Curator: Self-Managing Storage for Enterprise Clusters

Ignacio Cano, *University of Washington;* Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, and Vinayak Khot, *Nutanix Inc.;* Arvind Krishnamurthy, *University of Washington*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/cano

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Curator: Self-Managing Storage for Enterprise Clusters

Ignacio Cano[*w], Srinivas Aiyar[n], Varun Arora[n], Manosiz Bhattacharyya[n], Akhilesh Chaganti[n], Chern Cheah[n], Brent Chun[n], Karan Gupta[n], Vinayak Khot[n] and Arvind Krishnamurthy[w]

[w]University of Washington
{icano,arvind}@cs.washington.edu
[n]Nutanix Inc.
curator@nutanix.com

Abstract

Modern cluster storage systems perform a variety of background tasks to improve the performance, availability, durability, and cost-efficiency of stored data. For example, cleaners compact fragmented data to generate long sequential runs, tiering services automatically migrate data between solid-state and hard disk drives based on usage, recovery mechanisms replicate data to improve availability and durability in the face of failures, cost saving techniques perform data transformations to reduce the storage costs, and so on.

In this work, we present Curator, a background MapReduce-style execution framework for cluster management tasks, in the context of a distributed storage system used in enterprise clusters. We describe Curator's design and implementation, and evaluate its performance using a handful of relevant metrics. We further report experiences and lessons learned from its five-year construction period, as well as thousands of customer deployments. Finally, we propose a machine learning-based model to identify an efficient execution policy for Curator's management tasks that can adapt to varying workload characteristics.

1 Introduction

Today's cluster storage systems embody significant functionality in order to support the needs of enterprise clusters. For example, they provide automatic replication and recovery to deal with faults, they support scaling, and provide seamless integration of both solid-state and hard disk drives. Further, they support storage workloads suited for virtual machines, through mechanisms such as snapshotting and automatic reclamation of unnecessary data, as well as perform space-saving transformations such as dedupe, compression, erasure coding, etc.

Closer examination of these tasks reveals that much of their functionality can be performed in the background. Based on this observation, we designed and implemented a background self-managing layer for storage in enterprise clusters. As part of this work, we addressed a set of engineering and technical challenges we faced to realize such a system. First, we needed an extensible, flexible, and scalable framework to perform a diverse set of tasks in order to maintain the storage system's health and performance. To that end, we borrowed technologies that are typically used in a different domain, namely big data analytics, and built a general system comprised of two main components: 1) Curator, a background execution framework for cluster management tasks, where all the tasks can be expressed as MapReduce-style operations over the corresponding data, and 2) a replicated and consistent key-value store, where all the important metadata of the storage system is maintained. Second, we needed appropriate synchronization between background and foreground tasks. We addressed these synchronization issues by having the background daemon act as a client to the storage system, and we let the latter handle all the synchronization. Third, minimal interference with foreground tasks was required. We accomplished this by using task priorities and scheduling heuristics that minimize overheads and interference. The resulting framework let us implement a variety of background tasks that enable the storage system to continuously perform consistency checks,[1] and be self-healing and self-managing.

We performed this work in the context of a commercial enterprise cluster product developed by Nutanix.[2] We developed the system over a period of five years, and have deployed it on thousands of enterprise clusters. We report on the performance of the system and experiences gleaned from building and refining it. We found that Curator performs garbage collection and replication effectively, balances disks, and makes storage access efficient through a number of optimizations. Moreover, we realized that the framework was general enough to incorporate a wide variety of background transformations as well as simplified the construction of the storage system.

Nonetheless, we noticed that our heuristics do not necessarily work well in all clusters as there is significant heterogeneity across them. Thus, we recently started developing a framework that uses machine learning (ML) for addressing the issues of when should these background management tasks be performed and how much work they should do. The ML-based approach has two

[*]This work was done while the author was interning at Nutanix Inc.

[1]This eliminates heavyweight fsck-like operations at recovery time.

[2]Nutanix is a provider of enterprise clusters. For more details refer to http://www.nutanix.com.

key requirements: 1) high predictive accuracy, and 2) the ability to learn or adapt to (changing) workload characteristics. We propose using reinforcement learning, in particular, the Q-learning algorithm. We focus our initial efforts on the following tiering question: how much data to keep in SSDs and HDDs? Empirical evaluation on five simulated workloads confirms the general validity of our approach, and shows up to ∼20% latency improvements.

In summary, our main contributions are:

- We provide an extensive description of the design and implementation of Curator, an advanced distributed cluster background management system, which performs, among others, data migration between storage tiers based on usage, data replication, disk balancing, garbage collection, etc.
- We present measurements on the benefits of Curator using a number of relevant metrics, e.g., latency, I/O operations per second (IOPS), disk usage, etc., in a contained local environment as well as in customer deployments and internal corporate clusters.
- Finally, we propose a model, based on reinforcement learning, to improve Curator's task scheduling. We present empirical results on a storage tiering task that demonstrate the benefits of our solution.

2 Distributed Storage for Enterprise Clusters

We perform our work in the context of a distributed storage system designed by Nutanix for enterprise clusters. In this section, we provide an overview of the software architecture, the key features provided by the storage system, and the data structures used to support them. Herein, we present the necessary background information for understanding the design of Curator.

2.1 Cluster Architecture

The software architecture is designed for enterprise clusters of varying sizes. Nutanix has cluster deployments at a few thousand different customer locations, with cluster sizes typically ranging from a few nodes to a few dozens of nodes. Cluster nodes might have heterogeneous resources, since customers add nodes based on need. The clusters support virtualized execution of (legacy) applications, typically packaged as VMs. The cluster management software provides a management layer for users to create, start, stop, and destroy VMs. Further, this software automatically schedules and migrates VMs taking into account the current cluster membership and the load on each of the individual nodes. These tasks are performed by a *Controller Virtual Machine* (CVM) running on each node in the cluster.

The CVMs work together to form a distributed system that manages all the storage resources in the cluster. The CVMs and the storage resources that they manage provide the abstraction of a distributed storage fabric (DSF) that scales with the number of nodes and provides transparent storage access to user VMs (UVMs) running on any node in the cluster. Figure 1 shows a high-level overview of the cluster architecture.

Applications running in UVMs access the distributed storage fabric using legacy filesystem interfaces (such as NFS, iSCSI, or SMB). Operations on these legacy filesystem interfaces are interposed at the hypervisor layer and redirected to the CVM. The CVM exports one or more block devices that appear as disks to the UVMs. These block devices are virtual (they are implemented by the software running inside the CVMs), and are known as vDisks. Thus, to the UVMs, the CVMs appear to be exporting a storage area network (SAN) that contains disks on which the operations are performed.[3] All user data (including the operating system) in the UVMs resides on these vDisks, and the vDisk operations are eventually mapped to some physical storage device (SSDs or HDDs) located anywhere inside the cluster.

Although the use of CVMs introduces an overhead in terms of resource utilization,[4] it also provides important benefits. First, it allows our storage stack to run on *any* hypervisor. Second, it enables the upgrade of the storage stack software without bringing down nodes. To support this feature, we implemented some simple logic at the hypervisor-level to effectively multi-path its I/O to another CVM in the cluster that is capable of serving the storage request. Third, it provides a clean separation of roles and faster development cycles. Building a complex storage stack in the hypervisor (or even the kernel) would have severely impacted our development speed.

2.2 Storage System and Associated Data Structures

We now describe the key requirements of the DSF and how these requirements influence the data structures used for storing the metadata and the design of Curator.

R1 Reliability/Resiliency: the system should be able to handle failures in a timely manner.

R2 Locality preserving: data should be migrated to the node running the VM that frequently accesses it.

R3 Tiered Storage: data should be tiered across SSDs, hard drives, and the public cloud. Further, the SSD tier should not merely serve as a caching layer for hot data, but also as permanent storage for user data.

R4 Snapshot-able: the system should allow users to quickly create snapshots for greater robustness.

[3] Unlike SAN/NAS and other related solutions (e.g., OneFS [15], zFS [34], GlusterFS [18], LustreFS [39], GPFS [36]), the cluster nodes serve as both VM compute nodes as well as storage nodes.

[4] We are currently exploring some alternatives to reduce such overhead, e.g., pass-through drivers so that the CVMs can handle the disk I/O directly, RDMA to move replication data, etc.

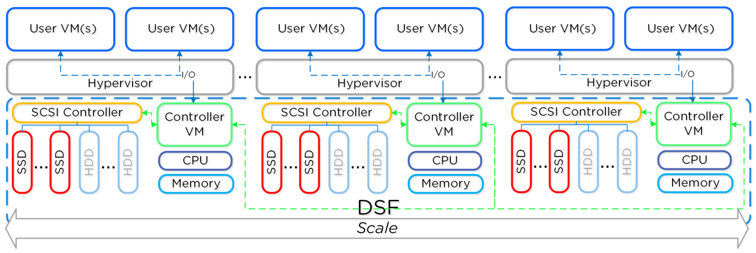

Figure 1: Cluster architecture and the Distributed Storage Fabric. UVMs access the storage distributed across the cluster using CVMs.

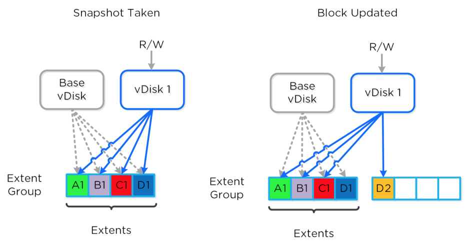

Figure 2: Snapshotting and copy-on-write update to snapshotted extents.

R5 Space efficient: the system should achieve high storage efficiency while supporting legacy applications and without making any assumptions regarding file sizes or other workload patterns.

R6 Scalability: the throughput of the system should scale with the number of nodes in the system.

The above set of requirements manifest in our system design in two ways: (a) the set of data structures that we use for storing the metadata, and (b) the set of management tasks that will be performed by the system. We discuss the data structures below and defer the management tasks performed by Curator to §3.2.

Each vDisk introduced in §2.1 corresponds to a virtual address space forming the individual bytes exposed as a disk to user VMs. Thus, if the vDisk is of size 1 TB, the corresponding address space maintained is 1 TB. This address space is broken up into equal sized units called vDisk blocks. The data in each vDisk block is physically stored on disk in units called extents. Extents are written/read/modified on a sub-extent basis (a.k.a. slice) for granularity and efficiency. The extent size corresponds to the amount of live data inside a vDisk block; if the vDisk block contains unwritten regions, the extent size is smaller than the block size (thus satisfying R5).

Several extents are grouped together into a unit called an extent group. Each extent and extent group is assigned a unique identifier, referred to as extentID and extentGroupID respectively. An extent group is the unit of physical allocation and is stored as a file on disks, with hot extent groups stored in SSDs and cold extent groups on hard drives (R3). Extents and extent groups are dynamically distributed across nodes for fault-tolerance, disk balancing, and performance purposes (R1, R6).

Given the above core constructs (vDisks, extents, and extent groups), we now describe how our system stores the metadata that helps locate the actual content of each vDisk block. The metadata maintained by our system consists of the following three main maps:

- vDiskBlock map: maps a vDisk and an offset (to identify the vDisk block) to an extentID. It is a logical map.
- extentID map: maps an extent to the extent group that it is contained in. This is also a logical map.
- extentGroupID map: maps an extentGroupID to the physical location of the replicas of that extentGroupID and their current state. It is a physical map.

Here are a few implications regarding the aforementioned data structures. Multiple vDisks created through snapshots can share the same extent. The vDiskBlock map of a snapshot can either directly point to an extent shared with a prior snapshot or have a missing entry, in which case the vDiskBlock map of the previous snapshot is consulted. This facility allows for instantaneous creation of snapshots, i.e., we can create an empty vDiskBlock map entry and have it point to the previous snapshot for all of its unfilled entries (R4). At the same time, it enables a later optimization of metadata lookup using lazy filling of the missing entries (§3.2.4). When a vDisk block is updated on the new snapshot, a new extent is created to hold the updated data. Figure 2 shows an example in which vDisk 1 is created as a snapshot and its vDiskBlock map has already been populated with the correct pointers to the corresponding extents (left portion). Later it is updated to point to a new extent upon an update to one of its vDisk blocks (right portion).

The level of indirection introduced by the extentID map allows efficient updates whenever data from one extent group is relocated to another (e.g., to optimize access), as it is a single place in which we store the physical extentGroupID in which the extent is located (thus aiding R2, R3).

Finally, a set of management operations can be performed by only consulting the extentGroupID map. For example, we can detect (and repair) if the number of replicas for a given extentGroupID falls under certain threshold by only accessing this map – the logical maps will remain untouched – thus addressing R1.

Overall, the resulting data structures set us up to perform the various management tasks described in §3.2 in an efficient and responsive manner.

3 Curator

Curator is the cluster management component responsible for managing and distributing various storage management tasks throughout the cluster, including continuous consistency checking, fault recovery, data migration, space reclamation, and many others. In this section, we describe Curator's architecture (§3.1), the tasks it performs (§3.2), and the policies under which those tasks are executed (§3.3). Finally, we demonstrate its value with a set of empirical results (§3.4), and share our experiences and lessons learned from building Curator (§3.5).

3.1 Curator Architecture

Curator's design is influenced by the following considerations. First, it should scale with the amount of storage served by the storage system and cope with heterogeneity in node resources. Second, Curator should provide a flexible and extensible framework that can support a broad class of background maintenance tasks. Third, Curator's mechanisms should not interfere with nor complicate the operations of the underlying storage fabric. Based on these considerations, we designed a system with the following key components and/or concepts:

Distributed Metadata: The metadata (i.e., the maps discussed in the previous section) is stored in a distributed ring-like manner, based on a heavily modified Apache Cassandra [22], enhanced to provide strong consistency for updates to replicated keys. The decision behind having the metadata distributed lies in the fact that we do not want the system to be bottlenecked by metadata operations. Paxos [23] is utilized to enforce strict consistency in order to guarantee correctness.

Distributed MapReduce Execution Framework: Curator runs as a background process on every node in the cluster using a master/slave architecture. The master is elected using Paxos, and is responsible for task and job delegation. Curator provides a MapReduce-style infrastructure [10] to perform the metadata scans, with the master Curator process managing the execution of MapReduce operations. This ensures that Curator can scale with the amount of cluster storage, adapt to variability in resource availability across cluster nodes, and perform efficient scans/joins on metadata tables.[5]

Although our framework bears resemblance to some data-parallel engines (e.g., Hadoop, Spark), the reason behind writing our own instead of re-purposing an existing one was two-fold: 1) efficiency, as most of these open-source big data engines are not fully optimized to make a single node or a small cluster work efficiently,[6] a must in our case, and 2) their requirement of a distributed storage system (e.g., HDFS), a recursive dependence that we did not want to have in our clustered storage system.

Co-design of Curator with Underlying Storage System: The distributed storage fabric provides an extended API for Curator, including but not limited to the following low-level operations: migrate an extent from one extent group to another, fix an extent group so that it meets the durability and consistency requirements, copy a block map from one vDisk to another, and perform a data transformation on an extent group. Curator only performs operations on metadata, and gives *hints* to an I/O manager service in the storage system to act on the actual data. It is up to the storage system to follow Curator's advice, e.g., it may disregard a suggestion of executing a task due to heavy load or if a concurrent storage system operation has rendered the operation unnecessary.[7] This approach also eliminates the need for Curator to hold locks on metadata in order to synchronize with the foreground tasks; concurrent changes only result in unnecessary operations and does not affect correctness.

Task Execution Modes and Priorities: During a MapReduce-based scan, the mappers and reducers are responsible for scanning the metadata in Cassandra, generating intermediate tables, and creating synchronous and asynchronous tasks to be performed by the DSF. Synchronous tasks are created for fast operations (e.g., delete a vDisk entry in the vDiskBlock metadata map) and are tied to the lifetime of the MapReduce job. Conversely, asynchronous tasks are meant for heavy operations (e.g., dedupe, compression, and replication) and are sent to the master periodically, which batches them, and sends them to the underlying storage system for later execution (with throttling enabled during high load). These tasks are not tied to the lifetime of the MapReduce job. Note that although these tasks are generated based on a cluster-wide global view using MapReduce-based scans, their execution is actually done in the individual nodes paced at a rate suitable to each node's workload.[8] In other words, we compute what tasks need to be performed in a bulk-synchronous manner, but execute them independently (in any order) per node.

3.2 Curator Management Tasks

In this section, we describe how the Curator components work together to perform four main categories of tasks. Table 2 in Appendix A includes a summary of the categories, tasks, and metadata maps touched by each of the tasks.

3.2.1 Recovery Tasks

Disk Failure/Removal (DF) and Fault Tolerance (FT): In the event of a disk or node failure, or if a user simply wants to remove/replace a disk, Curator receives a notification and starts a metadata scan. Such a scan

[5]Note that any metadata stored in a distributed key-value store should be able to utilize this MapReduce framework.

[6]They assume they will have enough compute as their deployments tend to scale out.

[7]Curator makes sure that the I/O manager knows the version of metadata it based its decision on. The I/O manager checks the validity of the operations based on metadata timestamps (for strong consistency tasks like Garbage Collection) or last modified time (for approximate tasks such as Tiering).

[8]The rate depends on the CPU/disk bandwidth available at each node.

finds all the extent groups that have one replica on the failed/removed/replaced node/disk and notifies the underlying storage system to fix these under-replicated extent groups to meet the replication requirement. This is handled by the storage system as a critical task triggered by a high-priority event, which then aims to reduce the time that the cluster has under-replicated data. Note that these tasks require access to just the extentGroupID map and benefit from the factoring of the metadata into separate logical and physical maps.

3.2.2 Data Migration Tasks

Tiering (T): This task moves cold data from a higher storage tier to a lower tier, e.g., from SSD to HDD, or from HDD to the public cloud. Curator is only involved in *down* migration, not *up*, i.e., it does not migrate data from HDD to SSD, or from the public cloud to HDD. *Up* migration, on the other hand, is done by the DSF upon repeated access to hot data. Taken together, the actions of Curator and DSF aim to keep only the hottest data in the fastest storage tiers in order to reduce the overall user access latency.

This task is costly as it involves actual data movement, not just metadata modifications. Curator computes the "coldness" of the data during a metadata scan, and notifies the DSF to perform the actual migration of the coldest pieces. The coldness is computed based on least recently used (LRU) metrics. The cold data is identified by the modified time (mtime) and access time (atime), retrieved during a scan. Both mtime (write) and atime (read) are stored in different metadata maps. The former is located in the extentGroupID map, whereas the latter resides in a special map called extentGroupIDAccess map. This latter access map was especially created to support eventual consistency for non-critical atime data (in contrast to the extentGroupID map's strict consistency requirements) and thereby improve access performance. As a consequence of being stored in separate maps, the mtime and atime of an extent group might be located in different nodes, therefore, communication may be required to combine these two attributes.

In order to compute the "coldness" of the data, a MapReduce job is triggered to scan the aforementioned metadata maps. The *map* tasks emit the extentGroupID as key, and the mtime (or atime) as value. The *reduce* tasks perform a join-like reduce based on the extentGroupID key. The reduce tasks generate the $(egid, mtime, atime)$ tuples for different extent groups and sort these tuples to find the cold extent groups. Finally, the coldest extent groups are sent to the DSF for the actual data migration.

Disk Balancing (DB): Disk Balancing is a task that moves data within the same storage tier, from high usage disks to low usage ones. The goal is to bring the usage of disks within the same tier, e.g., the cluster SSD tier, as close as possible to the mean usage of the tier. This task not only reduces the storage tier imbalance, but also decreases the cost of replication in the case of a node/disk failure. To minimize unnecessary balancing operations, Curator does not execute the balancing if the mean usage is low, even if the disk usage spread is high. Further, in case it executes the balancing, as with Tiering, it only attempts to move cold data. The MapReduce scans identify unbalanced source and target disks, together with cold data, and notifies the storage fabric to perform the actual migration of extent groups.

3.2.3 Space Reclamation Tasks

Garbage Collection (GC): There are many sources of garbage in the storage system, e.g., when an extent is deleted but the extent group still has multiple live extents and cannot be deleted, garbage due to wasting preallocated larger disk spaces on extent groups that became immutable and did not use all of the allocated quota, when the compression factor for an extent group changes, etc. GC increases the usable space by reclaiming garbage and reducing fragmentation. It does so in three ways:

- Migrate Extents: migrate live extents to a new extent group, delete the old extent group, and then reclaim the old extent group's garbage. It is an expensive operation as it involves data reads and writes. Therefore, Curator performs a cost-benefit analysis per extent group and chooses for migration only the extent groups where the benefit (amount of dead space in the extent group) is greater than the cost (sum of space of live extents to be migrated).
- Pack Extents: try to pack as many live extents as possible in a single extent group.
- Truncate Extent Groups: reclaim space by truncating extent groups, i.e., reducing their size.

Data Removal (DR): The data structures introduced in §2.2 are updated in such a way that there cannot be dangling pointers, i.e., there cannot be a vDisk pointing to an extent that does not exist, or an extent pointing to an extent group that does not exist. However, there can be unreachable data, e.g., an extent that is not referenced by any vDisk, or an extent group that is not referenced by any extent. These could be due to the side-effects of vDisk/snapshot delete operations or a consequence of failed DSF operations.

In DSF, extent groups are created first, then extents, and finally vDisks. For removal, the process is backwards; unused vDisks are removed first, then the extents, and finally the unreferenced extent groups. The DR task performs this removal process in stages (possibly in successive scans), and enables the reclamation of unused space in the system.[9]

3.2.4 Data Transformation Tasks

Compression (C) and Erasure Coding (EC): Curator scans the metadata tables and flags an extent group as a candidate for compression/coding if the current compression of the extent group is different from the desired compression type or if the extent group is sufficiently cold.

Once Curator identifies the extent groups (thus extents) for compression/coding, it sends a request to the DSF, which performs the actual transformation by migrating the extents. The main input parameters of this request are the set of extents to be compressed (or migrated), and the extentGroupID into which these extents will be migrated. If the latter is not specified, then a new extent group is created. This API allows us to pack extents from multiple source extent groups into a single extent group. Also, instead of always creating a new extent group to pack the extents, Curator can select an existing extent group and pack more extents into it. The target extent groups are also identified using MapReduce scans and sorts.

Deduplication (DD): Dedupe is a slightly different data transformation, as it involves accessing other metadata maps. During a scan, Curator detects duplicate data based on the number of copies that have the same precomputed fingerprint, and notifies the DSF to perform the actual deduplication.

Snapshot Tree Reduction (STR): As briefly mentioned in §2.2, the storage system supports snapshots, which are *immutable* lightweight copies of data (similar to a simlink), and can therefore generate an instantaneous copy of a vDisk. Every time the system takes a snapshot, a new node is added to a tree, called the snapshot tree, and the vDisk metadata is inherited. Snapshot trees can become rather deep. In order to be able to read a leaf node from a tree, the system needs to traverse a sequence of vDiskBlock map entries. The bigger the depth of a tree, the more inefficient the read operation becomes.

To address this, the STR task "cuts" the snapshot trees, by copying vDiskBlock map metadata from parents to child nodes. There are two flavors, *partial* and *full* STR, and their use depends on whether we need vDisk metadata only from some ancestors (*partial*) or from all of them (*full*). Once the copy is performed, the child vDisks have all the information needed for direct reads, i.e., there is no need to access the ancestors' metadata, thus, the read latency is reduced.

[9]Note that only the deletion of extent groups frees up physical space.

3.3 Policies

The tasks described in §3.2 are executed based on (roughly) four different policies, described below.

Event-driven: These tasks are triggered by events. For example, whenever a disk/node fails, a Recovery task is executed, no matter what. These are critical, higher priority tasks.

Threshold-based: These are dynamically executed tasks based on fixed thresholds violations. For example, when the tier usage is "high", or the disk usage is "too" unbalanced, etc. We provide both examples below.

In order to be eligible for the Tiering task, the storage tier usage from where we want to down migrate the data should exceed a certain threshold f, whereas the destination tier usage should not exceed a threshold d, i.e., it should have enough space to store the data to be moved. Further, a threshold h indicates by how much the usage percentage is to be reduced.[10]

Regarding DB, in order to be considered for balancing, the mean tier usage should exceed a threshold m and the disk usage spread should be greater than a threshold s. The disk usage spread is the difference between the disk with maximum usage and the disk with minimum usage within the tier.[11]

Periodic Partial: We next consider tasks that are neither triggered nor threshold-driven, but access only a subset of the metadata maps. These tasks are executed every h_1 hours, and are grouped based on the metadata tables they scan.

Periodic Full: All tasks are executed as part of a *full* scan every h_2 hours. We call this policy *full* as it scans all three metadata tables in Cassandra, the vDiskBlock, extentID, and extentGroupID maps. Because the *partial* scan only works on a subset of the metadata maps, it can run more frequently than the *full* scan, i.e., $h_1 < h_2$. In general, scans are expensive, hence, when a scan is running, Curator tries to identify as many asynchronous tasks as possible and lets them drain into the DSF over time. In other words, Curator combines the processing that must be done for the different tasks in order to reduce the scans' overheads.

3.4 Evaluation

In this section, we evaluate Curator's effectiveness with respect to a number of metrics. We report results on three different settings: a) customer clusters, where Curator is always turned on, b) internal corporate production clusters, where Curator is also on, and c) an internal local cluster, where we enable/disable Curator to perform controlled experiments.

[10]The default threshold values are $f = 75\%$, $d = 90\%$, and $h = 15\%$.
[11]The default threshold values are $m = 35\%$ and $s = 15\%$.

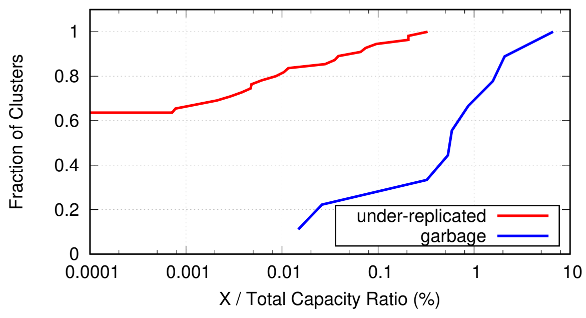

Figure 3: Under-Replicated Data and Garbage percentages (in log-scale) with respect to Total Storage Capacity

3.4.1 Customer and Corporate Clusters

We leverage historical data from a number of real clusters to assess Curator capabilities. In particular, we use ∼50 clusters over a period of two and a half months[12] to demonstrate Curator's contributions to the overall cluster resiliency, and data migration tasks. We also collect data from ten internal corporate clusters over a period of three days. These clusters are very heterogeneous in terms of load and workloads, as they are used by different teams to (stress) test diverse functionalities.

Recovery: Figure 3 shows the cumulative distribution function (CDF) of the average under-replicated data as a percentage of the overall cluster storage capacity (in log-scale) in our customer clusters. We observe that around 60% of the clusters do not present any under-replication problem. Further, 95% of the clusters have at most an average of 0.1% under-replicated data.

For further confirmation, we access the availability cases[13] of the 40% of clusters from Figure 3 that reported under-replication. We consider only those cases for the clusters that were opened within 2 weeks of the under-replication event (as indicated by the metric timestamp), and look for unplanned down time in those clusters. We do not find any unplanned down time in such clusters, which suggests that Curator ensured that replication happened upon detecting the under-replication event so that there was no availability loss.

Tiering: Figure 4 shows the CDF of SSD and HDD usage in our customer clusters. We observe that 40% of the clusters have a SSD usage of at most ∼70-75%. From the remaining 60% of the clusters, many of them have 75% SSD usage, which indicates that the Tiering task is doing its job; data has been down-migrated so that the SSDs can absorb either new writes or up-migration of hot data. In the other 10%, the SSD usage is slightly higher, which means that although the Tiering task is being executed, it cannot entirely cope with such (storage-heavy) workloads. We also note that HDD utilization is typically less, with 80% of clusters having less than 50% HDD usage.

[12] June to mid August 2016.

[13] We have access to a database of cases information corresponding to various issues encountered in real clusters, where we can query using different filters, e.g., availability problems, etc.

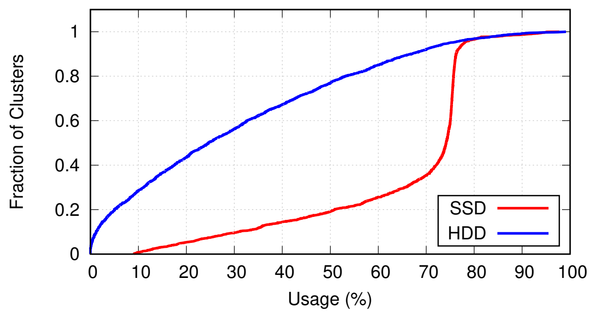

Figure 4: SSD and HDD Usage

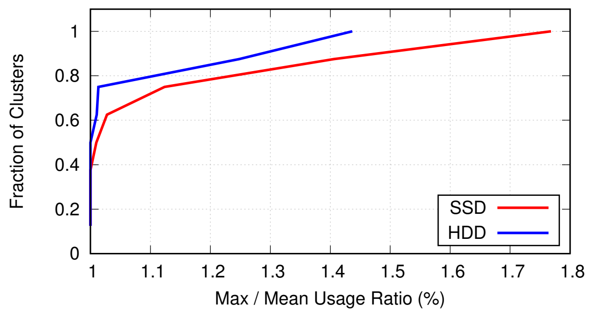

Figure 5: Max/Mean Usage Ratio

Garbage Collection: Figure 3 also illustrates the CDF of the (95th percentile) percentage of garbage with respect to the total storage capacity (in log-scale) in our corporate clusters. We observe that 90% of the clusters have less than 2% of garbage, which confirms the usefulness of the Garbage Collection task.

Disk Balancing: Figure 5 validates Disk Balancing in our corporate clusters. We plot maximum over mean usage ratio, for both SSDs and HDDs. We observe that in 60% (SSDs) and 80% (HDDs) of the cases, the maximum disk usage is almost the same as the mean.

3.4.2 Internal Cluster

We are interested in evaluating the costs incurred by Curator as well as the benefits it provides, with respect to a "Curator-less" system, i.e., we want to compare the cluster behavior with Curator enabled and when Curator is disabled. Given that we cannot toggle Curator status (ON-OFF) in customer deployments, in this section, we do so in an internal test cluster. We provide a summary of our findings in Table 3 in Appendix B.

Setup: We use a 4-node cluster in our experiments. The cluster has 4 SSDs and 8 HDDs, for a total size of 1.85 TB for SSDs, and 13.80 TB for HDDs, with an overall CPU clock rate of 115.2 GHz, and a total memory of 511.6 GiB.

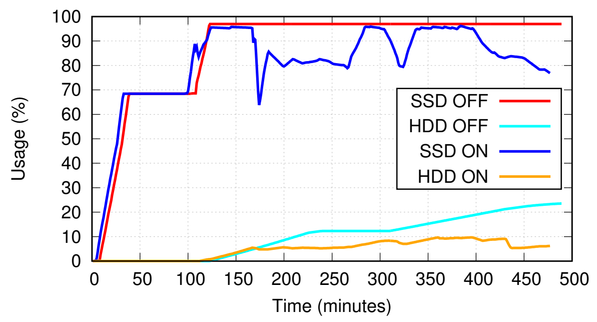

Figure 6: SSD and HDD Usage with Curator ON and OFF

Workloads: We use Flexible I/O Tester (fio[14]) to generate the exact same workloads for testing both settings, i.e., the behavior of the system when Curator is ON and OFF. We re-image the cluster to the same initial clean state when we toggle Curator status.

We simulate three online transaction processing (OLTP) workloads, *small*, *medium*, and *large*, which we execute *sequentially* as part of a single run. Each of these workloads go over three phases, *prefill*, *execution*, and *destroy*. In the *prefill* stage, they create their own user virtual machines (UVMs), together with their associated vDisks. After the *prefill* phase is done, they proceed to *execution*, where the actual workload operations (reads and/or writes) are executed. Following *execution*, the *destroy* stage begins, where the UVMs and associated vDisks are destroyed, i.e., the vDisks' space can be reclaimed. Appendix C describes the workloads in more detail.

Benefits: In terms of benefits, we consider latency and storage usage, which mainly highlight the benefits of T and DR tasks.

Figure 6 shows SSD and HDD usage over time for both Curator ON and OFF. We observe that SSD and HDD usage when Curator is OFF follows a non-decreasing pattern. When SSDs get full (∼125 minutes), all the data starts being ingested directly into HDDs. Instead, when Curator is ON, we see the effects of Tiering, where colder data is moved to HDDs when the default usage threshold is surpassed (§3.3). Even though Tiering kicks in "on time", the data ingestion rate is so high that the task cannot entirely cope with it, therefore, we observe SSD usage percentages in the 90's. At the end, we see that it reaches the 70's.

Figure 6 also illustrates the benefits of Garbage Collection and Data Removal in Curator. When Curator is disabled, we observe a 96% SSD and 23% HDD usage (∼5 TB) at the end of the run, whereas, when Curator is enabled, we see a 76% SSD and 6% HDD usage (∼2.27 TB). The average storage usage over the whole run is ∼2 TB and ∼3 TB for Curator ON and OFF respectively. These differences are mainly due to the DR task (§3.2.3). As described above, the *destroy* phase of each workload, where UVMs and associated vDisks are destroyed, allows the DR task to kick in and start the data removal process, allowing huge storage savings.

Regarding latency, we see an average of ∼12 ms when Curator in ON, and ∼62 ms when is OFF. We measure these values on the *execution* phase of the workloads. As time progresses, the latencies increase when Curator is disabled. We speculate this is due to the fact that the newest ingested data goes directly into HDDs, as SSDs are already full, thus, high latency penalties are paid when reads/writes are issued.

Costs: We consider CPU and memory usage, as well as the number of I/O operations performed.

We see that the number of IOPS executed is higher when Curator is ON, as many of its tasks require reading and writing actual data. Still, the overall average IOPS when Curator is enabled lies in the same ballpark as the disabled counterpart, ∼1400 as opposed to ∼1150 when Curator is OFF.

We also notice that when Curator is ON, the CPU usage is slightly higher. This is due to Curator internals, i.e., its MapReduce infrastructure. Although the mappers primarily scan the metadata (mostly I/O intensive), the reducers involve significant logic to process the scanned information (mostly CPU intensive). Even though the average CPU usage is higher when Curator is enabled, 18% as opposed to 14%, the value is still in an acceptable range, and shows a somewhat stable pattern over time. Regarding memory usage, we do not see a difference between both versions of the system, as shown in Table 3.

3.5 Experiences and Lessons Learned

In this section, we highlight some of the key experiences we gleaned from building Curator.

Firstly, the fact that we had a background processing framework in the form of Curator simplified the addition of new features into the file system. Whenever a new feature was to be added, we systematically identified how that feature could be factored into a foreground component (which would be run as part of the DSF) and a background component (which would be run as part of Curator). This allowed for easy integration of new functionality and kept the foreground work from becoming complex. As an example, our foreground operations do not perform transactional updates to metadata. Instead, they rely on Curator to roll-back incomplete operations as part of its continuous background consistency checks.

Secondly, having a background MapReduce process to do post-process/lazy storage optimization allowed us to achieve better latencies for user I/O. While serving an I/O request, the DSF did not have to make globally optimal

[14] https://github.com/axboe/fio

decisions on where to put a piece of data nor what transformations (compression, dedupe, etc.) to apply on that data. Instead, it could make decisions based on minimal local context, which allowed us to serve user I/O faster. Later on, Curator in the background would re-examine those decisions and make a globally optimal choice for data placement and transformation.

Thirdly, given that we use MapReduce, almost all Curator tasks were required to be expressed using MapReduce constructs (map and reduce operations). For most tasks, this was straightforward and allowed us to build more advanced functionality. MapReduce was however cumbersome in some others, as it required us to scan entire metadata tables during the map phase. We leveraged once again our infrastructure to first filter out which portions of the metadata maps to analyze before performing the actual analysis. This filter step became another map operation, and could be flexibly added to the beginning of a MapReduce pipeline. In retrospect, given our choice of the metadata repository (i.e., a distributed key-value store), we believe MapReduce was the right choice as it provided an easy and efficient way to process our metadata, where we could leverage the compute power of multiple nodes and also ensure that the initial map operations are performed on node-local metadata, with communication incurred only on a much smaller subset of the metadata communicated to the reduce steps.

In terms of the distributed key-value store, although it needed more hard work from the perspective of processing the metadata, it provided us a way to scale from small clusters (say three nodes) to larger (hundreds of nodes) ones. If we had decided to keep the data in a single node, a SQL-Lite like DB could have been enough to do most of the processing we are doing in our MapReduce framework. Many of the other commercial storage products had done this, but we observe two main issues: 1) special dedicated nodes for metadata cause a single point of failure, and 2) the vertical scale up requirement of such nodes – as the physical size of these storage nodes increases with the number of logical entities, they will need to be replaced or upgraded in terms of memory/CPU.[15]

Finally, we noticed a considerable heterogeneity across clusters. While nodes in a cluster are typically homogeneous, different clusters were setup with varying amount of resources. The workload patterns were also different, some ran server workloads, others were used for virtual desktop infrastructure (VDI), whereas some others were deployed for big data applications. Further, the variation in load across different times of day/week was significant in some clusters but not in others. Given this heterogeneity, our heuristics tended to be sub-optimal in many clusters. This motivated us to look at ML approaches to optimize the scheduling of tasks, which we discuss next.

[15]Note that the GFS file-count issue was one of the primary reasons that motivated Colossus [26].

4 Machine Learning-driven Policies

We have described so far an overview of the distributed storage fabric, and delved further into Curator's design and implementation, its tasks and policies of execution, etc. In this section, we propose our modeling strategy, based on machine learning, to improve the threshold-based policies introduced in §3.3. Note that the techniques presented here have not been deployed yet.

We motivate the need for machine learning-driven policies in §4.1. We provide background information on the general reinforcement learning framework we use for our modeling in §4.2.1, and describe with more details Q-learning in §4.2.2. We finally show the results of some experiments on Tiering, our primary use case, in §4.3.

4.1 Motivation

We observed a wide heterogeneity of workloads across our cluster deployments. Given these distinct characteristics of workloads, we noted that the threshold-based execution policies introduced in §3.3 were not optimal for every cluster, nor for individual clusters over time as some of them experienced different workloads at different times (seasonality effects). Thus, in order to efficiently execute Curators management tasks, it became necessary to build "smarter" policies that could adapt on a case-by-case basis at runtime.

The traditional way to improve performance is to use profiling in order to tune certain parameters at the beginning of cluster deployments. Nevertheless, simple profiling would not easily adapt to the varying loads (and changing workloads) our clusters are exposed to over their lifetime. We would need to run profilers every so often, and we would lose, in some sense, past knowledge. We therefore propose using a ML-based solution, which leverages the potential of statistical models to detect patterns and predict future behavior based on the past.

4.2 Background

We encompass this problem within the abstract and flexible reinforcement learning framework, explained in §4.2.1. In particular, we use the model-free popular Q-learning algorithm described in §4.2.2.

4.2.1 Reinforcement Learning (RL)

Reinforcement learning considers the problem of a learning agent interacting with its environment to achieve a goal. Such an agent must be able to sense, to some extent, the state of the environment, and must be able to take actions that will lead to other states. By acting in the world, the agent will receive rewards and/or punishments, and from these it will determine what to do next [35]. RL is about learning a policy π that maps situations to actions, so as to maximize a numerical reward signal.[16] The agent is not told which actions to take, but instead, it must discover which actions yield the most reward by trying them [42].

More formally, the agent interacts with the environment in a sequence of discrete time steps, $t = 0, 1, 2, 3....$ At each time step t, the agent senses the environment's state, $s_t \in S$, where S is the set of all possible states, and selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of all actions available in state s_t. The agent receives a reward, $r_{t+1} \in \mathbf{R}$, and finds itself in a new state, $s_{t+1} \in S$.

The goal of the agent is to maximize the total reward it receives over the long run. If the sequence of rewards received after time step t is $r_{t+1}, r_{t+2}, r_{t+3}, ...$, then the objective of learning is to maximize the *expected discounted return*. The discounted return G_t is given by:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad (1)$$

where $0 \leq \gamma \leq 1$ is called the discount factor. $\gamma = 0$ will make the agent "myopic" (or short-sighted) by only considering immediate rewards, while $\gamma \rightarrow 1$ will make it strive for a long-term high reward [42].

Given that we do not have examples of desired behavior (i.e., training data) but we can assign a measure of goodness (i.e., reward) to examples of behavior (i.e., state-action) [38], RL is a natural fit to our problem.

4.2.2 Q-Learning

Q-Learning [44] is a reinforcement learning algorithm, which falls under the class of temporal difference (TD) methods [40, 41], where an agent tries an action a_t at a particular state s_t, and evaluates its effects in terms of the immediate reward r_{t+1} it receives and its estimate of the value of the state s_{t+1} to which it is taken. By repeatedly trying all actions in all states, it learns which ones are best, i.e., it learns the optimal policy π^*, judged by long-term discounted return.

One of the strengths of this model-free algorithm is its ability to learn without requiring a model of the environment, something model-based approaches do need. Also, model-free methods often work well when the state space is large (our case), as opposed to model-based ones, which tend to work better when the state space is manageable [5].[17]

Q-Learning uses a function Q that accepts a state s_t and action a_t, and outputs the value of that state-action pair, which is the estimate of the expected value (discounted return) of doing action a_t in state s_t and then following the optimal policy π^* [21]. Its simplest form, one-step Q-learning, is given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \qquad (2)$$

where $0 \leq \alpha \leq 1$ is the learning rate, and determines to what extent new information overrides old one.

Although the learned Q-function can be used to determine an optimal action, the algorithm does not specify what action the agent should actually take [21]. There are two things that are useful for the agent to do, known as the exploration/exploitation trade-off:

- *exploit*: the knowledge that it has of the current state s_t by doing the action in $A(s_t)$ that maximizes $Q(s_t, A(s_t))$.
- *explore*: to build a better estimate of the optimal Q-function. That is, it should select a different action from the one that it currently thinks is the best.

The Q-function above can be implemented using a simple lookup table. Nevertheless, when the state-action space is large, e.g., continuous space, storing Q-values in a table becomes intractable. The Q-function needs to be approximated by a function approximator.

The compression achieved by a function approximator allows the agent to generalize from states it has visited to states it has not. The most important aspect of function approximation is not just related to the space saved, but rather to the fact that it enables generalization over input spaces [35], i.e., the algorithm can now say "the value of these kind of states is x", rather than "the value of this exact specific state is x" [1].

4.3 Use Case: Tiering

Having introduced the basics of reinforcement learning and Q-learning, in this section, we propose using the latter algorithm for deciding when to trigger Tiering. Although our initial efforts are on the tiering question of how much data to keep in SSDs, our approach generalizes to any of the threshold-based tasks described before.

4.3.1 State-Action-Reward

In order to apply Q-learning, we need to define the set of states S, the set of possible actions A, and the rewards r.

We define state s at time step t by the following tuple $s_t = (cpu, mem, ssd, iops, riops, wiops)_t$, where $0 \leq cpu \leq 100$ is the CPU usage, $0 \leq mem \leq 100$ the memory usage, $0 \leq ssd \leq 100$ the SSD usage, $iops \in \mathbf{R}$ the total IOPS, $riops \in \mathbf{R}$ the read IOPS, and $wiops \in \mathbf{R}$ the write IOPS, all at time step t, where $s_t \in S$.

We further define two possible actions for every state, either *not run* or *run* Tiering. Mathematically, the set of actions A is given by $A = \{0, 1\} \forall s_t \in S$, where 0 corresponds to *not run*, and 1 corresponds to *run* the task.

Finally, we use latency as reward. As higher rewards are better, though we prefer lower latencies, we actually

[16] The policy that achieves the highest reward over the long run is known as optimal policy, and typically denoted as π^*.

[17] We were (mainly) inclined to using Q-learning because of these two reasons.

use negative latencies,[18] i.e., the reward r at time step t is given by $r_t = -lat_t$, where $lat_t \in \mathbf{R}$ is the latency in milliseconds at time step t.

4.3.2 Function Approximator

Given that we have a continuous state space S, as defined in §4.3.1, we cannot use a tabular implementation of Q-learning. We therefore resort to a function approximator, and also gain from its advantages towards generalization.

Many approximators have been studied in the past, such as deep neural networks [28, 27], decision trees [33], linear functions [9, 43], kernel-based methods [30], etc. In this work, we choose a linear approximation. The reason behind this decision is two-fold: a) we do not have enough historical data to train more advanced methods, e.g., neural networks (§4.3.3), and b) we see that it works reasonably well in practice (§4.3.4).

4.3.3 Dataset

One key aspect of RL is related to how the agent is deployed. If there is enough time for the agent to learn before it is deployed, e.g., using batch learning with offline historical data, then it might be able to start making right choices sooner, i.e., the policy it follows might be closer to the optimal policy. Whereas if the agent is deployed without any prior knowledge, i.e., has to learn from scratch while being deployed, it may never get to the point where it has learned the optimal policy [21].

We also face this challenge as issues might arise from the time scales associated with online training from scratch in a real cluster; it may take a long time before the state space is (fully) explored. To overcome this limitation, we build a dataset from data collected from a subset of the 50 customer clusters mentioned in §3.4.1. In particular, we use ∼40 clusters, from which we have fine-grained data to represent states, actions, and rewards. The data consists of ∼32K transitions, sampled from the (suboptimal) threshold-based policy. Every cluster was using the same default thresholds described in §3.3. Even using a suboptimal policy to "bootstrap" a model can be helpful in reaching good states sooner, and is a common practice for offline RL evaluation [25].

Following ML practices, we split the dataset into training (80%) and test (20%) sets, and do 3-fold cross validation in the training set for hyper-parameter tuning. We standardize the features by removing the mean and scaling to unit variance. We train two linear models, one for each action, with Stochastic Gradient Descent (SGD) [7] using the squared loss.

4.3.4 Evaluation

In this section, we evaluate our Q-learning model, and compare it to the baseline model, i.e., the threshold-based solution. We use the same internal cluster setup as §3.4.2 to run our experiments.

We deploy our agent "pre-trained" with the dataset described in §4.3.3. Once deployed, the agent keeps on interacting with the environment, exploring/exploiting the state space. We use the popular ε-greedy strategy, i.e., with probability ε the agent selects a random action, and with probability $1-\varepsilon$ the agents selects the greedy action (the action that it currently thinks is the best). We use $\varepsilon = 0.2$ in all our experiments. It is possible to vary ε over time, to favor exploration on early stages, and more exploitation as time progresses. We leave that to future work. Further, we set $\gamma = 0.9$.

Table 1 presents results for five different workloads, described in Appendix D. We compute these numbers based on the *execution* phase of the workloads, i.e., after the pre-fill stage is done, and where the actual read/writes are executed. More results are included in Appendix E. The current experiments are within a short time frame (order of hours), we expect to see even better results with longer runs. We observe that in all of the cases our Q-learning solution reduces the average latency, from ∼2% in the *oltp-varying* workload, up to ∼20% in the *oltp-skewed* one, as well as improves the total number of SSD bytes read. We believe that further improvements could also be achieved by adding more features to the states, e.g., time of the day features to capture temporal dynamics, HDD usage, etc. We also notice that Q-learning demands more IOPS. This is the case since our solution, in general, triggers more tasks than the baseline, thus more I/O operations are performed. Overall, we see that our approach can trade manageable penalties in terms of number of IOPS for a significant improvement in SSD hits, which further translates into significant latency reductions, in most of our experimental settings.

5 Related Work

Our work borrows techniques from prior work on cluster storage and distributed systems, but we compose them in new ways to address the unique characteristics of our cluster setting. Note that our setting corresponds to clusters where cluster nodes are heterogeneous, unmodified (legacy) client applications are packaged as VMs, and cluster nodes can be equipped with fast storage technologies (SSDs, NVMe, etc.). Given this setting, we designed a system where client applications run on the same nodes as the storage fabric, metadata is distributed across the entire system, and faster storage on cluster nodes is effectively used. We now contrast our work with other related work given these differences in execution settings and design concepts.

Systems such as GFS [17] and HDFS [37] are designed for even more scalable settings but are tailored to work with applications that are modified to take advan-

[18]The choice of using negative latencies is rather arbitrary, we could have used their reciprocals instead.

Workload	**Metric**	**Policy**	
		fixed threshold	**q-learning**
oltp	Avg. Latency (ms)	12.48	**10.60**
	SSD Reads (GB)	31.68	**39.16**
	Avg. # of IOPS	**2551.54**	2903.20
oltp-skewed	Avg. Latency (ms)	18.55	**14.91**
	SSD Reads (GB)	151.99	**176.28**
	Avg. # of IOPS	**6686.90**	7221.01
oltp-varying	Avg. Latency (ms)	17.28	**16.95**
	SSD Reads (GB)	469.28	**488.17**
	Avg. # of IOPS	**7884.94**	8192.32
oltp-vdi	Avg. Latency (ms)	15.41	**13.92**
	SSD Reads (GB)	40.83	**41.27**
	Avg. # of IOPS	**4450.18**	5178.13
oltp-dss	Avg. Latency (ms)	61.65	**53.00**
	SSD Reads (GB)	4601.17	**6233.33**
	Avg. # of IOPS	**3105.60**	3239.59

Table 1: Results Summary

tage of their features (e.g., large file support, append-only files, etc.). Further, they do not distribute metadata, since a single node can serve as a directory server given the use of large files and infrequent metadata interactions. These systems do not take advantage of fast storage – all file operations involve network access and the incremental benefits of fast storage on the server side is minimal.

Cluster storage systems such as SAN and NAS also do not co-locate application processes/VMs with servers. They assume a disaggregated model of computing, wherein applications run on client machines and all the data is served from dedicated clusters [15, 34, 18, 39, 36]. These systems provide scalability benefits and a wide variety of features, such as snapshotting [14], which we borrow in our system as well. But the crucial points of differentiation are that our system uses fast local storage effectively through tiering, data migration, and disk balancing. Moreover, we believe that ours is the first system to run a continuous consistency checker which results in significant reductions in downtime.

We use a number of concepts and solutions from distributed systems: MapReduce [10] to perform cluster-wide computations on metadata, Cassandra [22] to store distributed metadata as a key-value store, Paxos [23] to perform leader election for coordination tasks. Interestingly, MapReduce is not a client application running on top of the storage system but rather part of the storage system framework itself.

In recent years, there has been an increasing amount of literature on applying ML techniques to improve scheduling decisions in a wide variety of areas, such as manufacturing [32, 31, 29, 47, 6], sensor systems [20], multicore data structures [12], autonomic computing [45], operating systems [16], computer architecture [19], etc. In Paragon [11], the authors propose a model based on collaborative filtering to greedily schedule applications in a manner that minimizes interference and maximizes server utilization on clusters with heterogeneous hardware. Their work focuses more on online scheduling of end user workloads, whereas ours, concentrates on the background scheduling of cluster maintenance tasks to improve the overall cluster performance.

Wrangler [46] proposes a model based on Support Vector Machines [8] to build a scheduler that can selectively delay the execution of certain tasks. Similar to our work, they train a linear model based on CPU, disk, memory, as well as other system-level features, in an offline-manner, and then deploy it to make better scheduling decisions. In contrast, our offline (supervised) trained model only “bootstraps” the RL one, which keeps on adapting and learning at runtime, i.e., in an online-manner. Smart Locks [13] is a self-tuning spinlock mechanism that uses RL to optimize the order and relative frequency with which different threads get the lock when contending for it. They use a somewhat similar approach, though they target scheduling decisions at a much lower level.

Perhaps the most similar line of work comes from optimal control [24, 2, 3, 4]. The papers by Prashanth et al. [2, 3] propose using RL for tuning fixed thresholds on traffic light control systems. They propose a Q-learning model that adapts to different traffic conditions in order to switch traffic light signals. We use a similar approach but in a different setting, where we learn to better schedule data migration in a multi-tier storage system.

6 Conclusions

Nowadays, cluster storage systems are built-in with a wide range of functionality that allows to maintain/improve the storage system’s health and performance. In this work, we presented Curator, a background self-managing layer for storage systems in the context of a distributed storage fabric used in enterprise clusters. We described Curator’s design and implementation, its management tasks, and how our choice of distributing the metadata across several nodes in the cluster made Curator’s MapReduce infrastructure necessary and efficient. We evaluated the system in a number of relevant metrics, and reported experiences gathered from its five-year period of construction, as well as thousands of deployments in the field. More recently, and given the heterogeneity across our clusters, we focused our attention on building “smarter” task execution policies. We proposed an initial model that uses reinforcement learning to address the issue of when Curator’s management tasks should be executed. Our empirical evaluation on simulated workloads showed promising results, achieving up to ∼20% latency improvements.

References

[1] Q-learning with Neural Networks. http://outlace.com/Reinforcement-Learning-Part-3/. Accessed: 2016-09-07.

[2] A., P. L., and Bhatnagar, S. Reinforcement Learning With Function Approximation for Traffic Signal Control. *IEEE Trans. Intelligent Transportation Systems 12*, 2 (2011), 412–421.

[3] A., P. L., and Bhatnagar, S. Threshold Tuning Using Stochastic Optimization for Graded Signal Control. *IEEE Trans. Vehicular Technology 61*, 9 (2012), 3865–3880.

[4] A., P. L., Chatterjee, A., and Bhatnagar, S. Adaptive Sleep-Wake Control using Reinforcement Learning in Sensor Networks. In *Sixth International Conference on Communication Systems and Networks, COMSNETS 2014, Bangalore, India, January 6-10, 2014* (2014), pp. 1–8.

[5] Andrew, M. Reinforcement Learning, Tutorial Slides by Andrew Moore. https://www.autonlab.org/tutorials/rl.html. Accessed: 2017-02-10.

[6] Aytug, H., Bhattacharyya, S., Kochlet, G. J., and Snowdon, J. L. A Review of Machine Learning in Scheduling. *IEEE Transactions on Engineering Management* (1994).

[7] Bottou, L. Large-scale Machine Learning with Stochastic Gradient Descent. In *COMPSTAT* (2010).

[8] Cortes, C., and Vapnik, V. Support-Vector Networks. *Mach. Learn. 20*, 3 (Sept. 1995), 273–297.

[9] Dayan, P. The Convergence of TD(λ) for General λ. *Machine Learning 8* (1992), 341–362.

[10] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[11] Delimitrou, C., and Kozyrakis, C. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, ACM, pp. 77–88.

[12] Eastep, J., Wingate, D., and Agarwal, A. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011* (2011), pp. 11–20.

[13] Eastep, J., Wingate, D., Santambrogio, M. D., and Agarwal, A. Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization. In *Proceedings of the 7th International Conference on Autonomic Computing, ICAC 2010, Washington, DC, USA, June 7-11, 2010* (2010), pp. 215–224.

[14] Edwards, J. K., Ellard, D., Everhart, C., Fair, R., Hamilton, E., Kahn, A., Kanevsky, A., Lentini, J., Prakash, A., Smith, K. A., and Zayas, E. FlexVol: Flexible, Efficient File Volume Virtualization in WAFL. In *USENIX 2008 Annual Technical Conference* (2008), ATC'08, USENIX Association, pp. 129–142.

[15] EMC. EMC Isilon OneFS: A Technical Overview, 2016.

[16] Fedorova, A., Vengerov, D., and Doucette, D. Operating system Scheduling on Heterogeneous Core Systems. In *Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures* (2007).

[17] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03, ACM, pp. 29–43.

[18] Gluster. Cloud Storage for the Modern Data Center: An Introduction to Gluster Architecture, 2011.

[19] Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08, IEEE Computer Society, pp. 39–50.

[20] Krause, A., Rajagopal, R., Gupta, A., and Guestrin, C. Simultaneous Placement and Scheduling of Sensors. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks* (2009), IPSN '09, IEEE Computer Society, pp. 181–192.

[21] L., P. D., and K., M. A. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[22] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[23] Lamport, L. Paxos Made Simple. In *ACM SIGACT News* (2001), vol. 32, pp. 51–58.

[24] Lu, C., Stankovic, J. A., Son, S. H., and Tao, G. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing 23* (2002), 85–126.

[25] Marivate, V. N. *Improved Empirical Methods in Reinforcement Learning Evaluation*. PhD thesis, 2015.

[26] McKusick, M. K., and Quinlan, S. GFS: Evolution on Fast-forward. *Queue 7*, 7 (2009), 10:10–10:20.

[27] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*. 2013.

[28] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level Control through Deep Reinforcement Learning. *Nature 518*, 7540 (02 2015), 529–533.

[29] Mönch, L., Zimmermann, J., and Otto, P. Machine Learning Techniques for Scheduling Jobs with Incompatible Families and Unequal Ready Times on Parallel Batch Machines. *Eng. Appl. Artif. Intell. 19*, 3 (Apr. 2006), 235–245.

[30] Ormoneit, D., and Sen, S. Kernel-Based Reinforcement Learning. In *Machine Learning* (1999), pp. 161–178.

[31] Priore, P., De La Fuente, D., Gomez, A., and Puente, J. A Review of Machine Learning in Dynamic Scheduling of Flexible Manufacturing Systems. *Artif. Intell. Eng. Des. Anal. Manuf. 15*, 3 (June 2001), 251–263.

[32] Priore, P., de la Fuente, D., Puente, J., and Parreño, J. A Comparison of Machine-learning Algorithms for Dynamic Scheduling of Flexible Manufacturing Systems. *Eng. Appl. Artif. Intell. 19*, 3 (Apr. 2006), 247–255.

[33] Pyeatt, L. D., and Howe, A. E. Decision Tree Function Approximation in Reinforcement Learning. Tech. rep., Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models, 1998.

[34] Rodeh, O., and Teperman, A. zFS - A Scalable Distributed File System Using Object Disks. In *IEEE Symposium on Mass Storage Systems* (2003), IEEE Computer Society, pp. 207–218.

[35] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 2 ed. Pearson Education, 2003.

[36] SCHMUCK, F., AND HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[37] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), MSST '10, IEEE Computer Society, pp. 1–10.

[38] SI, J., BARTO, A. G., POWELL, W. B., AND WUNSCH, D. *Handbook of Learning and Approximate Dynamic Programming (IEEE Press Series on Computational Intelligence)*. Wiley-IEEE Press, 2004.

[39] SUN. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System, 2007.

[40] SUTTON, R. S. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, 1984. AAI8410337.

[41] SUTTON, R. S. Learning to Predict by the Methods of Temporal Differences. In *MACHINE LEARNING* (1988), Kluwer Academic Publishers, pp. 9–44.

[42] SUTTON, R. S., AND BARTO, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, 1998.

[43] TSITSIKLIS, J. N., AND ROY, B. V. An Analysis of Temporal-Difference Learning with Function Approximation. Tech. rep., IEEE Transactions on Automatic Control, 1997.

[44] WATKINS, C. J. C. H., AND DAYAN, P. Technical Note: Q-Learning. *Mach. Learn. 8*, 3-4 (May 1992), 279–292.

[45] WHITESON, S., AND STONE, P. Adaptive Job Routing and Scheduling. *Eng. Appl. Artif. Intell. 17*, 7 (Oct. 2004), 855–869.

[46] YADWADKAR, N. J., ANANTHANARAYANAN, G., AND KATZ, R. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SOCC '14, ACM, pp. 26:1–26:14.

[47] YIH, Y. Learning Real-Time Scheduling Rules from Optimal Policy of Semi-Markov Decision Processes. *International Journal of Computer Integrated Manufacturing* (1992).

A Metadata Maps Accessed by Curator

Category	Task	Metadata Maps		
		vDiskBlock	extentID	extentGroupID
Recovery	**DF**			x
	FT			x
Data Migration	**T**[19]			x
	DB			x
Space Reclamation	**GC**	x	x	x
	DR	x	x	x
Data Transformation	**C**			x
	EC			x
	DD	x	x	x
	STR	x		

Table 2: Metadata Tables accessed by Curator tasks

B Benefits/Costs Curator ON/OFF

	Metric (Average)	**Curator**	
		OFF	**ON**
Benefits	Latency (ms)	61.73	**12.3**
	Storage Usage (TB)	3.01	**2.16**
Costs	CPU Usage (%)	**14**	18
	Memory Usage (%)	**14.703**	14.705
	# of IOPS	**1173**	1417

Table 3: Benefits/Costs Summary

C OLTP Workloads

Each OLTP workload is composed of two sections, *Data* and *Log*, which emulates the actual data space and log writing separation in traditional Database Management Systems. The *Data* section performs random reads and writes, 50% reads and 50% writes. Further, 10% of its I/O operations, either reads or writes, have 32k block sizes, and 90% 8k blocks. On the other hand, the *Log* section is write only, 10% of the writes are random, and all operations are 32k. The three workloads (*small*, *medium*, *large*) only differ on how much data they read/write, and the number of IOPS, as shown in Table 4.

Workload	Data		Log	
	Size (GB)	IOPS	Size (GB)	IOPS
small	800	4000	16	200
medium	1120	6000	16	300
large	1120	8000	12	400

Table 4: OLTP Workloads

[19]Also accesses an additional map, as discussed in 3.2.2.

D ML-driven Policies Workloads

We use the following five workloads to test our ML-driven policies:

- *oltp*: same as the OLTP large workload shown in Table 4, with which we intend to simulate a standard database workload.
- *oltp-skewed*: similar to OLTP large, but here the *Data* section is read only, and performs 8k block random reads according to the following distribution: 90% of the accesses go to 10% of the data. It has a working set size of 4480 GB, and performs 32000 I/O operations per second. With this workload we aim to better understand the effects of hot data skewness.
- *oltp-varying*: Alternates between the OLTP medium and small workloads shown in Table 4 every 20 minutes. In this case, we aim to simulate varying loads of the same type of workload within a cluster.
- *oltp-vdi*: Runs the OLTP large workload in one node while the remaining nodes execute VDI-like workloads in 100 VMs each. A VDI-like workload consists of a working set size of 10 GB, split into *Read* and *Write* sections. 80% of the reads in the *Read* section are random, 10% of the read operations have 32k block sizes, and 90% 8k blocks. Regarding the *Write* section, only 20% of the writes are random, and all have 32k block sizes. The writes are done in the last 2 GB of data, whereas the reads range spans the first 8 GB. The total number of IOPS per VM is 26 (13 each section). Here, the idea is to simulate (concurrent) heterogeneous workloads within a cluster.
- *oltp-dss*: Alternates between the OLTP medium and a DSS (Decision Support System) workload every 20 minutes. As DSS is a type of DB workload, it also has *Data* and *Log* sections, but as opposed to an OLTP workload, the *Data* section of a DSS-like workload performs only reads. Here, such section has a working set size of 448 GB, 100% of the reads are sequential, the read operations size is 1 MB, and the total number of IOPS is 2880. As regards to the *Log* section, the working set is 16 GB, 10% of the writes are random, the block sizes are 32k, and the rate of IOPS is 800. In this case, we attempt to simulate that the workload itself changes over time.

E ML-driven Policies Results

Impact of Tiering on Total Number of SSD Reads

Figure 7 shows the evolution of SSD reads for the *oltp* and *oltp-skewed* workloads. We observe that our method based on Q-learning performs on average more SSD reads (∼8 GB and ∼24 GB respectively) than the baseline for both workloads, which is a consequence of performing more Tiering operations during periods when the offered load from applications is low.

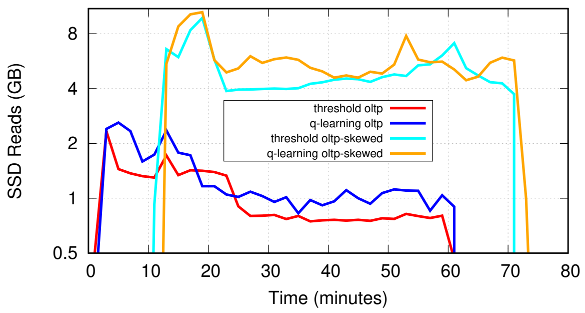

Figure 7: SSD Reads in the *oltp* and *oltp-skewed* workloads

Q-Learning in Action

We now provide performance data corresponding to a sample scenario and illustrate how the Q-Learning model operates in practice. Figure 8 shows the IOPS, latency, and scheduling decisions made while executing the *oltp-dss* workload with our ML-based scheduler. We only plot the *execution* phase of the workload, where the actual operations are executed. Our system polls the state of the cluster every 30 seconds, if it had not triggered Tiering recently, in order to assess whether Tiering should be performed. After a Tiering task is triggered, we wait for 5 minutes before making a new decision, as we do not want to schedule Tiering tasks back-to-back. Regarding the scheduling plot, we not only include the two choices the algorithm makes, *run* and *not run*, but also differentiate whether its decision was due to exploitation (solid lines) or exploration (dashed lines).

The workload keeps on alternating between the OLTP medium and DSS workloads every 20 minutes, as described in Appendix D. It starts with the former, continues with the latter, and so on. We observe that the OLTP medium workload, in general, demands more IOPS than the DSS one, and also achieves lower latencies (cyclic behavior).

At the beginning, even with high IOPS and low latency, the algorithm thinks that the best option is to trigger Tiering (0-20 minutes). When the DSS workload commences (early 20s), the algorithm still keeps scheduling Tiering tasks. In this case, it makes more sense as the cluster utilization is not too high but the latency is. Around minute 44, the algorithm explores the state space by not running Tiering (dashed red line that almost overlaps the solid red ones that follow). Given that this exploration seems to have found a "nice state" with low latency, it considers the best option is to not to run Tiering (first chunk of solid red lines around minute 45). Note that given our 30 seconds polling interval when we do not run Tiering, these lines seem to overlap.

At approximately the 47th minute, the algorithm per-

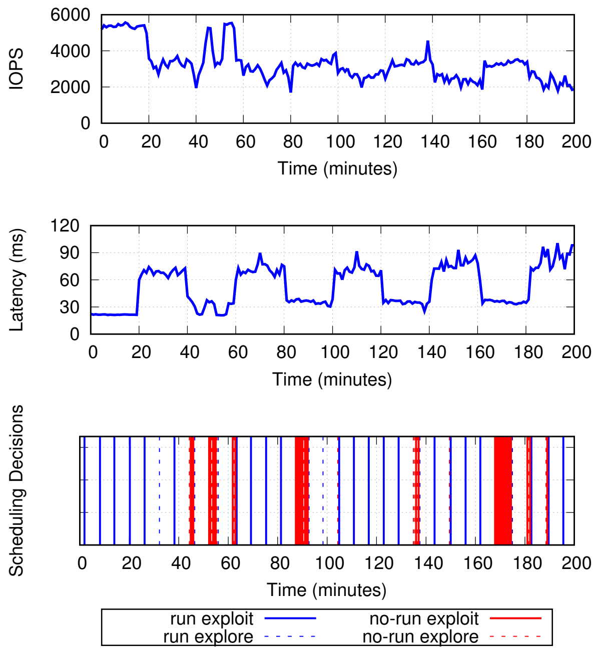

Figure 8: IOPS/Latency/Scheduling Decisions in the *oltp-dss* workload using Q-learning

forms an exploration that triggers Tiering (dashed blue line). It does not work out, as later on, the best decisions are still not to run Tiering (solid red lines around minutes 52-54). Around minute 63, when DSS commences again, the algorithm thinks it is best to run Tiering. At this point, the cluster is not very utilized, i.e., low IOPS, but the latency is high.

The key thing to notice is that the algorithm seems to be learning that when the cluster is highly utilized (high IOPS) and the latency is low, it should not trigger Tiering. During the first period (0-20mins), it was not aware of that, thus it ran Tiering, but later on, it started to figure it out (e.g., 40-60mins and 80-100mins periods). Even more noticeable is between the period 160-180mins, where we observe *many* solid red lines (which appears as a single thick one due to the 30 seconds interval). The 120-140mins period is somewhat surprising. We would have expected more solid red lines there, but they only start appearing towards the end of the period. We believe the algorithm makes early mistakes (minutes 123 and 128), and given that we wait for 5 minutes after running Tiering, it can only realize later on ($\sim$133), where it decides that it is actually better not to run.

Acknowledgments

We are grateful to our reviewers and colleagues for their help and comments on earlier versions of this paper. Special thanks to our shepherd Mosharaf Chowdhury for his valuable feedback. We are also grateful to Hinal Gala, Peihong Huang, and Jason Chan from the Curator QA team for providing very useful information for testing the system. Further, thanks to Chris Wilson from the Performance team for his help in generating the workloads. This work was supported by the National Science Foundation (CNS-1318396, CNS-1420703, CNS-1616774, and CNS-1614717). One of the authors was supported by the Argentine Ministry of Science, Technology and Productive Innovation with the program BEC.AR.