Automating Distributed Tiered Storage Management in Cluster Computing

Herodotos Herodotou
Cyprus University of Technology
Limassol, Cyprus
herodotos.herodotou@cut.ac.cy

Elena Kakoulli
Cyprus University of Technology
Limassol, Cyprus
elena.kakoulli@cut.ac.cy

ABSTRACT

Data-intensive platforms such as Hadoop and Spark are routinely used to process massive amounts of data residing on distributed file systems like HDFS. Increasing memory sizes and new hardware technologies (e.g., NVRAM, SSDs) have recently led to the introduction of storage tiering in such settings. However, users are now burdened with the additional complexity of managing the multiple storage tiers and the data residing on them while trying to optimize their workloads. In this paper, we develop a general framework for automatically moving data across the available storage tiers in distributed file systems. Moreover, we employ machine learning for tracking and predicting file access patterns, which we use to decide when and which data to move up or down the storage tiers for increasing system performance. Our approach uses incremental learning to dynamically refine the models with new file accesses, allowing them to naturally adjust and adapt to workload changes over time. Our extensive evaluation using realistic workloads derived from Facebook and CMU traces compares our approach with several other policies and showcases significant benefits in terms of both workload performance and cluster efficiency.

PVLDB Reference Format:
Herodotos Herodotou and Elena Kakoulli. Automating Distributed Tiered Storage Management in Cluster Computing. *PVLDB*, 13(1): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3357377.3357381

1. INTRODUCTION

Data-intensive analytic applications for business intelligence, social network analysis, and scientific data processing are routinely executed on Big Data platforms such as Hadoop YARN [43] and Spark [48], while processing massive amounts of data residing in distributed file systems such as HDFS [40]. Such applications tend to spend significant fractions of their overall execution in performing I/O [47]. Larger memory sizes as well as new hardware technologies such as Non-Volatile RAM (NVRAM) and Solid-State Drives (SSDs) are now commonly utilized for increasing I/O performance. In particular, in-memory distributed file systems like Alluxio [4] and GridGain [21] are used for storing or caching HDFS data in memory. HDFS has also added support for caching input files internally [22]. NVRAM and SSDs have been used as the storage layer for distributed systems [13, 25, 31] as well as in shared storage systems (e.g., ReCA [38], Hermes [28]). Recently, the OctopusFS distributed file system [27] introduced fine-grained tiering in compute clusters via storing file replicas on the various storage media (e.g., memory, SSDs, HDDs) that are locally attached on the cluster nodes. At the same time, HDFS generalized its architecture to support storage media other than HDDs, including memory and SSDs [1].

As multiple storage tiers are added into distributed file systems (and storage systems in general), *the complexity of data movement across the tiers increases significantly*, making it harder to take advantage of the higher I/O performance offered by the system [28]. Most aforementioned systems expose APIs for data caching or movement to application developers and data analysts. For example, an HDFS application can issue requests to cache files and directories. However, when the cache gets full, no additional caching requests will be served until the application *manually* uncaches some files [22]. Similarly, OctopusFS offers a placement policy for determining how to initially store the data across the storage tiers but lacks any features for automatically moving data afterwards. Other systems like Alluxio [4] and GridGain [21] implement basic policies for removing data from memory when full, such as LRU (Least Recently Used) [2]. However, such policies are known to under-perform in the big data setting as they were initially designed for evicting fixed-size pages from buffer caches [5, 2]. Furthermore, these systems do not offer any cache admission policies; i.e., they will place all data in the cache upon access, without any regards for the current state of the system, the data size, or any workload patterns.

Overall, the lack of automated data movement across storage tiers places a significant burden on users or system administrators, who are tasked with optimizing various types of data analytics workloads [28, 34]. The analysis of production workloads from Facebook, Cloudera, and MS Bing [5, 9] has shown that many jobs (small and large alike) exhibit data re-access patterns both in the short-term (within hours) and the long-term (daily, weekly, or monthly). Thus, identifying the reused data and keeping them in higher tiers can yield significant performance benefits [5]. In addition, data access patterns can change over time with the addi-

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3357377.3357381

tion and removal of users and jobs, leading to an increased need for adapting to these changes accurately and efficiently [34]. Hence, it is imperative for tiered storage systems to include automated data management capabilities for improving cluster efficiency and application performance.

In this paper, we propose a *general framework for automated tiered storage management in distributed file systems*. Specifically, our framework can be used for orchestrating data management policies for adaptively deciding (i) when and which data to retain or move to higher storage tiers for improved read performance and (ii) when and which data to move to lower tiers for freeing scarce resources. We show its generalization by implementing several conventional cache eviction and admission policies [2], related policies from recent literature [5, 16], as well as our own policies.

Furthermore, we propose the use of *machine learning (ML) for tracking and predicting file access patterns* in the system. In particular, we employ light-weight *gradient boosted trees* [8] to learn how files are accessed by the current workload and use the generated models to drive our automated file system policies. Our approach uses *incremental learning* to dynamically refine the models with new file accesses as they become available, allowing the models to naturally adjust and adapt to workload changes over time.

Our work lies at the intersection of distributed storage systems, hierarchical storage management (HSM) solutions, and caching, with a humbling amount of related work. Yet, to the best of our knowledge, *we propose and implement a new, fully automated, and adaptive approach to tiered storage management in distributed file and storage systems.* Our ML-based approach is also different from previous approaches in HSM and caching (a detailed comparison is provided in Section 2). Finally, we have implemented our approach in an existing distributed file system, namely OctopusFS [27], which is a backwards-compatible extension of HDFS [40].

In summary, the key contributions of this paper are:

1. The design and implementation of a general framework for automatically managing storage tiers in distributed file systems.
2. An online, adaptive machine learning-based policy for predicting file access patterns and dynamically moving data among storage tiers.
3. An extensive evaluation using realistic workloads derived from Facebook and CMU traces, showcasing significant benefits for workload performance and cluster efficiency.

The paper is organized as follows. Section 2 presents an overview and comparison with existing related work. Section 3 discusses the proposed tiered storage management framework and Section 4 formulates the ML models used for predicting file access patterns. Sections 5 and 6 outline several policies for moving files down and up the storage hierarchy, respectively. The experimental evaluation is presented in Section 7, while Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section, we provide a brief background of storage tiering and compare previous research with our work.

2.1 Distributed File Systems and Tiering

Distributed file systems like *HDFS* [40] serve the current generation of Big Data platforms [43, 48] via storing files on locally-attached hard disk drives (HDDs) on the cluster nodes. The files are typically broken down into large blocks (e.g., 128MB), which are then replicated and distributed in the cluster, as shown in Figure 1(a). HDFS has recently taken significant steps toward tiered storage via (i) enabling the HDFS cache [22], and (ii) supporting heterogeneous storage types such as SSDs and memory [1]. In the former, there is no support for automatically caching or uncaching files, while in the latter, there is support for a limited number of static policies for storing files on specific tiers [1].

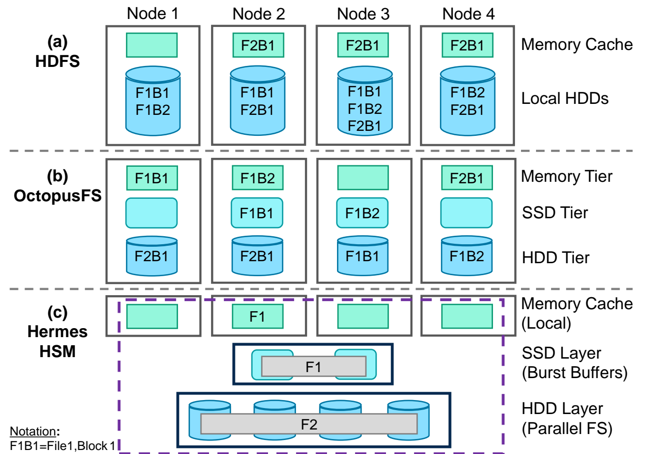


Figure 1: Data placement for three tiered storage systems

hatS [29] and *OctopusFS* [27] extended HDFS to support fine-grained storage tiering based on which file blocks are replicated and stored across both the cluster nodes and the storage tiers (see Figure 1(b)). hatS proposed a simple rule-based data placement policy, whereas OctopusFS developed one based on a multi-objective problem formulation for deciding how the file blocks should be distributed in the cluster in order to maximize performance, fault tolerance, and data and load balancing. However, neither hatS nor OctopusFS support any policies for automatically moving file replicas between the storage tiers. To exploit larger memories, in-memory file systems such as *Alluxio* [4] and *GridGain* [21] can be used for storing or caching data in clusters. In the Spark ecosystem, *Resilient Distributed Datasets (RDDs)* are a distributed memory abstraction that allows applications to persist a specified dataset in memory for reuse, and use lineage for fault tolerance [48]. While conventional cache eviction policies such as LRU are used in these systems, they rely on the user to manually cache the data.

PACMan [5] is a memory caching system that explores memory locality of data-intensive jobs. PACMan implements two eviction policies: (i) *LIFE* that minimizes job completion time by prioritizing small inputs, and (ii) *LFU-F* that maximizes cluster efficiency by evicting less frequently accessed inputs. However, PACMan does not allow applications to specify hot data in memory for subsequent efficient accesses and does not implement cache admission policies. *Big SQL* [16] is an SQL-on-hadoop system that utilizes HDFS cache for caching table partitions. Big SQL presents two algorithms, namely *SLRU-K* and *EXD*, that explore the tradeoff of caching objects based on recency and frequency of data accesses. The two algorithms drive both cache eviction and admission policies but, unlike our approach, do not learn from file access patterns as the workload changes.

2.2 Hierarchical Storage Management

Hierarchical storage management (HSM) solutions store data across storage layers that typically consists of arrays of tapes, compressed disks, and high-performance disks, while more recently memory and NVRAM are also exploited [26, 28] (see Figure 1(c)). HSM supports both *tiering* (i.e., a file will only reside on one of the storage layers) and *caching* (i.e., a copy of a file will be moved to the cache), but, unlike distributed file systems, HSM does not offer any locality or storage-media awareness to higher-level applications.

The process of moving or copying files from one storage layer to another is typically based on predefined policies and parameters (e.g., low/high thresholds for disks capacities) [26, 28]. The *ReCA* storage system uses the SSD layer as a dynamic cache [38]. Based on the incoming application I/Os, ReCA categorizes the workload in one of five types, and reconfigures the cache when detecting a change in the workload. The *DataSpaces* framework [26] exploits memory and SSDs to support dynamic data staging in HSM, driven by user-provided hints about expected data read patterns. *Hermes* adds memory and NVRAM in the storage hierarchy and proposes three policies that can be manually configured by the user. The cost models and tiering mechanisms used in prior approaches in HSM cannot be directly applied to analytics applications since they are designed to handle block level I/Os (e.g., 4–32 KB) for POSIX-style workloads (e.g., server, database, file systems) [11]. In addition, our main approach for automatically moving data across the storage hierarchy is not based on parameter-driven or user-defined policies but rather on machine learning.

2.3 Caching

Caching is a well-studied problem that appears in various contexts and discussed extensively in several surveys [36, 2]. We offer a quick overview of the area and highlight the most closely related work. For CPU caches, virtual memory systems, and database buffer caches, there is extensive work on *cache eviction* policies (e.g., LRU, LFU, ARC, MQ), which are classified as a) recency-based, b) frequency-based, c) size-based, d) function-based, and e) randomized [36]. Other policies found in main memory databases attempt to identify hot and cold data, also based on access frequencies [30, 17, 14]. Another recent policy augmented the Marker caching algorithm [15] with a machine learned oracle for improving its competitive ratio [33]. Unlike our approach, these policies operate on fixed-size pages and assume that every accessed page will be inserted into the cache.

Many caching policies have been developed for web caches that operate on variable size objects, including SIZE, Hyper-G, Greedy-Dual-Size, and Hybrid [2]. Most of these policies have been designed for improving the hit ratio in web accesses but that does not necessarily improve the performance in storage systems [38]. *Machine learning* techniques such as logistic regression, artificial neural networks, genetic algorithms, random forests, and others have also been used for developing more intelligent web caching policies [2, 44, 7, 3, 41]. However, most of these approaches try to identify and predict relationships between web objects; for example, a visit to web page X is typically followed by a visit to web page Y. Other approaches try to capture associations between file attributes (e.g., owner, creation time, and permissions) and properties (e.g., access pattern, lifespan, and size) [34]. Another recent approach used neural networks to analyze the inter-relationships among web requests for making caching decisions [24]. However, such relationships and associations are not expected to be present in big data analytics workloads and, hence, are not applicable in our setting. Reinforcement learning has also been attempted by using multiple experts to select the best cache eviction policy to use at any given time, but these approaches are known to outperform only the static policies (e.g., LRU, LFU) and are computationally and memory expensive [6, 20, 45].

3. TIERED STORAGE MANAGEMENT

The main operations of distributed file systems, such as HDFS [40] and OctopusFS [27], are to store and retrieve *files*, which are broken into large *blocks*. In HDFS, blocks are replicated 3 times by default and distributed across cluster nodes. Replication offers three main benefits: (1) it prevents data loss due to disk or node failures; (2) it enables higher I/O rates since the same block can be read in parallel (from different replicas); and (3) it increases the chances of compute-data co-location [40]. With caching, extra block replicas are created in memory for improving read I/O performance and reducing read latencies [22].

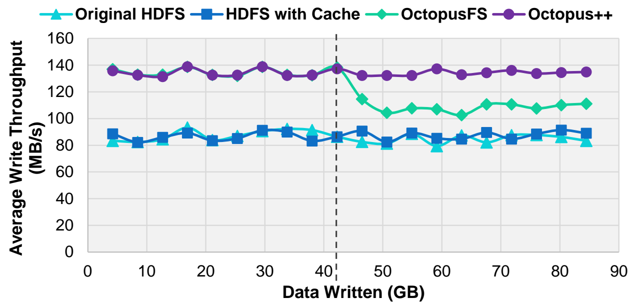
In OctopusFS, the blocks are replicated and distributed both across nodes and across storage tiers based on the decision of an automated block placement policy. For example, a block may have 1 replica in memory, 1 on SSD, and 1 on HDD on three different nodes (see F1B1 in Figure 1(b)). Storing data across tiers introduces two additional benefits: (1) it increases the overall I/O performance and the cluster resource utilization for both write and read operations; and (2) it enables higher-level systems to make both locality-aware and tier-aware scheduling decisions [27].

3.1 Effect of Tiered Storage in DFSs

In order to study the effects of storing and retrieving block replicas to and from multiple storage tiers, we used the *DFSIO* benchmark [40] to write and read 84GB of data in a 12-node cluster with 3 storage tiers: memory, SSD, and HDD. The experimental setup is detailed in Section 7. We repeated this experiment in four scenarios:

- **Original HDFS**, storing 3 replicas on HDDs across 3 different nodes;
- **HDFS with Cache**, storing 1 additional replica in memory on a node that already contains 1 HDD replica;
- **OctopusFS**, which uses a policy to determine the best node and storage tier for storing each of 3 replicas;
- **Octopus++**, our extension of OctopusFS with smart ML-based policies that dynamically move existing replicas between storage tiers.

The average write and read throughput per node for the four scenarios are shown in Figures 2(a) and 2(b), respectively. We focus on the average throughput per node because the total bandwidth is linear with the number of nodes [40]. Comparing the I/O throughput between *Original HDFS* and *OctopusFS* during the generation of the first 42GB of data, we observe that *OctopusFS* achieves a 54% increase on average write throughput over *Original HDFS* (from 87 to 135MB/s). Up until that point, *OctopusFS* places the three replicas of each block on the three different tiers – memory, SSD, and HDD. Placing replicas on multiple tiers has a modest effect on write performance since the data are written in



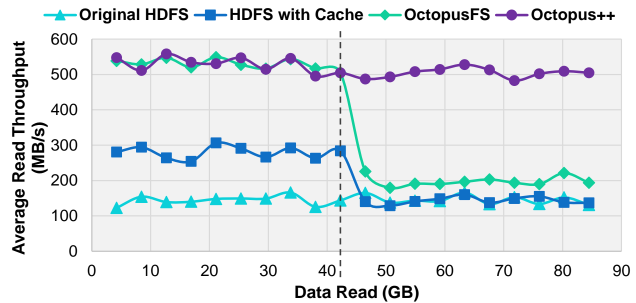


Figure 2: (a) Average write and (b) read throughput per node for HDFS, HDFS with Cache, OctopusFS, and Octopus++

a pipeline and the performance is bottlenecked by storing 1 replica on HDD. However, by placing 1 replica in memory and 1 on SSD, the average read throughput increases 3.7x over storing all replicas on HDDs. Enabling caching in HDFS has no effect on the observed write throughput as caching takes place asynchronously, but the introduction of 1 cached replica in memory leads to a 2x higher read throughput compared to *Original HDFS*.

The trends in Figures 2(a) and 2(b) change after the generation of 42GB of data because the aggregated memory available to the file systems is exhausted. After that point, *OctopusFS* places either 1 replica on SSD and 2 replicas on HDD or the other way around, reducing the I/O benefits to only 28% and 36% of average write and read throughput over *Original HDFS*, respectively. Enabling caching in HDFS has no effect on write or read throughput anymore as there is no space available to cache any data in memory.

The above results showcase the benefits of tiered storage in distributed file systems in terms of improved I/O performance and better resource utilization. At the same time, they highlight some important issues from making static data placement decisions. First, as time goes by, memory fills up with files that may no longer be needed, which prevents newer or more important files from getting stored there. On the other hand, some files may be accessed more frequently than others and, hence, it would be more beneficial to move or copy them in memory. Finally, several previous studies [5, 35] have shown that file access patterns evolve over time so a file system must be able to adjust its behavior in order to avoid big variations in performance, as the ones observed in Figure 2. Octopus++, which utilizes ML-based policies (see Section 4) for automatically moving file replicas up and down the storage tiers, is able to maintain the same performance as OctopusFS for both writes and reads *throughout the entire experiment* (see Figure 2).

3.2 Adaptive Tiered Storage Management

Based on the above results and discussion, it is essential to build *mechanisms and algorithms for automatically moving data across the storage tiers over time* in order to avoid wasting resources and missing various optimization opportunities. We have separated the data movement into two categories based on the way the data moves betweens tiers and formulated the following two definitions:

DEFINITION 1. ***Replication downgrade*** *is the process of (i) moving a file replica from a higher storage tier to a lower one, or (ii) deleting a file replica.*

DEFINITION 2. ***Replication upgrade*** *is the process of (i) moving a file replica from a lower storage tier to a higher one, or (ii) creating a new file replica.*

We have identified four important decision points that are necessary for guiding replication downgrades or upgrades:

1. When to start the downgrade (upgrade) process
2. Which file to downgrade (upgrade)
3. How to downgrade (upgrade) the selected file
4. When to stop the downgrade (upgrade) process

These 4 decision points constitute a *generalization* of both conventional cache management as well as tiering and caching in hierarchical storage management (HSM). For example, consider a database buffer cache that stores data blocks read from disk. The "downgrade" (i.e., eviction) process starts when the buffer cache gets full (decision #1). A data block is selected based on a cache eviction policy such as LRU (decision #2) and is deleted from the cache (decision #3). The eviction policy is invoked again until there is enough room in the buffer cache to fit the new data (decision #4). In an HSM system, when the access frequency of a file f becomes higher than a threshold (decision #1), f (decision #2) is moved from the HDD layer to the SSD layer (decision #3).

In a multi-tier DFS, there are multiple interesting options for all four decision points and by treating them differently we get better separation of concerns. For instance, the system does not need to wait until a storage tier is full to initiate a downgrade process; rather, it can start it proactively in order to overlap the downgrade of a file with the creation of new files. Similarly, the system does not need to wait until a file is accessed to upgrade it but can start moving it to a higher tier if it expects the file to be used in the near future. In addition, when the system decides to downgrade or upgrade a file, it then needs to decide whether to delete, move, or copy the file and where. All these decisions will be handled through *pluggable downgrade and upgrade policies*, elaborated in Sections 4-6. The decisions are made at the granularity of files (rather than blocks) since previous work [5, 16] has shown that performance improvement is attained only when entire files are present in a higher tier (called the "all-or-nothing" property in [5]).

3.3 System Design and Implementation

HDFS, and by extension OctopusFS, uses a multi-master/worker architecture that consists of *Masters*, *Workers*, and *Clients*. Each Master contains (i) the *FS Directory*, which offers a traditional hierarchical file organization and operations; (ii) a *Block Manager*, which maintains the mapping from file blocks to nodes and storage tiers; and (iii) a *Node Manager*, which contains the network topology and maintains node statistics. The Workers are responsible for (i) storing and managing file blocks on the storage media; (ii)

serving read and write requests from Clients; and (iii) performing block creation, deletion, and replication upon instructions from the Masters. The Client exposes APIs for all typical file system operations such as creating and deleting directories or writing and reading files.

We have extended OctopusFS by adding a *Replication Manager* in the Masters for orchestrating the automatic data movement across the storage tiers, based on the decisions of pluggable downgrade and upgrade policies. The policies implement 4 main methods that correspond to the 4 core decision points and callback methods for receiving notifications after a file creation, access, modification, or deletion. Finally, the policies have access to file and node statistics maintained by the system in order to make informed decisions. In addition, a *Replication Monitor* is responsible for handling the data movement requests from the Replication Manager, as well as monitoring the overall system for any over- or under-replicated blocks. We also modified the Workers to enable the transfer of blocks between storage tiers efficiently. We did not modify the Client and kept it backward compatible with OctopusFS and HDFS. For ease of reference, we call our version of the system *Octopus++*.

Even though we implemented our approach in OctopusFS, it is not specific to the internal workings of OctopusFS. We are confident our framework can be easily implemented in (i) HDFS with caching; (ii) an in-memory distributed file system (e.g., Alluxio, GridGain); or (iii) an HSM system (e.g., ReCA, Hermes) for deciding when and what data to move across the available storage tiers.

4. FILE ACCESS PATTERN MODELING

Previous studies have shown that file access behavior is not random; it is driven by application programs and analytical needs, leading to various types of data re-access patterns [5, 9]. For example, some data may be shared by multiple applications and reused for a few hours before becoming cold, while others are reused for longer periods such as days or weeks. In addition, data access patterns tend to evolve over time as users and applications are added and removed from a cluster [34]. The above two observations have motivated our approach of modeling file access patterns and creating a *feature-based classifier* to predict whether a file will be accessed in the near future (and hence, should be upgraded) or it has become cold (and hence, should be downgraded). The overall approach comprises data preparation, normalization, online incremental training, and binary classification with gradient boosted trees (discussed next).

4.1 Training Data Preparation

The three most important factors that can influence a replacement or prefetching process in a cache are: (i) *recency*, i.e., the time of the last file access; (ii) *frequency*, i.e., the number of accesses to the file; and (iii) the *size* of the file [2]. All typical file systems already maintain each file's size, last access time, and creation time. Even though it would be easy to keep track of access frequencies, that would not reveal any information regarding potential re-access patterns. Thus, we maintain the last k access times for each file (overheads are discussed in Section 7.6), which, combined with the file size and creation time, constitute our input data.

The next data preparation step in a classification pipeline is the generation of the *feature vectors* $\vec{x}_i$ and the *class labels* y_i, shown in Figure 3. Timestamps are not good feature

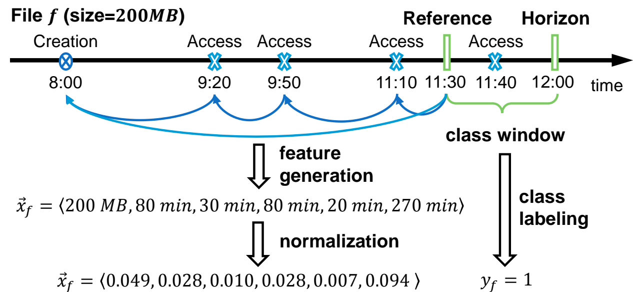


Figure 3: Pipeline for training data preparation

candidates for machine learning since their value constantly increases over time. Hence, we use the timestamps to generate *time deltas*, that is, the time difference between (i) two consecutive time accesses, (ii) the oldest time access and creation time, (iii) a reference time and the most recent access, and (iv) a reference time and creation time. A *reference time* is a particular point in time chosen to separate the perceived "past" from the perceived "future". The past represents when and how frequently a file has been accessed and, thus, it is used to generate the feature vectors $\vec{x}_i$. On the other hand, the future shows whether the file will be re-accessed in a given forward-looking *class window*, which is used to generate the class label y: if a file is accessed during the window, then $y = 1$; otherwise $y = 0$. Note that by sliding the reference time in the time axis, we can generate multiple training points (i.e., feature vectors and corresponding class values) based on the access history of a single file.

The final step in our data preparation involves normalizing the features by rescaling all the values to lie between 0 and 1. Normalization is performed by diving the time deltas by a maximum time interval (e.g., 1 month), which is useful for avoiding outliers from situations where a file was not accessed for a long time. Figure 3 shows a complete example of the training data preparation process. As the file is accessed 3 times before the chosen reference time, the feature vector will contain 5 normalized time deltas as explain above and one file size feature. The remaining $k - 3$ access-based features are encoded as missing values. The class value is set to 1 since the file is accessed within the class window.

4.2 Incremental Learning

In supervised learning, data $D = ((\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n))$ consisting of feature vectors $\vec{x}_i$ and class values y_i are used to infer a model $M \approx p(y|\vec{x})$. Unlike traditional batch (offline) learning, which trains a model from a fixed training dataset D, *incremental learning* dynamically refines a model with new data points as they become available gradually over time [18]. The benefits are that it is unnecessary to determine a fixed training window and that the model naturally adjusts and adapts to changes over time.

The key requirement for incremental learning is the ability to generate training data (i.e., both feature vectors and class values) efficiently while the system is running. Our data preparation pipeline (recall Section 4.1) makes this possible as follows. Suppose our goal is to build a model that predicts whether some file will be re-accessed in a given class window of size w (e.g., in the next 30 minutes). The system, at the current point in time t_c and for some file f, can generate a training data point via: (1) setting the reference time $t_r = t_c - w$ (i.e., setting the reference time 30 minutes

before the current time); (2) generating the feature vector based on the file size, creation time, and access times before t_r as explained in Section 4.1; and (3) determining the class label based on whether the file was accessed during the time interval between t_r and t_c. By repeating the above three steps periodically for a sample of (or all) the files in the file system, we can generate continuous data points for (re-) training the model over time. Note that we can efficiently compute the feature vectors for each file incrementally since time deltas between consecutive file accesses do not change over time. Finally, in order to ensure the generation of positive training data (i.e., data points with class value $y = 1$), we also repeat the above three steps right after a file is accessed (but only for that file).

4.3 Learning Model Selection

The following requirements for a machine learning model are desirable, since we plan to use the model in a real system:

- **Accurate:** The model must be able to accurately predict whether a file will be accessed soon or not be accessed for some time.
- **Efficient:** Both model training and predictions must be inexpensive in terms of computational and storage requirements.
- **Adaptable:** The model must support incremental learning and efficiently adapt to new workloads patterns.

As a learning model we selected *XGBoost* [8], a state-of-the-art gradient boosting tree algorithm that satisfies all three requirements. The XGBoost model has the form of an ensemble of weak models (single trees), which is trained following a stage-wise procedure under the same (differentiable) loss function [8]. XGBoost is very effective in practice and has been used in multiple winning solutions for regression, classification, and ranking in recent ML competitions [19]. As an ensemble model, XGBoost is more complex and less understandable than simple regression or rule-based models. In response, various methods have been proposed recently to help admins compute feature importance measures and understand the reasons of individual predictions [32, 37].

We also considered other well-known classifiers such as Naive Bayes, Bayesian Belief Networks (BBN), Support Vector Machines (SVM), and Artificial Neural Networks (ANN), but each failed to satisfy some of our needs. Naive Bayes assumes that attributes are conditionally independent of one another, and thus cannot be used effectively to learn a sequence of access patterns. BBNs model attribute dependence in networks, but require a priori knowledge about the structure of the network, which is exactly what we are trying to determine [34]. Finally, we empirically found SVM and ANN to have a much higher cost in terms of training time (up to two orders of magnitude slower compared to XGBoost) and lower accuracy than XGBoost. On the contrary, *XGBoost requires minimal storage, is fast to train and make predictions, and can learn incrementally over time.*

Hyperparameter tuning: In pre-analysis, we found that only two configuration parameters had a noticeable effect on the XGBoost performance: the maximum depth d of the trees and the number of rounds r for boosting [46]. Hence, we used grid search to optimize them offline using training data from our two workload traces (see Section 7) and found the same values for both workloads: $d = 20$ and $r = 10$. We used default values for other configuration parameters.

Algorithm 1 Downgrade process outline

```
1: procedure DOWNGRADE(StorageTier fromTier)
2:   if policy.startDowngrade(fromTier) then
3:     repeat
4:       file = policy.selectFileToDowngrade(fromTier)
5:       toTier = policy.selectDowngradeTier(file, fromTier)
6:       downgradeFile(file, fromTier, toTier)
7:     until policy.stopDowngrade(fromTier)
```

4.4 File Access Predictions

Section 4.2 described how a model M is trained incrementally over time. Before the system can start using M to make predictions, it needs to ensure that M has been trained enough. To achieve this, the system will occasionally use some training data points for evaluating the performance of M (before using them for training M). When the classification error rate drops below a certain threshold (e.g., 0.01), then the system can start using M for predicting whether a file will be accessed in the forward-looking class window of size w (e.g., in the next 30 minutes). At the current point in time t_c and for some file f, the system can get the prediction via: (1) setting the reference time $t_r = t_c$ (2) generating the feature vector based on the file size, creation time, and access times before t_r as explained in Section 4.1; and (3) using the model to predict the class label of f. Specifically, an XGBoost model will return a *probability score* indicating how likely is f to be accessed in the next w minutes.

The probability score is then used by the policies to decide which file(s) to upgrade or downgrade. We generate two separate models for this purpose — one for the upgrade policy and one for the downgrade policy — whose only difference lies in the class window size w. The upgrade policy wants to determine which files will be accessed in the immediate future and, hence, we want to set a small w (e.g., 30 minutes). On the other hand, the downgrade policy wants to determine which files have become cold, that is, will not be accessed for some time; hence, we want to set a large w (e.g., 6 hours). The two policies are further elaborated in Sections 5 and 6.

5. DOWNGRADE POLICIES

Algorithm 1 shows the outline of the downgrade process responsible for moving a file replica from a higher storage tier to a lower one. The procedure utilizes the four main methods implemented by a downgrade policy, which correspond to the four decision points discussed in Section 3.2. The downgrade procedure is invoked every time some data is added to a storage tier (e.g., after a file creation or replication) but the actual process starts based on the policy's decision (line #2). At that point, the policy is responsible for selecting a file to downgrade (line #4) and the target storage tier (line #5). The system will then schedule the downgrade request (line #6), which will take place asynchronously. Finally, the process will repeat until the policy decides to stop it (line #7).

5.1 When to Start the Downgrade Process

In a traditional cache, an object is discarded when the cache is full and a new object needs to enter the cache. This approach ensures that the cache is fully utilized and works well for fixed-size disk pages or small web objects. However, typical file sizes in analytics clusters are in the order of tens

Table 1: Downgrade policies

Acronym	Policy Name	Description
LRU	Least Recently Used	Downgrade the file that was accessed less recently than any other
LFU	Least Frequently Used	Downgrade the file that was used least often than any other
LRFU	Least Recently & Frequently Used	Downgrade the file with the lowest weight based on recency and frequency
LIFE	LIFE (PACMan [5])	Downgrade either the old LFU file or the largest file
LFU-F	LFU-F (PACMan [5])	Downgrade the LFU file among files older than a time window
EXD	Exponential Decay (Big SQL [16])	Downgrade the file with the lowest weight based on recency and frequency
XGB	XGBoost-based Modeling	Downgrade the file with the lowest access probability in the distant future

to hundreds of MBs [5, 9], so having a file write wait for other files to be downgraded would introduce significant latency delays. Hence, it is crucial for the system to be proactive and for the downgrade process from a tier T to start before T is full. All of our policies will start the downgrade process from a tier when its used capacity becomes greater than a threshold value (e.g., 90%), allowing for a more efficient overlapping between file writes and file downgrades.

5.2 Which file to downgrade

Once the downgrade process is activated for a particular storage tier T, the policy must select a file to remove from T in order to make room for new data. This particular decision is known as the *replacement or eviction policy* in the literature with a long history of related work [2, 36]. For comparison purposes, we have implemented three conventional eviction policies, three related policies from recent literature, and one new policy, listed in Table 1.

LRU (Least Recently Used) selects the file used least recently. LRU is widely used and is designed to take advantage of the temporal locality often exhibited in data accesses.

LFU (Least Frequently Used) selects the file with the least number of accesses. LFU is a typical web caching policy that keeps more popular files and evicts rarely used ones.

LRFU (Least Recently & Frequently Used) selects the file with the lowest weight, which is computed for each file based on both the recency and frequency of accesses. The weight W for a file f is initialized to 1 when f is created and updated each time f is accessed based on Formula 1:

$$W = 1 + \frac{H * W}{(timeNow - timeLastAccess) + H} \tag{1}$$

Parameter H represents the "half life" of W, i.e., after how much time the weight is halved. For e.g., if $H = 6$ hours and a file is accessed 6 hours after its last access, then its new weight will equal 1 plus half the old weight. Hence, files that are recently accessed multiple times will have a large weight, as opposed to files accessed a few times in the past.

LIFE aims at minimizing the average completion time of jobs in PACMan [5] by prioritizing small and recent files. Specifically, LIFE divides the files into two partitions: P_{old} containing the files that have not been accessed for at least some time window (e.g., 9 hours) and P_{new} with the rest. If P_{old} is not empty, then the LFU file is selected from it. Otherwise, LIFE selects the largest file from P_{new}.

LFU-F aims at maximizing cluster efficiency in PACMan [5] by evicting less frequently accessed files. LFU-F divides the files in the same two partitions as LIFE, namely, P_{old} and P_{new}. If P_{old} is not empty, then the LFU file is selected from it. Otherwise, LFU-F selects the LFU file from P_{new}.

EXD (Exponential Decay) explores the tradeoff between recency and frequency in data accesses in Big SQL [16]. In particular, it selects the file with the lowest weight W computed using the following formula:

$$W = 1 + W * e^{-\alpha \, * \, (timeNow \, - \, timeLastAccess)} \tag{2}$$

The parameter α determines the weight of frequency vs. recency and it is set to $1.16 * 10^{-8}$ based on [16].

XGB (XGBoost-based Modeling) incrementally trains and utilizes an XGBoost model (recall Section 4) for deciding which file will not be accessed in the distant future. Specifically, XGB will compute the access probability for the k least recently used files and select the file with the lowest access probability to downgrade. We compute probabilities for LRU files in order to avoid cache pollution with files that are never evicted, while we limit the computations to k files in order to bound the (low) overhead of building the features and using the model. In practice, we set k to be large (e.g., $k = 200$), and it has had limited impact on our workloads.

5.3 How to downgrade the selected file

When an object is selected for cache eviction, it is typically simply deleted. In our case, however, when a file is selected for downgrade from a higher tier, we typically want to move that file replica to a lower tier in order to retain the same number of replicas, and hence, maintain the file system's properties of high fault tolerance and availability. This decision entails selecting one lower storage tier for moving the file replica. For this purpose, we adapted the data placement policy used by OctopusFS, which makes decisions by solving a *multi-objective optimization problem.* In particular, the policy aims at finding a Pareto optimal solution that optimizes 4 objectives simultaneously: (1) fault tolerance for avoiding data loss due to failures; (2) load balancing for distributing I/O requests across storage tiers; (3) data balancing for distributing data blocks across storage tiers; and (4) throughput maximization for optimizing the overall I/O throughput of the cluster. For details, see [27].

5.4 When to stop the downgrade process

In conventional caches, eviction stops when there is enough room in the cache to fit the newly inserted object. Since we start the downgrade process proactively (i.e., before the cache is full), we need a different approach for stopping it. Specifically, all policies will stop the downgrade process from a storage tier T when its used capacity becomes lower than a threshold value (e.g., 85%), allowing for a small percent of T's capacity to be freed together.

6. UPGRADE POLICIES

Algorithm 2 outlines the upgrade process guided by the 4 decision points presented earlier in Section 3.2. The upgrade procedure is invoked (i) every time a file is accessed (but before it is actually read) and (ii) periodically in case the policy

Table 2: Upgrade policies

Acronym	Policy Name	Description
OSA	On Single Access	Upgrade a file into memory upon access (if not there already)
LRFU	Least Recently & Frequently Used	Upgrade a file if its weight is higher than a threshold
EXD	Exponential Decay (Big SQL [16])	Upgrade a file if its weight is higher than the weight of to-be-evicted files
XGB	XGBoost-based Modeling	Upgrade files with high access probability in the near future

Algorithm 2 Upgrade process outline

```
1: procedure UPGRADE(StorageTier fromTier, File accessedFile)
2:   if policy.startUpgrade(fromTier, accessedFile) then
3:     repeat
4:       file = policy.selectFileToUpgrade(fromTier)
5:       toTier = policy.selectUpgradeTier(file, fromTier)
6:       upgradeFile(file, fromTier, toTier)
7:     until policy.stopUpgrade(fromTier)
```

wants to make a proactive decision (an accessed file is not available in this case). Given a storage tier and the file that was just accessed (optional), the upgrade policy is responsible for deciding when the process starts (line #2), which file to upgrade (line #4), and the target storage tier (line #5). The system will then schedule the upgrade request (line #6), which will be piggybacked during the subsequent read or take place asynchronously. Finally, the process will repeat until the policy decides to stop it (line #7).

6.1 When to Start the Upgrade Process

Typically, all data accesses in a system that uses a cache must go through the cache first. If the accessed object O is located in the cache, it will be served from there; otherwise, O will be inserted into the cache. Unlike cache eviction policies, *cache admission* policies are not very common as they complicate the read process without major benefits in a traditional cache [16]. In our case, moving a file into a higher storage tier is costlier as it may be executed asynchronously and it may involve a large amount of data (10s to 100s of MBs). Hence, the decisions of when and what to upgrade are as important as when and what to downgrade. For comparison purposes, we have implemented two conventional admission policies, one related policy from recent literature, and one new policy, listed in Table 2.

OSA (On Single Access) implements the common approach of upgrading each file when it is accessed and not already present in the memory tier. With OSA, we do not allow upgrades from the HDD to the SSD tier to avoid (i) the overhead associated with moving large amounts of data between disks, and (ii) under-utilizing the HDDs available in the cluster.

LRFU (Least Recently & Frequently Used) starts the upgrade process for an accessed file f when the computed weight for f is greater than a threshold value. The weight takes into account both the recency and the frequency of accesses and is computed using Formula 1. The threshold value is empirically set to 3 in order to favor files that are accessed recently multiple times.

EXD (Exponential Decay) is used in Big SQL [16] for selecting which files to insert into the cache. If there is enough space in a higher storage tier to fit the accessed file f, then f will get upgraded. Otherwise, EXD will upgrade f only if its weight (computed using Formula 2) is higher than the sum of weights of the files that will need to be downgraded to make room for f.

XGB (XGBoost-based Modeling) incrementally trains and utilizes an XGBoost model (recall Section 4) for predicting if a file will get accessed in the near future. Specifically, XGB will compute the access probability for the k (e.g., $k = 200$) most recently used files and start the upgrade process if the access probability of a file is higher than the discrimination threshold. In binary classification, the discrimination threshold determines the boundary between the two classes, and it is empirically set to 0.5 (see Section 7.5).

6.2 Which file to upgrade

The decisions of when to start upgrading and which file to upgrade are tightly coupled in the upgrade process. Hence, all policies will select the file that triggered the process (described in Section 6.1 above) as the file to upgrade.

6.3 How to upgrade the selected file

We use the same multi-objective optimization problem formulation with the downgrade policies (recall Section 5.3) for selecting a higher storage tier for performing the upgrade, while considering the tradeoffs between fault tolerance, data and load balancing, and throughput maximization.

6.4 When to stop the upgrade process

With the exception of XGB, all other policies base their decision to start the upgrade process on the currently accessed file f. If they decide to start, f will be upgraded and the loop terminates. XGB, on the other hand, will continue the upgrade process until either there are no more files that are likely to be accessed in the near future, or until the total size of the scheduled upgrades exceeds a threshold (e.g., 1GB) to avoid upgrading a large amount of data at once.

7. EXPERIMENTAL EVALUATION

The evaluation is conducted on a 12-node cluster running CentOS Linux 7.2 with 1 Master and 11 Workers. The Master node has a 64-bit, 8-core, 3.2GHz CPU, 64GB RAM, and a 2.1TB RAID 5 storage configuration. Each Worker node has a 64-bit, 8-core, 2.4GHz CPU, 24GB RAM, one 120GB SATA SSD, and three 500GB SAS HDDs. The file systems are configured to use three storage tiers consisting of 4GB of memory, 64GB of SSD, and 400GB of HDD space each for storing file blocks on each Worker node. The default replication factor is 3 and the block size is 128MB.

7.1 Workload Properties

Our evaluation is based on two workloads derived from real-world production traces from Facebook and Carnegie Mellon University clusters. The *FB* trace was collected over a period of 6 months from a 600-node Hadoop cluster at Facebook and contains arrival times, durations, file sizes, and other data about executed MapReduce jobs [10]. The *CMU* trace contains similar data from scientific MapReduce workloads executed over a period of 31 months at OpenCloud, a 64-node Hadoop cluster [12]. From the traces, we

Table 3: Job size distributions. The jobs are binned by their data sizes in our FB and CMU workloads

Bin	Data size	% of Jobs		% of Resources		% of I/O		Task Time (mins)	
		FB	CMU	FB	CMU	FB	CMU	FB	CMU
A	0-128MB	74.4%	63.4%	25.0%	32.3%	3.2%	10.9%	76.7	119.5
B	128-512MB	16.2%	29.1%	12.2%	27.9%	16.1%	30.5%	37.6	103.2
C	0.5-1GB	4.0%	0.9%	7.3%	1.3%	12.0%	2.4%	22.3	5.0
D	1-2GB	3.0%	4.9%	13.4%	21.0%	19.3%	23.3%	41.0	77.6
E	2-5GB	1.6%	1.5%	20.8%	15.1%	21.9%	27.8%	63.9	55.7
F	5-10GB	0.8%	0.3%	21.4%	2.5%	27.5%	5.2%	65.6	9.2

used SWIM [42], a statistical workload injector for MapReduce, to generate two realistic and representative workloads that preserve the original workload characteristics such as the distribution of input sizes and the skewed popularity of data [5]. Using SWIM, we replay each workload with the same inter-arrival times and input/output files as in the original workload, allowing us to mimic the access patterns of the files. In order to reflect the smaller size of our cluster and to simulate the load experienced by the original clusters, we scale down the file sizes proportionately [5].

The derived FB and CMU workloads consist of 1000 and 800 jobs, respectively, scheduled for execution over a 6-hour period. To separate the effect of storage tiering on different jobs, we split them based on their input data size into 6 bins. Table 3 shows the distribution of jobs by count, cluster resources they consume, amount of I/O they generate, and aggregate task execution time. The jobs in both workloads exhibit a heavy-tailed distribution of input sizes, also noted in previous studies [5, 9]. In particular, the FB workload is dominated by small jobs; 74.4% of them process <128MB of data. However, these jobs only account for 25% of the cluster resources consumed and perform only 3.2% of the overall I/O. On the contrary, FB jobs processing >1GB of data, account for 54% of resources and 68% of I/O. The distribution of input sizes is less skewed for CMU with 63.4% of the jobs processing <128MB of data. Similarly, CMU jobs processing >1GB of data, account for 38% of resources and 56% of I/O. In both workloads, even though larger jobs (Bins D-F) constitute only a small fraction of the workload, they account for about half the total task execution time.

In terms of files, the FB and CMU workloads process 1380 and 1305 files with a total size of 92GB and 85GB, respectively. The popularity of files is also skewed in data-intensive workloads, with a small fraction of the files accessed very frequently, while the rest are accessed less frequently [5, 9]. Specifically, 5.7% of FB files and 2.8% of CMU files are accessed more than 5 times. Such repeatability must be exploited to improve job performance by ensuring their inputs have replicas residing in the highest storage tier (i.e., memory). In addition, a sizable fraction of the files (23% for FB and 18% for CMU) are created but not accessed afterwards. Hence, it is important for a downgrade policy to identify such cases and remove their replicas from memory early on. CDFs for the workload statistics are presented in [23].

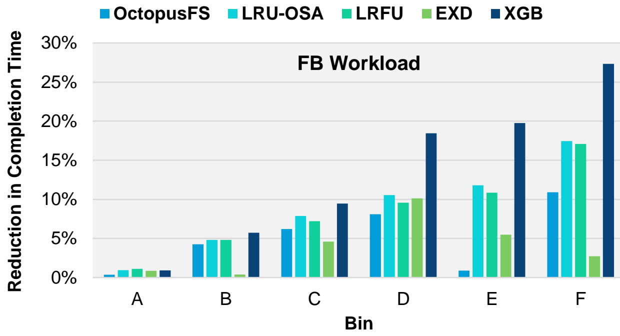
7.2 End-to-End Evaluation

For this evaluation, we executed the two workloads over HDFS v2.7.7, the default OctopusFS (i.e., without any downgrade or upgrade policies), Octopus++ using the LRU downgrade policy and the OSA upgrade policy (as a baseline for Octopus++), and Octopus++ using all common downgrade and upgrade policies (i.e., LRFU, EXD, and XGB; recall Tables 1 and 2). We compare them using two complementary performance metrics: (i) the *average completion time* of jobs, and (ii) the *cluster efficiency* (defined as finishing the jobs by using the least amount of resources [5]).

Figure 4 shows the reduction percentage in job completion time compared to the HDFS setting for both workloads for each bin. Small jobs (Bins A, B) experience only a small improvement in completion time for all policies, with less than 5% and 10% gain in FB and CMU, respectively. This result is not surprising since the time spent in I/O is only a small fraction compared to CPU processing and scheduling overheads. As the job input sizes increase, so do the gains in job completion time, while we start observing *different behavior across the policies and between the two workloads.* Specifically, the LRU-OSA and LRFU policies are performing quite well for the FB workload — as it exhibits good temporal locality of reference — resulting in up to 17% reduction in completion time for large jobs (Bin F). The CMU workload, on the other hand, has different access patterns that make the LRU-OSA and LRFU policies perform poorly, limiting gains down to 4-10%. In fact, LRU-OSA performs even worse than OctopusFS in some cases, which lacks any policy for automatically moving data across the storage tiers after the initial placement.

EXD performs well only for jobs in Bin D with 10% gains in completion time, while it performs poorly for larger jobs in FB. Recall that EXD explores the tradeoff between recency and frequency without taking file size into account. In several cases, EXD downgrades large files before they are accessed, causing their data to be (re-)read from SSDs or HDDs. Interestingly, EXD performs well for CMU, resulting in 13–16% gains for Bins D and E. Finally, our XGB policy is able to provide the *highest reduction in average completion time across all job bins and for both workloads.* For FB, there is a clear increasing gain as the job size gets larger with 18–27% benefits, almost *double* compared to the second-best policy. For CMU, the gains are higher for medium jobs (Bins D, E) with benefits over 21%, while the benefits for large jobs (Bin F) are still high at 15%. Overall, XGB is able to effectively learn the different access patterns and detect data reuse across jobs for both workloads.

Every time data is accessed from memory or even SSDs, the efficiency of the cluster improves. Figure 5 shows how this improvement in efficiency is derived from the different job bins. Larger jobs have a higher contribution in efficiency improvement compared to small jobs since they are responsible for performing a larger amount of I/O (recall Table 3). Across the different policies, the trends for the efficiency improvement are similar to the trends for the completion time reduction discussed above: LRU-OSA and LRFU generally offer good benefits for FB; EXD offers good benefits for CMU; and *XGB offers the best gains in both workloads.* Hence, improvements in cluster efficiency are often accompanied by lower job completion times, doubling the benefits. In



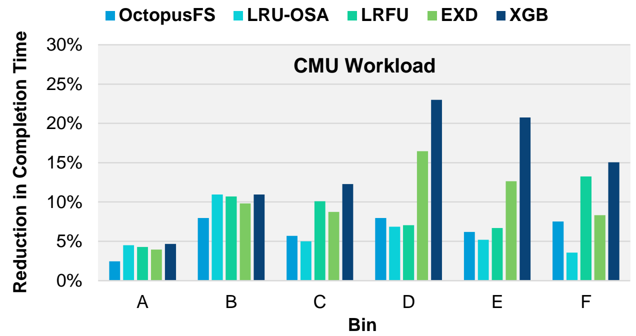
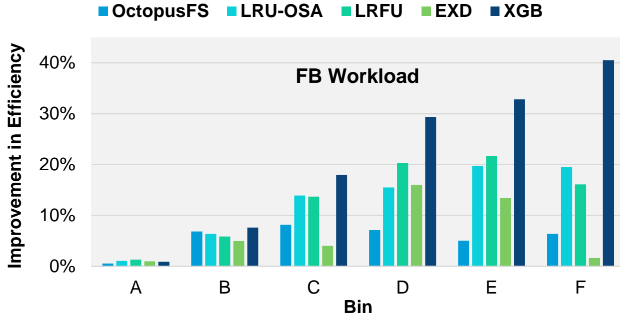


Figure 4: Percent reduction in completion time over HDFS for the (a) FB and (b) CMU workloads



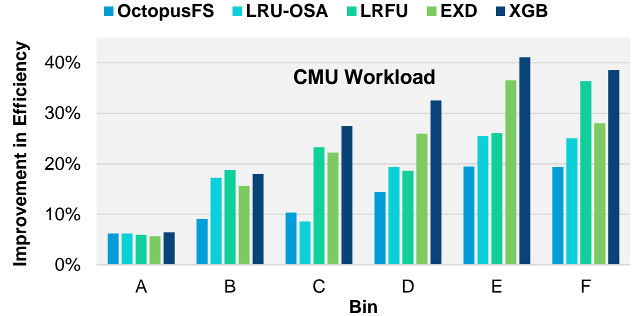


Figure 5: Percent improvement in cluster efficiency over HDFS for the (a) FB and (b) CMU workloads

FB, for example, XGB is able to reduce the completion time of large jobs by 27% while consuming 41% less resources.

One interesting observation is that the magnitude of the gain in efficiency is higher than the magnitude of the reduction in completion times. This is explained in two ways. First, the jobs are executed as a set of parallel tasks. Even if a large fraction of the tasks consume less resources via avoiding disk I/O, the remaining tasks may delay the completion of a job (even though the benefits do propagate to all tasks due to lower I/O congestion). Second, the job completion time also accounts for CPU processing as well as the output data generation, both of which are independent of the input I/O. Nonetheless, improving cluster efficiency leaves significantly more room for executing more jobs and for better overlapping the background I/O generated by our policies.

Additional drill down results (not shown due to space constraints) reveal that XGB: (1) results in the highest percentage of memory accesses; (2) is the most selective in terms of upgrade I/O; and (3) offers the highest byte hit ratio.

Scalability results: We repeated our experiments on Amazon EC2 using the m4.2xlarge instance type (8 cores, 32GiB RAM) with SSD and HDD attached EBS volumes to resemble our local cluster setup. We scaled up the EC2 cluster from 11 to 88 worker nodes (plus 1 master node) while proportionally increasing the workload data sizes and we were able to fully replicate our results. The key insights when using our XGB policies are: (1) the improvement in cluster efficiency increases with the cluster size, especially for small-medium jobs, revealing the increasing benefits of avoiding disk I/O and better utilizing the cluster resources; and (2) the gains in completion time are similar for small-medium jobs but decrease for large jobs (from 24% to 15% gains) as the cluster size increases because of the increasing cost of the output data generation with a replication factor of 3. These results are detailed in our extended tech report [23].

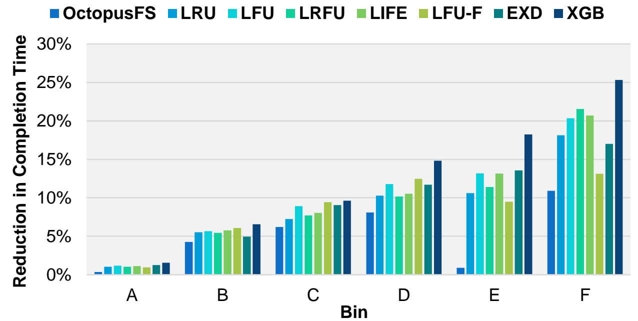


Figure 6: Percent reduction in completion time for downgrade policies over HDFS for the FB workload

7.3 Comparison of Downgrade Policies

For this experiment, we executed the two workloads over Octopus++ using all downgrade policies listed in Table 1, while disabling upgrades. Our goals are (i) to isolate the effects of downgrading from upgrading, and (ii) to gain additional insights from the downgrade policies. Figure 6 shows the percent reduction in average completion time for all jobs in FB broken down into bins. The trends for the policies we already studied in Section 7.2 are similar in general. However, a careful comparison between Figures 4(a) and 6 yields some interesting observations. First, the LRU and LRFU downgrade policies cause the same amount of gains in completion time as when upgrade policies are present. The EXD upgrade policy has a strong negative effect on the completion time benefits compared to only enabling the downgrade policy (5–14% lower gains for large jobs). The extra amount of data upgraded is causing the downgrade of other useful files, negatively affecting performance. The pairing of the XGB downgrade and upgrade policies has a small positive affect of 1.5-3.6% additional reduction in completion times

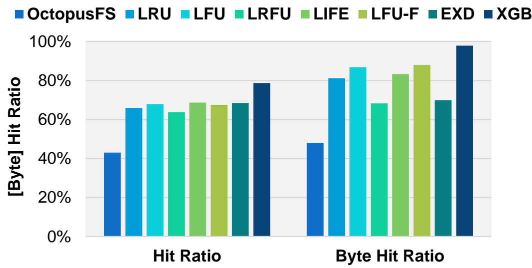


Figure 7: Hit Ratio and Byte Hit Ratio for downgrade policies for FB based on memory accesses

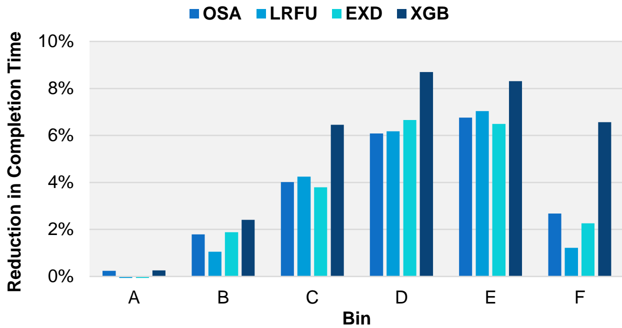


Figure 8: Percent reduction in completion time for upgrade policies over HDFS for the FB workload

compared to using the XGB downgrade policy alone. This highlights the effectiveness of both XGB policies in making the right decisions when moving files across the storage tiers.

The LIFE policy [5], which is designed to improve job completion time, performs fairly well in this metric, especially for larger jobs (13–21% gains for Bins E, F). However, it falls short compared to the 18–25% benefits provided by XGB for Bins E and F. LRF-U [5], which targets at improving cluster efficiency, ranks second in this metric for small to medium jobs (Bins B-D) but performs poorly for larger jobs. Recall that LRF-U does not take file size into account. *XGB, on the other hand, provides the highest gains in cluster efficiency for all bins.* The CMU results do not provide any additional insights and are not presented due to space constraints.

To further analyze the performance of the downgrade policies we computed two additional metrics: (i) *Hit Ratio (HR)*, i.e., the percentage of requests that can be satisfied by the memory tier; and (ii) *Byte Hit Ratio (BHR)*, i.e., the percentage of bytes satisfied by the memory tier [2]. We focus on the memory tier as it provides the highest benefits. Figure 7 shows the HR and BHR of all downgrade policies based on memory accesses. OctopusFS achieves less than 50% for both ratios as less than half of the files have replicas in memory. With the exception of XGB, all other policies achieve a similar HR of ~67%. BHR reveals a different picture with LRFU and EXD offering ~69% BHR (which explains their lower gains in cluster efficiency), while the rest offer ~85%. *XGB is able to achieve a 98% BHR, highlighting the policy's ability to maintain the most relevant files in memory.*

7.4 Comparison of Upgrade Policies

In this section, we evaluate all upgrade policies listed in Table 2 in isolation. We instructed the data placement policy of Octopus++ to initially place all file replicas on the HDD tier and let the upgrade policies decide when to move replicas to higher tiers. Figure 8 shows the gains in job completion time for each upgrade policy for each job bin. Overall, the gains are limited to less than 9% since gains are only possible when an input file is accessed repeatedly, provided that a policy was able to upgrade the file early on. OSA, the simple policy that upgrades a file into memory upon access, performs relatively well for most bins (with 2–7% gains), while the other policies offer similar benefits in most cases. The only policy that stands apart is the XGB one, which offers the highest benefits, showcasing the predictive powers of our ML model.

Table 4: Statistics for upgrade policies for FB

Policies	GB Read from Memory Tier	GB Upgraded to Memory Tier	Byte Accuracy	Byte Coverage
OSA	9.41	34.52	0.27	0.21
LRFU	9.03	22.82	0.40	0.21
EXD	6.45	22.59	0.29	0.15
XGB	13.77	27.66	0.50	0.31

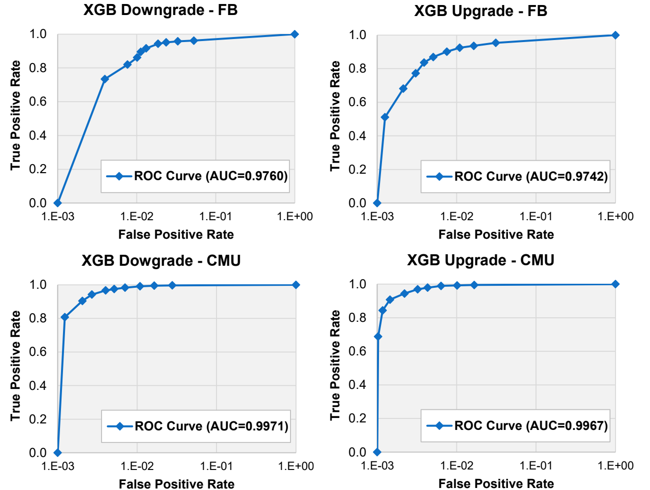


Figure 9: ROC curves for XGB downgrade/upgrade

Table 4 lists two insightful statistics — amount of data upgraded to and read from memory — as well as two important web prefetching metrics: (1) *Byte Accuracy (BAc)*, defined as the ratio of data read from memory to the total amount of data upgraded; and (2) *Byte Coverage (BCo)*, defined as the ratio of data read from memory to the total amount of data read [2]. As the least selective policy, OSA upgraded the largest amount of data (34.5GB) leading to the 2nd highest amount of data read from memory (9.4GB) but with a fairly low BAc. LRFU and EXD upgraded about the same amount of data (~22.7GB) but only LRFU was able to score a relatively high BAc of 40%. Even though XGB upgraded slightly more data (27.6GB), it was able to achieve a higher BAc at 50% and the highest BCo at 31%, explaining the high performance benefits attained by XGB.

7.5 XGBoost Model Evaluation

We evaluate the performance of our XGBoost models using a receiver operating characteristic (ROC) curve and the area under the curve (AUC) [39]. The ROC curve takes as input the file access probabilities predicted by the model and the true class labels. It then plots the true positive rate (i.e., the probability of detection) against the false positive

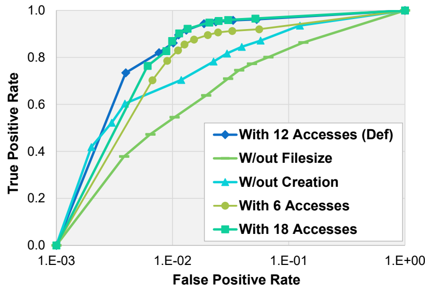


Figure 10: ROC curves for FB downgrade XGBoost models with selected features

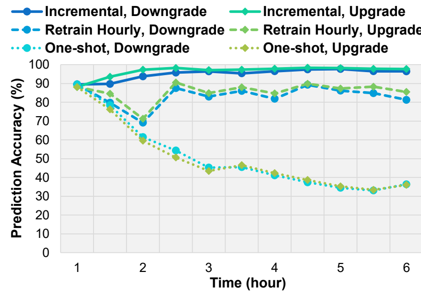


Figure 11: Prediction accuracies of the incremental learning, retrain hourly, and one-shot approaches for FB

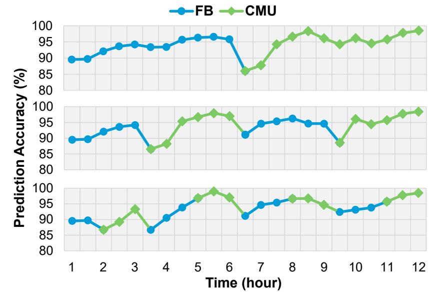


Figure 12: Prediction accuracies of downgrade XGBoost models when alternating FB and CMU workloads

rate (i.e., the probability of false alarm) at various threshold settings. To train our models and perform a proper out-of-sample analysis, we split our 6-hour data set into training (first 4 hours), validation (5th hour), and test (6th hour) sets. All results presented in this Section are based on test data. The FB and CMU data sets consist of 17968 and 13920 data points, respectively, evenly distributed (approximately) in each hour. Figure 9 shows the 4 ROC curves for the downgrade and upgrade models trained and evaluated over the FB and CMU workloads (note the logarithmic scale of the x-axis). In all cases, the curves are near point $(0, 1)$ with AUC values higher than 0.97 (1 is the max), which indicate the very high prediction performance of our models. The prediction accuracy at the chosen discrimination threshold (0.5) is between 97-99% in all four cases.

The features used by our XGBoost model formulation are the file size, the creation time, and the last k (default=12) access times. To assess the impact of the selected features, we evaluated the performance of the XGBoost model while varying the features used. In particular, Figure 10 shows the ROC curves for the FB downgrade case (other cases are similar) when the features (i) do not include the file size; (ii) do not include the creation time; (iii) use only the last 6 access times; (iv) use the default 12 access times; and (v) use 18 access times. Note that (iii)-(v) do use the file size and creation time. The results reveal that both the file size and creation time are individually important predictors of file accesses and improve the overall performance of the model. Using only 6 time accesses still leads to good model performance, albeit slightly lower than our default case, while using 18 time accesses has a marginal impact. Overall, these results verify the selection of the aforementioned features to be used for predicting file access patterns.

Next, we evaluate our *incremental learning* approach that cumulatively trains a model over time against (i) retraining the model every one hour and (ii) a *one-shot* learner that trains on data once from the first hour. Figure 11 shows how the prediction accuracy (i.e., the ratio of true predictions over all of them) for the three approaches varies over time for the downgrade and upgrade policies over FB. Although a one-shot learner may start with a high initial accuracy near 90%, over time it leads to a significant degradation of accuracy below 40% as the workload evolves. The retraining approach exhibits an oscillating patter of accuracy that increases right after each training but stays within 80% and 90%. On the contrary, the incremental learner becomes better over time, efficiently adapting to new workloads, and continues to produce accurate predictions (~98%) over the entire duration of our experiments.

Finally, we investigate the impact of sudden access pattern changes to the XGBoost incremental model performance. Figure 12 shows the prediction accuracy of the XGB downgrade policy over time as we switch between the FB and CMU workloads. In the first experimental variation, we execute FB for 6 hours and then switch to CMU, which exhibits different access patterns. At that point, accuracy drops 9.8% down to 86% but then quickly increases to over 95% as the model starts learning the new access patterns. In the next 2 variations, we alternate the FB and CMU execution every 3 and 1.5 hours, respectively, and observe that (i) as time goes by, the drops in accuracy decrease in magnitude, (ii) the more we interleave the workloads, the lower the drop as the model learns both workloads, and (iii) the model is always able to learn and increase accuracy quickly.

7.6 System Overheads

Adding one training sample in an XGBoost model takes on average 0.16ms, while making a prediction takes 1.8ns. Overall, during a 6-hour-long experiment, model training accounted for 5.3 CPU seconds in total, while selecting a file to downgrade or upgrade amounted to 0.49 CPU seconds; showcasing the negligible CPU overhead caused by XGBoost. In terms of memory, an XGBoost model consumes ~200KB. In addition, we maintain a max of 956 bytes per file (at most 12 access times plus some auxiliary data) for generating the feature vectors, which in our experiments added just a few MB of memory. Even in large clusters with millions of files, the extra memory overhead is <1GB, which is a fraction of the total memory needed by an HDFS NameNode at that scale and justifiable given the attainable performance benefits.

8. CONCLUSIONS

This paper describes a framework for automatically managing data across storage tiers in distributed file systems (DFSs) using a set of pluggable policies. The generality of the framework is evident by the 11 downgrade and upgrade policies implemented based on both old and new techniques. Our proposed policies employ light-weight gradient boosted trees for learning how files are accessed by a workload and use that information to make decisions on which files to move up or down the tiers. The models are incrementally updated based on how the file access patterns change and so are able to maintain high prediction accuracy over time. The framework and all policies have been implemented in a real distributed file system and successfully evaluated over two realistic workloads.

9. REFERENCES

[1] A. Agarwal. *Enable Support for Heterogeneous Storages in HDFS*, 2016. https://issues.apache.org/jira/browse/HDFS-2832.

[2] W. Ali, S. M. Shamsuddin, and A. S. Ismail. A Survey of Web Caching and Prefetching. *Intl. Journal of Advances in Soft Computing & Its Applications*, 3(1):18–44, 2011.

[3] W. Ali, S. Sulaiman, and N. Ahmad. Performance Improvement of Least-Recently-Used Policy in Web Proxy Cache Replacement using Supervised Machine Learning. *Intl. Journal of Advances in Soft Computing & Its Applications*, 6(1), 2014.

[4] *Alluxio: In Memory Distributed Storage*, 2019. http://www.alluxio.org/.

[5] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 267–280. USENIX, 2012.

[6] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. Long. ACME: Adaptive Caching Using Multiple Experts. In *Proc. of the Workshop on Distributed Data and Structure (WDAS)*, pages 143–158, 2002.

[7] Z. Chang, L. Lei, Z. Zhou, S. Mao, and T. Ristaniemi. Learn to Cache: Machine Learning for Network Edge Caching in the Big Data Era. *IEEE Wireless Communications*, 25(3):28–35, 2018.

[8] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proc. of the 22nd ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 785–794. ACM, 2016.

[9] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *PVLDB*, 5(12):1802–1813, 2012.

[10] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance using Workload Suites. In *Proc. of the 2011 IEEE Intl. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 390–399. IEEE, 2011.

[11] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. CAST: Tiering Storage for Data Analytics in the Cloud. In *Proc. of the 24th Intl. Symp. on High Performance Distributed Computing (HPDC)*, pages 45–56. ACM, 2015.

[12] *CMU OpenCloud Hadoop Cluster Trace*, 2016. http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html.

[13] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proc. of the 2011 ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 25–36. ACM, 2011.

[14] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking through Siberia: Managing Cold Data in a Memory-optimized Database. *PVLDB*, 7(11):931–942, 2014.

[15] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive Paging Algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[16] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J. Schmitz-Hermes. Adaptive Caching in Big SQL using the HDFS Cache. In *Proc. of the 7th ACM Symp. on Cloud Computing (SoCC)*, pages 321–333. ACM, 2016.

[17] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *PVLDB*, 5(11):1424–1435, 2012.

[18] A. Gepperth and B. Hammer. Incremental Learning Algorithms and Applications. In *Proc. of the 24th European Symp. on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*. H.A.L. Publishing, 2016.

[19] B. Gorman. *A Kaggle Master Explains Gradient Boosting*, 2017. http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/.

[20] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari. Adaptive Caching by Refetching. In *Proc. of the 15th Intl. Conf. on Neural Information Processing Systems (NIPS)*, pages 1489–1496. MIT Press, 2002.

[21] *GridGain In-Memory Computing Platform*, 2019. http://www.gridgain.com/.

[22] *HDFS Centralized Cache*, 2016. https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[23] H. Herodotou and E. Kakoulli. Automating Distributed Tiered Storage Management in Cluster Computing. *arXiv preprint arXiv:1907.02394*, 2019. https://arxiv.org/abs/1907.02394.

[24] Y. Im, P. Prahladan, T. H. Kim, Y. G. Hong, and S. Ha. SNN-Cache: A Practical Machine Learning-based Caching System Utilizing the Inter-Relationships of Requests. In *Proc. of the 52nd Conf. on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2018.

[25] H. Jeon, K. El Maghraoui, and G. B. Kandiraju. Investigating Hybrid SSD FTL Schemes for Hadoop Workloads. In *Proc. of the 2013 ACM Intl. Conf. on Computing Frontiers (CF)*, pages 20:1–20:10. ACM, 2013.

[26] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, et al. Exploring Data Staging Across Deep Memory Hierarchies for Coupled Data Intensive Simulation Workflows. In *Proc. of the 2015 IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1033–1042. IEEE, 2015.

[27] E. Kakoulli and H. Herodotou. OctopusFS: A Distributed File System with Tiered Storage Management. In *Proc. of the 2017 ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 65–78. ACM, 2017.

[28] A. Kougkas, H. Devarajan, and X.-H. Sun. Hermes: A Heterogeneous-aware Multi-tiered Distributed I/O Buffering System. In *Proc. of the 27th Intl. Symp. on High Performance Distributed Computing (HPDC)*, pages 219–230. ACM, 2018.

[29] K. Krish, A. Anwar, and A. R. Butt. hatS: A Heterogeneity-aware Tiered Storage for Hadoop. In *Proc. of the 14th IEEE/ACM Intl. Symp. on Cluster,*

Cloud and Grid Computing (CCGrid), pages 502–511. IEEE, 2014.

[30] J. J. Levandoski, P.-Å. Larson, and R. Stoica. Identifying Hot and Cold Data in Main-memory Databases. In *Proc. of the 29th IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 26–37. IEEE, 2013.

[31] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A Platform for Scalable One-pass Analytics Using MapReduce. In *Proc. of the 2011 ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 985–996. ACM, 2011.

[32] S. M. Lundberg and S.-I. Lee. A Unified Approach to Interpreting Model Predictions. In *Proc. of the 31st Intl. Conf. on Neural Information Processing Systems (NIPS)*, pages 4768–4777. Curran Associates Inc., 2017.

[33] T. Lykouris and S. Vassilvtiskii. Competitive Caching with Machine Learned Advice. In *Proc. of the 35th Intl. Conf. on Machine Learning (ICML)*, pages 3296–3305. PMLR, 2018.

[34] M. Mesnier, E. Thereska, G. R. Ganger, and D. Ellard. File Classification in Self-* Storage Systems. In *Proc. of the 2004 IEEE Intl. Conf. on Autonomic Computing (ICAC)*, pages 44–51. IEEE, 2004.

[35] M. Mihailescu, G. Soundararajan, and C. Amza. MixApart: Decoupled Analytics for Shared Storage Systems. In *Proc. of the 11th USENIX Conf. on File and Storage Technologies (FAST)*, pages 133–146. USENIX, 2013.

[36] S. Podlipnig and L. Böszörmenyi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[37] M. T. Ribeiro, S. Singh, and C. Guestrin. Why Should I Trust You?: Explaining the Predictions of any Classifier. In *Proc. of the 22nd ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1135–1144. ACM, 2016.

[38] R. Salkhordeh, S. Ebrahimi, and H. Asadi. ReCA: An Efficient Reconfigurable Cache Architecture for Storage Systems with Online Workload Characterization. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 29(7):1605 – 1620, 2018.

[39] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[40] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of the 26th Intl. Conf. on Massive Storage Systems and Technology (MSST)*, pages 1–10. IEEE, 2010.

[41] S. Sulaiman, S. M. Shamsuddin, A. Abraham, and S. Sulaiman. Intelligent Web Caching using Machine Learning Methods. *Neural Network World*, 21(5):429, 2011.

[42] *SWIM: Statistical Workload Injector for MapReduce*, 2016. https://github.com/SWIMProjectUCB/SWIM/wiki.

[43] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the 4th ACM Symp. on Cloud Computing (SoCC)*, page 5. ACM, 2013.

[44] P. Venketesh and R. Venkatesan. A Survey on Applications of Neural Networks and Evolutionary Techniques in Web Caching. *IETE Technical Review*, 26(3):171–180, 2009.

[45] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan. Driving Cache Replacement with ML-based LeCaR. In *Proc. of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, 2018.

[46] *XGBoost Documentation*, 2019. https://xgboost.readthedocs.io/.

[47] B. Xie, Y. Huang, J. S. Chase, et al. Predicting Output Performance of a Petascale Supercomputer. In *Proc. of the 26th Intl. Symp. on High Performance Distributed Computing (HPDC)*, pages 181–192. ACM, 2017.

[48] M. Zaharia, M. Chowdhury, T. Das, et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 15–28. USENIX, 2012.