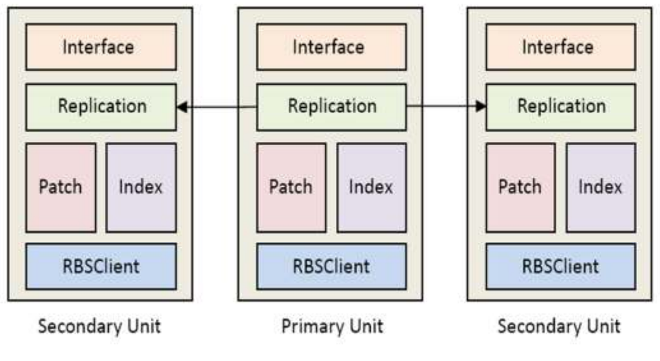


Fig. 4: Structure of a PIS slice

As shown in the figure, requests received by Atlas will first be processed by PIS's replication module. When the module on the primary unit receives a KV write request, it assigns a 64-bit identifier (ID) unique in the slice to the data contained in the request. It then sends the data along with its ID to the replication modules in the secondary units in the same slice. The patch modules in the slice are responsible for saving the data into their respective patch files on their local disks for data durability. When the replication module in a secondary unit receives the data, it asks the patch module to append the data into a log containing data already in the patch. When this is done, the module acknowledges the replication module in the primary unit with the data's ID. When the replication module in the primary unit receives such acknowledges from at least one secondary unit, it writes the data to the patch at the primary unit and acknowledges the requestor. The latency of a write request is not as long as it seems to be, even though data has to be written into local files (patches). The patch modules do not use synchronous writes to the disks. A replication module can simply leave dirty data in the memory buffer before it sends its acknowledgement. This does not compromise data durability as we make sure that three units of a slice are distributed on different server racks. Because dirty data will be flushed onto the disks within a few seconds and the servers are always protected by battery backup powers, it is unlikely that all three copies of data in memory get lost during this short time window.

The patch module keeps track of offset and length of each data added into the current patch. When the patch grows to 64MB (padding data is added if the patch is not exactly 64MB), or a block is formed, a new patch is set up to receive and accumulate incoming data. The patch module at the primary unit passes the 64MB block to the RBSClient module, which then calls the RBS write API to write the block to a number of RBS partservers. After a successful writing, the RBS function will return a GUID for the block. As the patch module has recorded the offset and length for each data in the block, these two metadata items together with block ID become the value to form a KV pair, whose key is just the key of the write request. In other words, the value in the original KV pair is replaced with [*block_id(int64), offset(int64), length(int64)*]. The KV pairs for all the data in the block are sent to the two secondary units via their respective replication modules. At every unit of the slice, the KV pairs are written into its index module, which is managed by Baidu's log-structured-merge(LSM) tree, or a KV store similar to Google's LevelDB. At this time the data in the block has been stored in the Atlas system and its corresponding patch can be removed at each unit. When the secondary units complete the operations, they need to send acknowledges to the primary unit.

C. The RBS subsystem

The RBS subsystem is a storage system that provides efficient and reliable data write, read, and deletion operations. Its API is summarized in Table III. As mentioned, it requires write and deletion in the unit of 64MB block, but allows read at any offset and of any length in the block.

For a request of writing a 64MB block, RBS, specifically the RBSClient module in a PIS unit, evenly partitions the block into eight 8MB parts. It then applies the Reed-Solomon coding over the eight parts to generate four 8MB parity parts so that each block has 12 parts. Because ARM processors do not support SSE instructions, we use a bit matrix to implement the coding algorithm. According to properties of the Reed-Solomon coding algorithm, a block cannot be recovered and

Command	Format
Write	Write (UINT64* block_id, BYTE *data)
Read	Read (UINT64 block_id, UINT32 offset, UINT32 length, BYTE* data)
Deletion	Delete (UINT64 block_id)

TABLE III: RBS's interface

becomes unavailable only after five or more of the block's 12 parts are lost. Because the 12 parts are guaranteed to be distributed on 12 different servers, only after five servers are lost together can RBS possibly lose data. In comparison to the approach of GFS [13] in which each 64MB chunk has three replicas, RBS provides a comparably strong data durability and availability. However, RBS requires only 50% more storage space for its redundancy while GFS needs 200% more space. Admittedly each piece of data on RBS has only one copy. When a set of data on a partserver becomes very hot, the server can become a bottleneck. However, in practice this is not an issue. First, caching systems or CDNs are deployed before Atlas and any hot accesses would be absorbed before they reach the storage. Second, in RBS data in each block are striped over eight partservers and accessed in parallel. Third, Atlas is used to support cloud storage whose workload usually is not very intensive.

As mentioned, RBS employs a global metadata service. When an RBSClient needs to write a block into RBS, it sends a request to the RBS master for a globally unique 64bit block ID (block_ID) and 15 server IP addresses. The least significant four bits of the block ID are always 0. Accordingly the 12 parts of a block (eight data parts and four parity parts), each with an in-block part ID (part_ID) from 0 to 11, can easily obtain their respective globally unique part ID as $[block\_ID|part\_ID]$. In this way, block ID and part ID can be quickly obtained from the global part ID without resorting to looking up an index table. The server IPs returned by the master are guaranteed to be on different servers. It is also guaranteed that at most two of the IPs are from the same rack, so that losing any two racks does not lead to loss of any blocks. In the meantime, the master tries to uniformly distribute parts across the partservers.

To store a block RBSClient needs only 12 partserver IPs. However, the master provides three extra IPs, so that RBS does not need to communicate with the master for additional IPs should some servers fail to store parts. The RBSClient then writes the block's 12 parts to 12 partservers in parallel. For failed writes, it uses backup IPs to re-try until all of them are exhausted and it has to request more IPs for additional re-trials. When the RBSClient successfully stores all 12 parts to partservers, it submits the block ID and the list of partserver IPs where the block's parts have been stored to the RBS master. On the master the metadata is managed as KV pairs in an LSM-tree manner, where key is the block ID and the value is its corresponding IP list. This is called **block table**. The master also maintains the partserver table, which is a reversed block table consisting of the KV pairs. The key is partserver IP, and value is the list of global part IDs representing all parts stored on the partserver. Maintaining the partserver table facilitates efficient garbage collection at partservers, which will be described in section II.E. The master is accompanied by a shadow metadata server, which syncs with the master. When the master is out of service, the shadow master kicks in to assume its role.

The RBS master periodically issues heartbeat messages to all partservers to detect any loss of servers. If a server is found to be lost, a recovery operation is initiated. The master looks up the partserver table to identify all the lost parts on the partserver. It then updates its block table by setting all parts on the lost server as invalid so that a read request for data in any of the lost parts would trigger a reconstruction of the data, including retrieval of other eight parts belonging to the same corresponding block. The recovery of the lost parts entails generation of repair tasks (each for one 8MB part), dispatching the tasks to selected partservers, and updating block table and partserver table accordingly in the master. The tasks are evenly distributed over all partservers to balance the recovery load. To keep the recovery load from disruptively impacting service of normal requests, Atlas ensures that no servers would receive more than one such task per second. Therefore, the more servers in the cluster, the faster the recovery becomes.

D. Serving Read Requests in Atlas

When the key of a KV pair in a read request is hashed into a slice, one of the slice's units will receive the request. The request is first processed by the unit's patch module. If the key matches one in the patch, the corresponding value is immediately returned to the requester. Otherwise, the key is processed by the unit's index module, a KV store, to see if there is a match. A mismatch means that there is not such a data item associated with the key not only in this PIS slice but also in the entire system, because keys are distributed across the slices by consistently using the same hash function. Otherwise, the KV store returns the value associated with the key. The value includes the ID of the block that contains the requested data, and offset and length of the data in the block. With this information, the read function in the RBS API is called in the RBSClient module to ask RBS to retrieve the data.

The above read function first needs to know the partservers' IPs indicating locations of the block's parts. This knowledge can be cached in the PIS slices to alleviate load on the master. If it is not found in the cache, the slice retrieves it from the master. Having the offset and length information, it then calculates which part(s) need to be retrieved and sends read request(s) to the corresponding partserver(s). If all the required data is retrieved, it is returned to the requester. Otherwise, the function selects eight out of available parts of the block to read and decode to recover the requested data. If this also fails, the function asks the master for an up-to-date IP list in cases that local cached IP list is out of date or the master has updated the IP list since the last retrieval. This trial continues until data is successfully read or a a threshold number of trials is reached.

E. Serving Deletion Requests and Garbage Collection in Atlas

The KV pairs stored in Atlas are immutable. They can be deleted and overwritten by inserting pairs of the same keys, but cannot be modified in place. Accordingly, blocks in Atlas are also immutable, and the space held by deleted KV pairs can only be reclaimed in a garbage collection operation.

When a PIS's primary unit receives a deletion request, its replication module forwards the request to the replication modules in the two secondary units of the same slice. In all three units the request is processed by their respective patch and index modules. If the key is in the current patch, it is removed from the patch so that future read requests cannot see the KV pair and the pair will not be written to the RBS. Otherwise, in the index module the deletion operation is simply logged in the KV store and waits for future data merge operation to remove metadata about the key. When the primary unit receives acknowledgements from at least one secondary unit about their completion of respective deletion operations, it informs the requester of the completion. Since this time the KV pair is not visible to applications, though it may still reside in the RBS storage system.

The space held by the deleted data is reclaimed periodically through garbage collection. This process is expensive and needs support from an off-line system. First, two types of metadata information are fed into an off-line MapReduce system. One is the KV store managed by the index module in every PIS slice. The other is information about all block IDs and their respective creation times recorded in the block table in the RBS master server. On the MapReduce system, the metadata from PIS in the form of [*key:block_id, offset, length*] is transformed into the form of [*block_ID:offset, length*] showing what blocks and what segments in them hold valid data. This metadata is then computed against metadata from the RBS master in the form of [*block_ID:create_time*] to answer a question for any block whose creation time is earlier than a threshold (such as one week ago): is the ratio of valid data in the block smaller than a threshold (usually set at 80%)? A positive answer suggests an old and garbage-substantial block. For such a block, Atlas reads its valid data items and writes them back to the system in a controlled rate. Because all metadata is managed in the logged structure tree, these writes will invalidate the data items in the original block. When all of data items are invalidated, the block can be removed.

To remove a block, Atlas removes any metadata about the block in the RBS master, including those in the block table and in the partserver table. At this time partservers storing the block's parts are not yet aware of the fact that space used by the parts can be reclaimed. A partserver periodically asks the RBS master for the list of its valid parts recorded in the master's partserver table. By comparing the list with the parts it actually stores, the partserver identifies invalid parts and removes them.

Atlas is designed to support Baidu's cloud storage service. Our experience shows that deletion operations hold a very small portion of its entire workload. Less than 0.1% of data in the system is deleted in a day. The garbage collection operations can be performed infrequently (once every week or longer) to maintain a high space utilization (usually well above 80%).

F. Summary of Atlas's Design Features

Atlas is a KV storage system designed for workloads whose data sizes are between those friendly to conventional KV stores such as LevelDB (a few kilobytes or smaller) and those friendly to most file systems (a few megabytes or larger). Furthermore, as a cloud storage system, it stores a huge amount of user data (currently more than 200PB) and the data set is relatively stable. It grows at a rate of around 3% each day and has a relatively low deletion rate. A number of design considerations have been carefully made to accommodate the system's usage.

First, from the perspective of Atlas's clients, Atlas does not separate metadata service and data access. All data movements pass through PIS, rather than directly between the clients and RBS. Because the data is not large and PIS has been highly distributed, this design choice has minimal performance impact but can substantially ease the system design and management. In the meantime, metadata service (in the RBS master) and the data service (in the RBS partservers) are separated when PIS accesses blocks or parts from RBS. This helps to improve system efficiency as the data (blocks/parts) is large and the metadata server (the RBS master) is centralized.

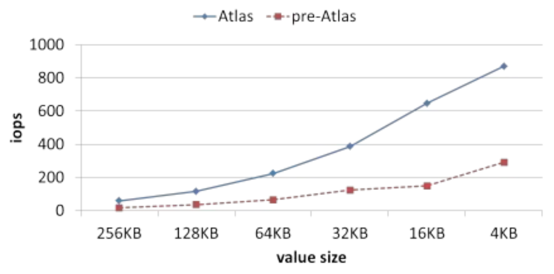
Second, PIS removes values from KV pairs and manages them in an LSM-based KV store, whose efficiency is greatly improved by involving much less data in the system's compaction operation. Value of a KV pair in the index module is replaced by the block ID and in-block offset. This is actually the logical location of the value, and needs an indirection provided by the RBS master to obtain its physical location, such as partserver IP(s). This design enables flexible data managements in RBS, including efficient data migration for load balance, garbage collection, and part recovery.

Third, Atlas treats all PIS slice units and RBS partservers as virtual servers that can be co-located and migrated if needed. They have the form of processes, rather than virtual machines, so they can be very light and each physical server can accommodate a large number of them. On each physical server currently there can be a few hundred units and usually one partserver. Deploying a large number of units allows efficient accommodation of new servers during system expansion, as existing units can simply migrate to the new servers without any change of hash function and data rehashing. Also because partservers need large storage space and slice units demand more CPU cycles for processing metadata, this co-location helps to balance the usage of a server's resources.

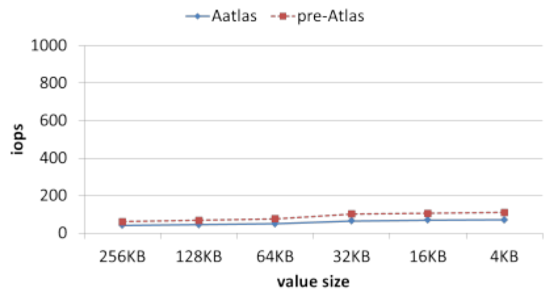
Fourth, because Atlas is a KV storage system, the consistency issue is less challenging. A request to Atlas is served by a unique slice, and a KV pair has only one copy stored in the RBS subsystem. In addition, write and deletion requests can only be served by a primary unit. So Atlas provides strong consistency if it routes read requests only to primary units. If secondary units can also serve read requests, a read request may see out-of-date data because of delayed data update during data replication among units in a slice. In Atlas write and deletion are atomic, as all write/deletion requests are serialized (each assigned a unique ID), and all blocks/parts on RBS are immutable.

III. Evaluation

In this section, we first evaluate the design of Atlas on small-scale clusters with micro-benchmarks, including comparison with an Atlas's predecessor system that does not take out data from KV pairs to store in a different storage subsystem (RBS), as well as Atlas's performance on the customized ARM

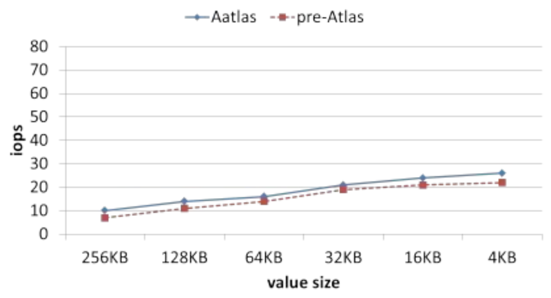


(a) Write throughput (100% write)

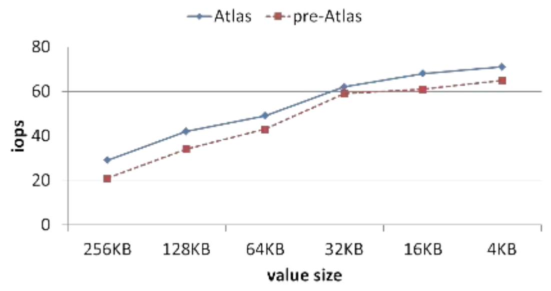


(b) Read throughput (100% read)

Fig. 5: Throughput (requests per second) of random accesses with either write or read requests of various value sizes.



(a) Write throughput (mixed workload)



(b) Read throughput (mixed workload)

Fig. 6: Throughput (requests per second) of random accesses. The requests are mixed reads and writes with a 3:1 ratio.

servers. We then present Atlas's performance behaviors on our production cloud storage system.

A. Comparison with Atlas's Predecessor on Conventional Platform

Before the deployment of the Atlas system, Baidu used its own developed LSM-based KV store to manage the cloud storage. The prior system, which we refer to as *pre-Atlas*, also has its PIS subsystem whose structure and functionalities are similar to their counterparts in Atlas. For example, pre-Atlas's PIS consists of multiple slices; each slice covers a distinct key space for load distribution and consists of three units to provide data redundancy for their durability. The difference from Atlas is the storage of KV pairs. Rather than forming patches and employing the RBS subsystem to store the values, each of a slice's three units simply stores its received KV pairs into its KV store managed by a system similar to LevelDB. Because pre-Atlas runs on servers with x86 processors, in this experiment we compare Atlas and its predecessor on an x86 cluster to observe how Atlas's design helps with the system's performance. The experimental cluster consists of 12 servers, each with two 4-core 2.4GHz E5620 processors and a 2.7TB hard disk, connected with 10Gbps network.

In the experiment we use five client machines to send requests to the cluster. We increase the load on the cluster by increasing the number of threads at each client until the cluster is saturated (when 5 percentile latency is over 100ms). The threads keep sending synchronous requests to the cluster back to back.

Our measurements show that when the system is saturated for both pre-Atlas and Atlas the average disk utilization (the percentage of disk busy time) is over 95%, and the average CPU utilization (the percentage of CPU busy time) is less than 20%. In both systems for write request a server's disk would be accessed with more data than that received (or sent) through the server's network (the NIC), as all received data is written to the disk and the disk is also involved in the data compaction operations. Accordingly the system's throughput is limited by the use of the disk, on which pre-Atlas and Atlas have different management strategies.

Figures 5a and 5b show the throughput of the two systems on the same cluster serving either write or read requests whose keys are randomly selected. As shown in Figure 5a, Atlas consistently has a write throughput around three times as high as that of pre-Atlas over the wide range of value size (from 4KB to 256KB). This result is due to Atlas's design choice of separating metadata and data. As all of the value sizes are much larger than key size, Atlas minimizes the impact of LSM-tree compactions on the write amplification, leading to consistently higher throughput. However, compared to performance difference between f-LSM and s-LSM shown in Figure 2, the improvement of Atlas over pre-Atlas is smaller. In this experiment, each server co-hosts multiple Atlas's service components competing for the disk bandwidth. In addition to PIS's LSM-tree for managing the index in each slice unit and RBS for storing block parts, the disk also supports the patch function for temporarily storing data on the disks. All these operations make disk bandwidth available for storing data lower, limiting Atlas's potential improvement on throughput. As shown in Figure 5b, Atlas's read throughput is around 30% lower than its counterpart in pre-Atlas for all read workload. Atlas needs two disk operations, one for the index in PIS and the other for data in RBS, assuming they are not cached.

However, in our real workloads, the ratio of read and write request on the system is around 3: 1, which is smaller than that from users' perspective because many read requests have been served in the CDNs. Figures 6a and 6b show the write and read throughput, respectively, with this ratio. Atlas achieves higher throughput for both write and read than pre-Atlas as it reduces the load on the disks. From the experiments with these two kinds of workloads, we can see that Atlas's performance advantage is mainly its efficient support of write requests. When the system experiences bursty write requests, Atlas is more likely to maintain a stable system with consistently high quality service.

B. Atlas Performance on Customized ARM-based Platform

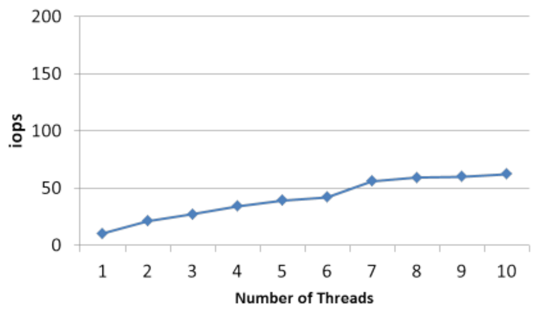
Having shown that Atlas can make more disk bandwidth available for serving users' requests and that x86 processors are significantly under-utilized, we run Atlas on a cluster of customized ARM servers as described in Section I. Specifically, this experiment employs a cluster of 12 ARM servers, each hosting PIS slices and RBS's partservers. Each ARM server has a 4-core 1.6GHz Marvell PJ4Bv7 processor, 4GB of memory, four 3TB 7200 RPM SATA disks, and a 1Gbps full-duplex Ethernet adapter. One RBS master node runs on an x86 server configured with 16-core 3GHz AMD Opteron processors, 32GB of memory, sixteen 2.7TB 7200 RPM SATA disks, and 1Gbps full-duplex Ethernet adapter. We use six x86 machines as clients to send requests to the Atlas cluster. As in the previous experiments, each client thread keeps sending synchronous requests. Therefore the number of threads represents load, or request intensity, on the system. All requests in the experiments are for KV pairs whose value is 256KB, the representative size in the real workload, if not specified otherwise.

1) System Throughput and Request Latency: We first send only write requests to the system. Figure 7a shows the throughput averaged over all the ARM servers when the request intensity, represented by number of threads, changes. Note that with well-balanced load across the cluster each ARM server contributes almost the same amount of throughput. Because requests are assumed to be evenly distributed across the ARM servers, this average throughput is a metric helpful for estimating throughput of larger scale systems. In the following presentation all reported throughput refers to the average one. As shown in the figure, the write throughput increases with the increase of write intensity until it becomes saturated at about 60 requests/second when there are more than eight threads issuing write requests.

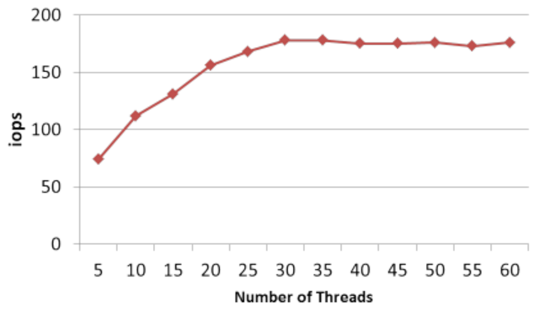
Now we send read requests to the system. Figure 7b shows the read throughput. It demonstrates a similar trend as that for write except that the read throughput reaches its peak at a larger number of threads. In addition, the average read throughput is much higher than that for write request (180 reads/s vs. 60 writes/s). Both observations are due to the fact that write requests consume much more network and disk bandwidths than read requests. The bandwidth consumption will be quantitatively analyzed in the next subsection.

We then send read and write requests to the system in a 3:1 ratio. Figure 7c shows the average throughput for this workload. It falls into the range between the corresponding ones for all-read and all-write workloads. Since most of the requests are reads, Atlas can achieve a maximum throughput of 125 requests per second.

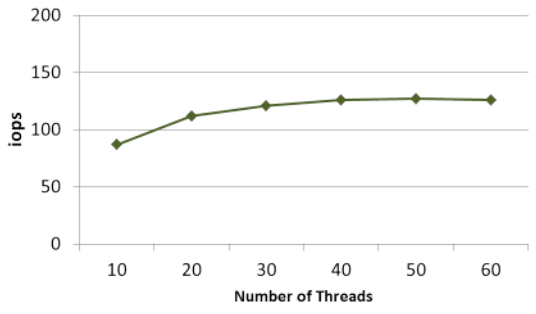
Figures 8a, 8b, and 8c show the latencies observed at the client side for the above three kinds of workloads, respectively. As the request intensity increases, the pressure on both the disks and the network increases, and the request service time accordingly increases. When the throughput reaches to its peak, the latency keeps increasing at a faster pace. Because write requests demand more resources, their latencies are more affected. Atlas has a threshold on request latency. A request with a response time larger than the threshold is considered as unsuccessful and is retried. In the production system, the threshold is set at $100ms$.



(a) Write workload

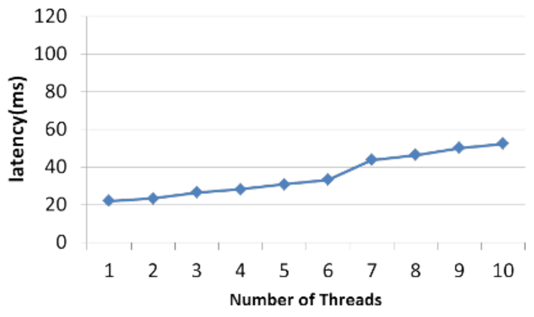


(b) Read workload

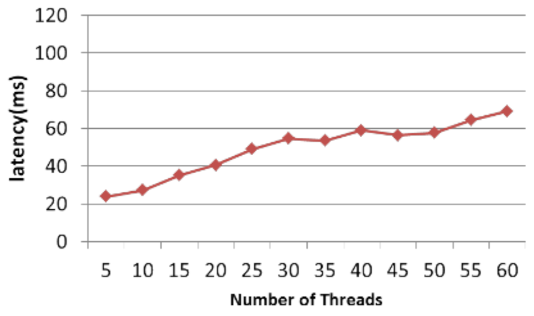


(c) Mixed workload with read and write of a 3:1 ratio.

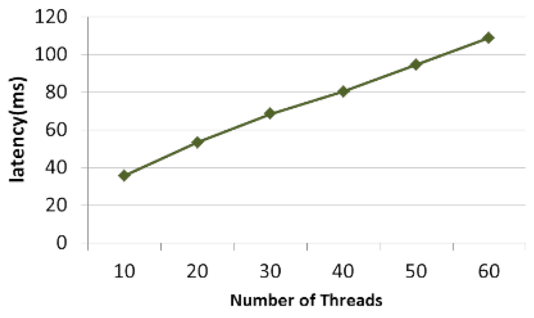
Fig. 7: Throughput with different workloads (write, read, or mixed). Workload intensity is represented by number of threads. The request size is 256KB. The reported throughput is an average of those of all ARM servers.



(a) Write workload

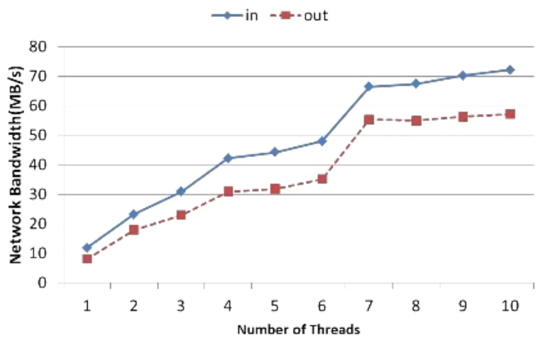


(b) Read workload

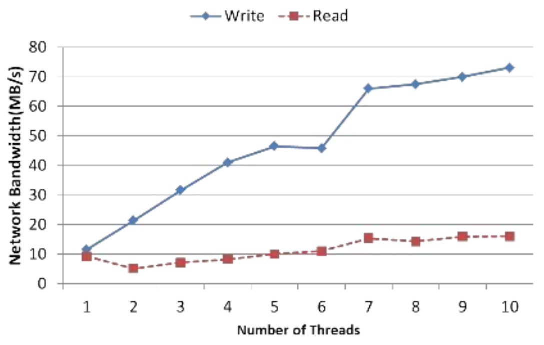


(c) Mixed workload with read and write of a 3:1 ratio.

Fig. 8: Request latencies with different workloads (write, read, or mixed). Workload intensity is represented by number of threads. The request size is 256KB.



(a) Network bandwidth



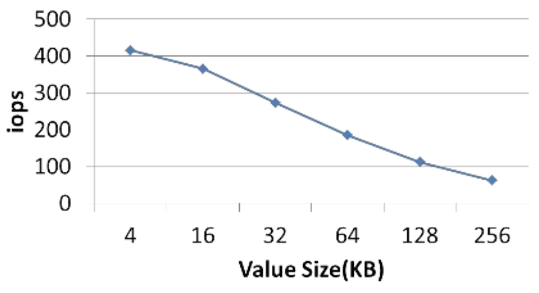
(b) Disk bandwidth

Fig. 9: Network and disks bandwidth with all-write workload.

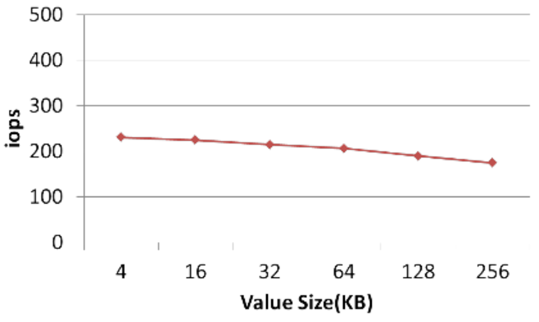
2) Bandwidth of Disks and Network: For the benefit of lower cost to enable larger-scale cluster, the system adopts 1Gbps network interface card (NIC). Accordingly it is necessary to understand which component, either disk or network, is the performance bottleneck. In this subsection we present measurements of the disk bandwidth and the network bandwidth with all-write workload as write requests place higher load on the network and disks.

Figure 9 shows both input and output bandwidths of the network and the disk write/read bandwidths in the all-write workload. With the increase of write intensities (more request-issuance threads), bandwidths of both network and disks increase. The ratio of input/output bandwidths and that of write/read bandwidths, as shown in the figures, match our analytic estimates. Assume the client sends requests for storing N MB data to the primary unit of a PIS slice, which sends the data to another two secondary units ($2N$ MB) of the slice via the network and saves its copy to the local disk. The other two units (or the servers hosting the units) receive their respective copies and save them to their local disks. Then the primary unit reads its copy from the local disk and writes the RS-coded data ($1.5N$ MB) to the RBS server via the network. Since the hashing function can evenly distribute the requests across the servers, each server would receive roughly the same amount of data from other servers as it sends to others. Consequently, the input/output bandwidth ratio of each network NIC would be about 4.5:3.5, and write/read bandwidth ratio of the disks would be about 4.5:1. This estimated ratios are confirmed in Figure 9. Also as shown in the figures, if each server has only one NIC and multiple disks, it is more likely that the network would be the first to become the system's bottleneck with increased load.

3) Impact of Request Size on the Throughput: In order to study the impact of request size on performance of Atlas, we send requests of different sizes to the system to its sustained highest throughput, which is reported in Figure 10. As shown, throughput decreases with the increase of request size for both read and write requests. For the small request size, such as 4KB or 8KB, Atlas achieves a much higher write throughput than read throughput. For writes, data is accumulated into large patches in PIS and stored to the RBS in large disk accesses (at least 8MB). The disk efficiency is always well exploited and the effective disk bandwidth is close to its peak one regardless of write request size or request pattern (access of continuous



(a) Write throughput



(b) Read throughput

Fig. 10: Performance of the system with different request sizes.

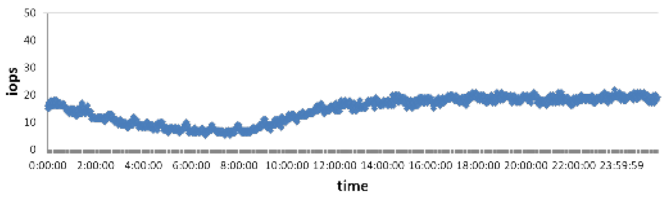
keys or random keys). So the write throughput almost linearly decreases with increase of the request size.

For read requests, the change of throughput with the request size is not as significant as that for write requests. For small requests, the service time is dominated by that of disk seeks as each individual read needs a disk access operation. Thus, read workload cannot achieve a throughput as high as that for write workload. With larger read requests, data transfer contributes more to the request service time and disk seek time is amortized over large data. In this case read workload has a higher throughput than write workload, which demands much more data transfer for data replication and data encoding. Our measurements also show that the request latency increases with the request size for both read and write requests. Their trends are similar to those for throughput.

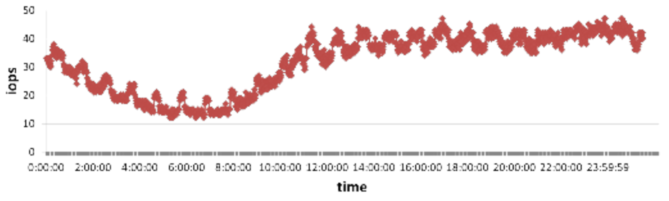
C. Measurements on a Production System

Atlas aims to provide a large-scale low-cost cloud storage service to a very large number of users. Currently it has more that 120 million users, each of whom can use up to 3TB storage space. In this paper, we choose one of its online clusters that has 700 ARM servers to analyze the real system performance. The cluster has similar setup as our test platform, including configuration of ARM server, types of NIC and hard disks, and co-hosting PIS slices and RBS servers on each ARM server.

1) Performance: Figures 11a and 11b show the write and read throughput of the cluster in a 24-hour service time period, respectively. We list read and write throughput separately to show the workload composition. As we mentioned, the throughput is an average of all the ARM server throughputs.



(a) Write requests



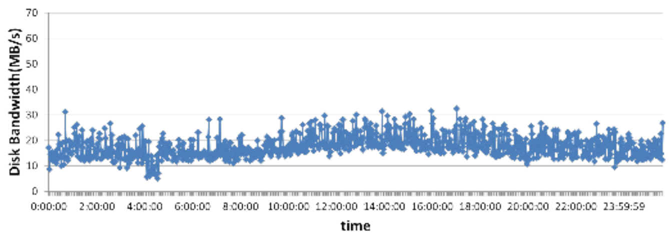
(b) Read request

Fig. 11: System throughput in a 24-hour time period.

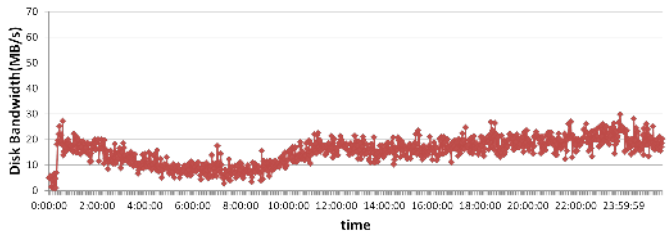
The reported throughput numbers suggest runtime performance behaviors of the entire system as the all ARM servers behave similarly. As shown, the write throughput varies in the range of 5 to 22 and the read throughput varies in the range of 10 to 48. We can observe that low request intensity happens between 4am and 9am (China time zone). For the rest of a day, both read and write intensities are at a stable level. Comparing these numbers to the peak throughput obtained from the testing cluster, we can tell that the servers still have substantial unused capacity. This conclusion can be further supported by the disk and network bandwidth measurements shown in Figures 12a, 12b, 12c, and 12d. Though the throughputs vary in a large range during the 24 hours, they do not reach the maximum bandwidth of the disks or the network NIC.

2) Failure Recovery: To evaluate Atlas's recovery performance, we set up a cluster of 12 servers and shut down a partserver during the system's running to simulate a system failure. We let a client keep sending read requests to the system, each for a random data in the failed partserver. We present throughput and latency of the read requests observed at the client in Figure 13. At the time when the server is shut down, there are about $65K$ parts stored on it. As shown in the figure, after the failure it takes almost one hour to get the lost data fully recovered when both throughput and latency are back to their normal readings. This time seems very long. Note that the performance of recovery process is approximately proportional to the total number of servers in the cluster, as the recovery load is evenly distributed onto all of them. In a system of hundreds of servers, this time would reduce to less than one minute.

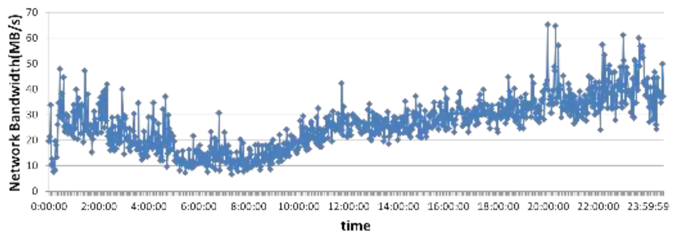
As shown in the figure, before the recovery is complete a read request for data in an uncovered part would take time much longer than regular read requests. An RBSClient usually sets a timeout threshold, and a retry is initialized whenever the threshold is passed. This retry can be conducted for at most three times before the data is successfully obtained. The threshold is usually set at 1 second. However, for lost data eight parts have to be read for decoding to produce the requested



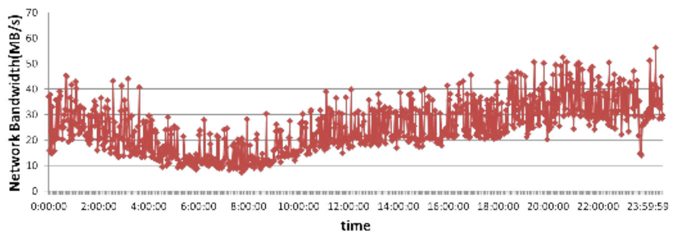
(a) Disk write bandwidth



(b) Disk read bandwidth

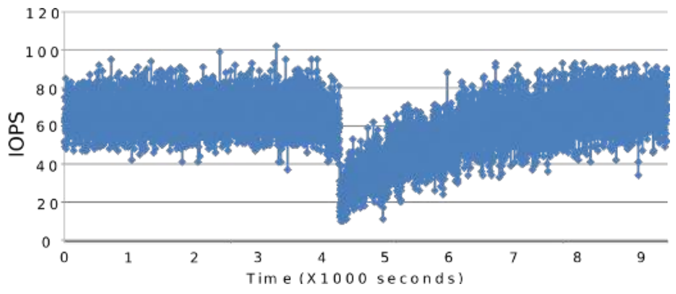


(c) Network incoming bandwidth

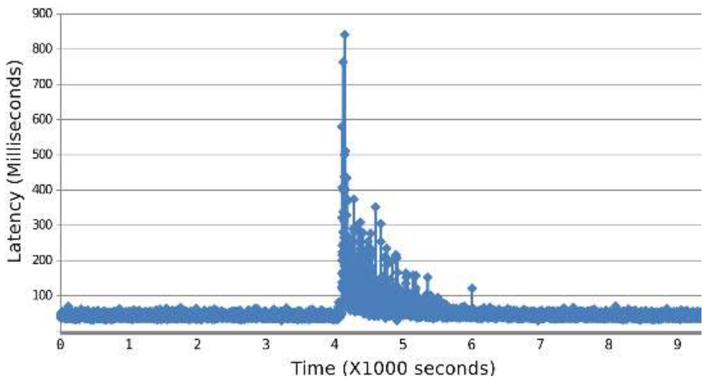


(d) Network outgoing bandwidth

Fig. 12: Disk and network bandwidths in a 24-hour time period.



(a) Read throughput



(b) Read latency

Fig. 13: Throughput (IOPS) and latency (milliseconds) of read requests for data on a server before and after it fails. The failure occurs at about the 4100th second.

data, which usually takes time longer than the threshold. Being aware of possible occurrence of such scenario, Atlas incrementally increases the threshold for the second and third retries to avoid unnecessary load on the RBS subsystem. Impact of the recovery on read requested for un-lost data and write requests are very minor.

3) Analysis of Power and Cost: One important design goal of Atlas is to reduce power consumption and cost. In this subsection, we compare the Atlas system with pre-Atlas, the x86 based system that had been deployed in Baidu for its cloud storage service. In the x86 based system, data are stored with three copies. In pre-Atlas each server has two 4-core Intel E5620 processors, 32GB main memory, 12 disks (total capacity 360TB), and a network adapter with 1Gbps bandwidth.

For real-world workloads, the average power consumption per TB can be reduced by about 53% after using Atlas system with ARM based servers. The reason is two fold: (1) the ARM processor consumes less power than the x86 processor; (2) the rack of ARM servers occupies much less space than that of x86 servers for the same storage capacity, leading to reduced consumption of electricity for power supply and thermal dissipation. In addition, Atlas with ARM-based servers can save about 70% of hardware cost per GB storage compared to x86 servers using 3-copy replication, including saving on CPU, memory, and hard disks.

IV. Additional Related Work

Google's GFS is a major effort on building scalable distributed file system for large-scale data-intensive applications [13]. A major difference between GFS and Atlas is the assumption on common data object sizes. GFS assumes files that are large or even huge ("Multi-GB files are common." [13]). Accordingly GFS divides files into 64MB chunks, and both space allocation and metadata management use the large chunk as the data unit. As Atlas assumes much small data objects, it has the PIS system for accumulating data into patch and producing 64MB blocks. While GFS stores there copies of each 64MB chunk on its chunkservers, Atlas divides a block into 8MB parts and stripes them across partservers for higher access parallelism and more efficient space usage. GFS has a master server managing metadata of the files. Compared to GFS's master that has to provide file namespace management and mechanisms such as lease and locking for consistency, Atlas's master server is much lighter and less complicated – it only needs to provide location service to partservers. Atlas moves the metadata service to the distributed PIS subsystem to keep the master from being a bottleneck.

In FDS [16] a blob is divided into 8MB tracts, which are stored in tractservers. It uses hashing to implement deterministic placement of tracts across tractservers, and the amount of metadata in its master is proportional only to the number of tractservers. In contrast, the amount of metadata in Atlas's RBS master is proportional to number of blocks. However, this does not limit its scalability as Atlas does not expect all the metadata have to be in memory. Instead, metadata cached at RBSClients remove the need to contact the master whenever Atlas serves read requests, which are dominant in its workloads. Similar to Panasas [23] using RAID5 to stripe files across many servers to accelerate data recovery, Atlas stripes its 64MB blocks across partservers. Atlas treats its metadata as KV pairs managed by the LSM tree. Similarly, TABLEFS [18] uses LSM tree to manage metadata and small files. However, it is a local file system.

While KV stores have become increasingly popular for managing small data items [6], [15], [12], their expensive internal data sorting has been a headache, especially when values are large. One effort to reduce the overhead is VT-tree [20], which attempts to minimize amount of moved data during data compaction. RocksDB takes a compaction arrangement different from LevelDB. Rather than involving two levels in a compaction, it compacts several consecutive levels at once intending to sort and push data faster to the lower level [8]. However, the improvement is limited as fundamentally the amplification is due to the difference on the sizes of data sets from different levels involved in a compaction. To keep the compactions from excessively degrading service quality of users' requests, bLSM limits the amount of data involved in each compaction operation [19]. More recently, LSM-trie is proposed to significantly reduce write amplification. It introduces a linear tree growth pattern, and uses a hash-function-based prefix tree structure to organize KV pairs [24]. Knowing that values would be larger than what are usually assumed for a typical KV store, Atlas minimizes the value size by replacing actual user data with a small piece of metadata.

A major design choice in distributed file systems or object stores is how to map names of entities (files or objects) to their physical locations. In Ceph files are striped across multiple objects, which are mapped to the OSDs (object storage devices) using with a two-level hashing approach [21]. Objects are first hashed to placement groups, which are then hashed to OSDs using a CRUSH function. A CRUSH function considers how to spread replicas of an object onto different server shelves, cabinets of shelves, and rows of cabinets for higher reliability and performance [22]. Similarly, Openstack Swift stores objects in partitions and spreads partition replicas onto different regions, zones, servers, and devices in a cluster, and a structure, called ring, is used to record and maintain the location information [7]. Atlas also uses two-level mapping to locate a KV pair. The PIS subsystem (the Index unit in a slice) tracks onto which block a pair is mapped, and the RBS subsystem (the RBS master) tracks on which partservers a block is stored. Because Atlas intends to minimize memory demand and increase efficiency of accessing relative small KV pairs, it makes a major effort on managing first-level mapping information at each server by using LSM-based KV stores. Its second-level mapping considers minimizing data loss due to failures. However, unlike opensource systems that have to consider all possible system configurations and workload characteristics, Atlas takes a simple approach – it only ensures that at most two parts from the same block are on the same rack and all parts are evenly spread in the cluster. This design meets our requirement on fault tolerance and greatly reduces the implementation cost.

Using Reed-Solomon coding, instead of data replication, means more data need to be read for data recovery. A new set of codes for erasure coding (LRC) have been proposed to reduce the cost [14]. In Atlas, the master makes efforts to ensure that parts used for recovery are read from a large number of servers and the read traffic is evenly spread. We have the plan to look into other coding methods, including LRC, for better performance and reduced redundant data.

FAWN is an energy-efficient key-value storage system that employs both low-power embedded CPUs and small amount of local flash storage on each node [9]. By using both low-power CPUs and flash storage, FAWN is aggressive in its efforts on improving energy efficiency. Similarly, Atlas also uses low-power processors to support I/O-intensive workload. To store a massive amount of data, FAWN has to replace flash with hard disks, as Atlas does. However, FAWN uses in-memory hash table and on-flash/disk linear (unsorted) data log. By organizing data in this way, its metadata (the hashtable) can be excessively large when the data set becomes large. In contrast, Atlas adopts LSM-tree structure to organize data.

V. Conclusions

Atlas is positioned as an object store whose workloads contain data objects that can be too small for file systems and too large for KV stores to efficiently manage. It has incorporated techniques proven to be effective for either large-scale file systems or KV stores. Meanwhile, Atlas distinguishes itself by proposing an architecture supporting a large-scale cloud storage service efficiently in terms of both performance and energy consumption. Atlas separates data and metadata into different subsystems (PIS and RBS), so that metadata can be efficiently managed in a KV store using LSM tree and data can be accumulated into large and fixed-unit blocks to enable RAID-like placement for parallel access and the Reed-Solomon coding for data reliability in a space-efficient manner. Experiments and online system profiling show that Atlas provides a highly scalable, reliable, and cost-effective service.

Acknowledgement

We are very grateful to our shepherd Dr. Stergios Anastasiadis, who made an extraordinary effort on the improvement of the paper's quality. Yuehai Xu helped with experiments described in Section I.C. and Xingbo Wu helped draw a number of graphs. Song Jiang had worked on the project during his visiting professorship at the Center for Energy-Efficient Computing and Applications in Peking University. His later effort was partially supported by the National Science Foundation under grants CAREER CCF 0845711 and CNS 1217948.

REFERENCES

[1] Your stuff, anywhere. In *https://www.dropbox.com/*

[2] Box — Free cloud storage and file sharing services. In *https://www.box.com/*

[3] Reed Solomon Error Correction. In *http://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction*

[4] Fio. In *hhttp://freecode.com/projects/fio*.

[5] Chinese Internet Giant Baidu Rolls Out World's First Commercial Deployment of Marvell's ARM Processor-based Server. In *http://www.marvell.com/company/news/pressDetail.do?releaseID=3576* February, 2013.

[6] Leveldb: a Fast and Lightweight Key/Value Database Library by Google. In *http://code.google.com/p/leveldb/*.

[7] Swifts documentation. In *http://docs.openstack.org/developer/swift/*

[8] Under the Hood: Building and Open-sourcing RocksDB. In *https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920*

[9] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. "FAWN: A Fast Array of Wimpy Nodes", In *ACM Symposium on Operating Systems Principles*, 2009

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. "Finding a Needle in Haystack: Facebooks Photo Storage", In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: a Distributed Storage System for Structured Data", In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazons Highly Available Key-value Store", In *ACM Symposium on Operating Systems Principles*, 2007

[13] S. Ghemawat, H. Gobioff, and S. Leung. "The Google File System", In *ACM Symposium on Operating Systems Principles*, 2003.

[14] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. "Erasure Coding in Windows Azure Storage", In *USENIX Annual Technical Conference*, 2012.

[15] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. "SILT: a Memory-Efficient, High-Performance Key-Value Store", In *ACM Symposium on Operating Systems Principles*, 2011.

[16] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. "Flat Datacenter Storage", In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[17] P. ONell, E. Cheng, D. Gawlick, and E. ONell. "The Log-Structured Merge-Tree (LSM-tree)", In *Acta Informatica*, 1996.

[18] K. Ren and G. Gibson. "TABLEFS: Enhancing Metadata Effciency in the Local File System", In *USENIX Annual Technical Conference*, 2013.

[19] R. Sears and R. Ramakrishnan. "bLSM: A General Purpose Log Structured Merge Tree", In *ACM SIGMOD International Conference on Management of Data*, 2012.

[20] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. "Building Workload-Independent Storage with VT-Trees", In *USENIX Conference on File and Storage Technologies*, 2013.

[21] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System", In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[22] S. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data", In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2006.

[23] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small1, J. Zelenka, and B. Zhou. "Scalable Performance of the Panasas Parallel File System", In *USENIX Conference on File and Storage Technologies*, 2008.

[24] X. Wu, Y. Xu, Z. Shao, and S. Jiang. "LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data", In *USENIX Annual Technical Conference*, 2015.