AutoCache: Employing Machine Learning to Automate Caching in Distributed File Systems

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

***Abstract*—The use of computational platforms such as Hadoop and Spark is growing rapidly as a successful paradigm for processing large-scale data residing in distributed file systems like HDFS. Increasing memory sizes have recently led to the introduction of caching and in-memory file systems. However, these systems lack any automated caching mechanisms for storing data in memory. This paper presents AutoCache, a caching framework that automates the decisions for when and which files to store in, or remove from, the cache for increasing system performance. The decisions are based on machine learning models that track and predict file access patterns from evolving data processing workloads. Our evaluation using real-world workload traces from a Facebook production cluster compares our approach with several other policies and showcases significant benefits in terms of both workload performance and cluster efficiency.**

***Index Terms*—automated caching, distributed file systems**

I. Introduction

Social graph analysis, business analytics, and scientific data processing are among data-intensive jobs that are frequently executed on big data platforms such as Hadoop YARN [1] and Spark [2]. Such jobs tend to spend significant fractions of their overall execution in reading and writing data residing in distributed file systems (DFSs) such as HDFS [3]. In order to increase I/O performance, larger memory sizes are now commonly utilized in cluster computing. Specifically, HDFS has recently added support for caching input files internally [4], while in-memory DFSs like Alluxio [5] and GridGain [6] can be used for storing or caching HDFS data in memory. Finally, the OctopusFS distributed file system [7] supports storing file replicas on the storage media that are locally attached on the cluster nodes, including memory and SSDs.

Most aforementioned systems expose cache-related APIs to users or higher-level systems. For instance, an application using HDFS can issue requests to cache files. However, when the cache gets full, no additional caching requests will be served until the application *manually* uncaches some files [4]. Similarly, OctopusFS offers a placement policy for determining whether or not to store a file replica in memory at creation time, but lacks any features for automatically moving it afterwards. Alluxio and GridGain, on the other hand, implement basic policies for removing data from memory when full, such as LRU (Least Recently Used) [8]. 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 [8], [9]. In addition, these systems do not offer 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. This lack of automated caching places a significant burden on both system admins and application developers when they attempt to optimize system performance [10], [11].

Analysis of production workloads from Facebook, Cloudera, and MS Bing [9], [12] have revealed several types of data re-access patterns (e.g., hourly, daily) for both small and large analytics jobs. Hence, distinguishing reused data and keeping them in memory can yield significant performance benefits [9]. Workloads also tend to change over time producing different data access patterns [11], which need to be taken into account while managing a cache. Overall, it is crucial for the underlying DFSs to include automated caching capabilities for improving cluster efficiency and workload performance.

In this paper, we introduce *AutoCache, a framework for automated cache management in DFSs*. Specifically, AutoCache involves caching policies for deciding (i) when and which files to *admit* into the cache for improved read performance, and (ii) when and which files to *evict* from the cache to release memory pressure. Within AutoCache, we implement several conventional cache eviction and admission policies [8], policies from recent literature [10], and our own policies.

Our proposed policies employ *machine learning (ML) for tracking and predicting file access patterns*. In particular, we use light-weight *gradient boosted trees* [13] to learn how files are accessed by the workload and use the generated models for making admission and eviction decisions. The efficiency by which the models are learned, allows AutoCache to relearn them frequently (e.g., every few hours) and, hence, to naturally adapt to workload changes over time. Even though there is a vast amount of related work in caching, we offer a *unique ML-based, automated, and adaptive approach to cache management in DFSs*. Finally, we have implemented our approach in one of the post popular DFSs, namely HDFS.

In summary, the key contributions of this paper are:

1) The design and implementation of the AutoCache framework for automatically managing a file cache in DFSs.
2) ML-based policies for predicting file access patterns and dynamically moving files to and from the cache.
3) An experimental evaluation using real-world workload traces from a Facebook production cluster, showcasing significant performance benefits.

The paper is organized as follows. Section II presents the proposed cache management framework, while Section III

formulates the ML models for predicting file access patterns. Sections IV and V outline several policies for cache eviction and admission, respectively. The experimental evaluation is presented in Section VI and Section VIII concludes the paper.

II. Cache Management Framework

Distributed file systems like HDFS [3] are the most popular form of storage systems used with current Big Data platforms such as Hadoop [1] and Spark [2]. They store *files* that are typically broken down into large *blocks* (e.g., 128MB in size). The blocks are then replicated and stored on locally-attached hard disk drives (HDDs) on the cluster nodes, as shown in Figure 1. HDFS has recently added support for caching files into the nodes' local memory for improving read I/O performance and reducing read latencies [4]. However, caching requests can only be made through an API and there is no support for automatically caching or uncaching files.

HDFS uses a multi-master/worker architecture (see Figure 1) that consists of *NameNodes*, *DataNodes*, and *Clients*.

NameNodes: Each NameNode contains (i) the *FS Directory*, offering a traditional hierarchical file organization and operations; (ii) a *Block Manager*, maintaining the mapping from file blocks to nodes; and (iii) a *Node Manager*, containing the network topology and maintaining node statistics.

DataNodes: The DataNodes are responsible for (i) storing and managing file blocks on the locally-attached HDDs and memory; (ii) serving read and write requests from Clients; and (iii) performing block creation, deletion, and replication upon instructions from the NameNodes.

Clients: The Clients expose APIs for all typical file system operations such as creating directories or reading/writing files.

We have extended HDFS by adding two core components in the NameNodes. The *AutoCache Manager* is responsible for orchestrating the caching and uncaching of files based on the decisions of *pluggable admission and eviction policies*. Each policy makes two key decisions: (i) when to start and stop the admission (or eviction) process; and (ii) which file(s) to admit (or evict). These decisions are guided by file and node statistics maintained by the system as well as notifications received after a file creation, access, modification, or deletion.

The *AutoCache Monitor* is responsible for handling the caching and uncaching requests from the Cache Manager, as well as monitoring the overall state of the distributed cache. We also modified the DataNodes to enable caching while piggybacking on file writes/reads as well as to directly store file blocks in memory (rather than using memory-mapped files) to increase efficiency. We did not modify the Client to keep it backward compatible with HDFS.

Even though we implemented our approach in HDFS, it is not specific to the internal workings of HDFS and we believe it can be easily implemented in other in-memory distributed file systems (e.g., Alluxio [5], GridGain [6]). Finally, we have focused on caching at the file level (vs. block level) since previous research (e.g., [9], [10]) has shown that workload performance is improved only when the entire file is present in memory (coined the "all-or-nothing" property in [9]).

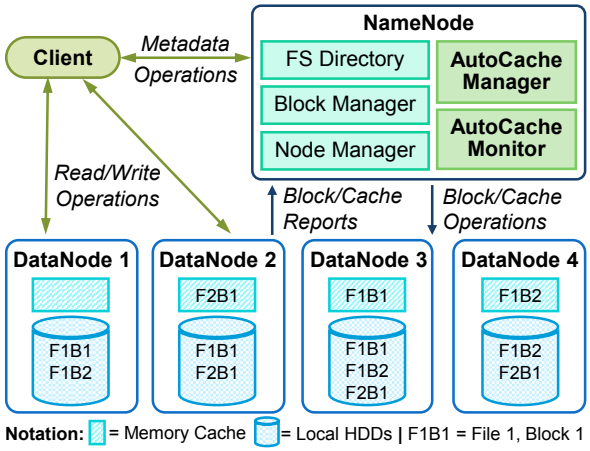

Fig. 1: HDFS Architecture with AutoCache

III. File Access Pattern Modeling

File accesses in clusters are typically driven by analytical workloads, causing various types of data re-access patterns [9], [12]. For instance, some files may be read by multiple jobs and reused for a few hours before becoming cold, while others are reused for longer periods of time. Such access patterns tend to change over time as workloads naturally evolve based on analytical needs [11]. The above observations have motivated our approach of modeling file access patterns using *feature-based binary classification* to predict whether a file will be accessed in the near future (and hence should be cached) or it has become cold (and hence should be uncached).

A. Training Data Preparation

The 3 most important file properties impacting cache management decisions are *recency* (i.e., time of the last access), *frequency* (i.e., number of accesses), and *size* [8]. All typical file systems already maintain each file's size, last access time, and creation time. Keeping track of frequency is simple but it does not capture any potential re-access patterns. Hence, we keep track of the time accesses during a particular time window (e.g., the last 24 hours). Overall, the file size, creation time, and access times for files constitute our input data.

The first step in data preparation for binary classification is converting the input data into *feature vectors* $\vec{x}_i$ and *class labels* y_i. Given a *reference* point in time, say 12:00, the time accesses before can be used for feature generation and the time accesses after can be used for class labeling. However, timestamps are not good feature candidates for machine learning because their value constantly increases over time. Hence, we propose two approaches for feature generation. The first approach discretizes time into *fixed time intervals* and counts the number of file accesses in each interval. The counts are then used as features and intuitively serve as proxies for any underlying file access patterns. Figure 2 shows an example where time is divided into 1-hour intervals. Between 11:00-12:00, the file is accessed two times, whereas between 10:00-11:00, it is not accessed at all. The second approach discretizes time into *growing time intervals* that double each time as you

move into the past. In the example, the interval boundaries are 15, 30, 60, etc. minutes before 12:00. For both approaches, the file size as well as the time difference between the reference and creation time form the final two features.

The class label y indicates whether the file will be accessed in a given forward-looking *class window*: if a file is accessed during the window, then $y = 1$; otherwise $y = 0$. Note that by sliding the reference point 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. In a real online setting, the model should be retrained periodically (e.g., every 24 hours) based on the latest file access times in order to ensure that the model stays relevant with any shifts in the file access patterns.

B. Learning Model Selection

XGBoost [13], a state-of-the-art gradient boosting tree algorithm, was selected as our learning model because it satisfies two key requirements: (i) the model is able to *accurately* predict whether a file will be accessed soon or not accessed for some time; and (ii) both model training and predictions are *inexpensive* in terms of computational and storage needs. 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 [13].

We also examined other well-established classifiers but each failed to satisfy some of our needs. Specifically, Naive Bayes assumes that features are conditionally independent of one another, and thus cannot be used effectively to learn a sequence of time accesses. Bayesian Belief Networks model attribute dependence in networks, but require a priori knowledge about the structure of the network, which is what we are trying to determine. Finally, we empirically found Support Vector Machines and Artificial Neural Networks 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, efficient to use, and accurate in making predictions*, all of which are validated in our experimental evaluation in VI-A.

C. File Access Predictions

In order to make predictions using the generated model, the file system must first create a feature vector dynamically. This is easy to do via (1) setting the reference point to the current time; (2) discretizing time using either the fixed or growing intervals approach; and (3) creating the features based on the file size, creation time, and access times, as explained in Section III-A. Next, the feature vector is used to probe the model and get the predicted class label. Specifically, an XGBoost model will return a *probability score* indicating how likely the file is to be accessed in the next class window.

The probability score is used by the policies to decide which file(s) to add or remove from the cache. We generate two separate models for this purpose, one for the admission and one for the eviction policy, whose only difference lies in the class window size w. The admission policy wants to determine which files will be accessed in the immediate future and, hence, we set a small w (e.g., 30 minutes). On the other hand, the eviction policy wants to determine which files will not be accessed for some time and, hence, we set a large w (e.g., 6 hours). The two policies are elaborated in Sections IV and V.

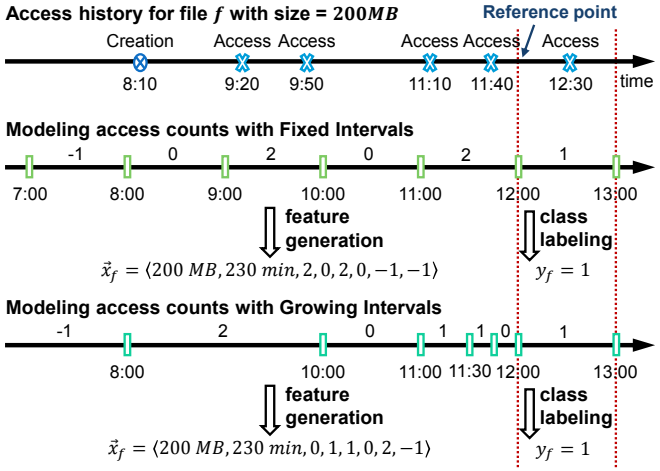

Fig. 2: Training data preparation for two discretization approaches: fixed and growing time intervals

IV. Cache Eviction Policies

An object is evicted from a traditional cache when a new one needs to enter but the cache is full. This approach works well for fixed-size disk pages or small web objects. However, typical file sizes in analytics clusters are in the order of MBs-GBs [12], so having a file operation wait for other files to be evicted would introduce significant delays. Hence, our policies start the eviction process *proactively* when the cache becomes fuller than a threshold value (e.g., 90%), allowing for better overlapping between file operations and evictions. Similarly, the policies will stop the eviction process when the cache capacity becomes lower than a threshold value (e.g., 85%), allowing for a small percent of the cache to be freed together.

Once the eviction process starts, the policy must select a file to remove from the cache in order to make room for new files. For comparison purposes, we have implemented three conventional eviction policies, one related policy from recent literature, and one new policy, listed in Table I.

LRU (Least Recently Used) selects the file used least recently, trying to take advantage of the temporal locality typically exhibited in data accesses.

LFU (Least Frequently Used) selects the file with the least number of accesses, i.e., it evicts rarely used files.

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 it is updated each time f is accessed based on Formula 1:

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

TABLE I: Cache eviction policies

Acronym	Policy Name	Description
LRU	Least Recently Used	Evict the file that was accessed less recently than any other
LFU	Least Frequently Used	Evict the file that was used least often than any other
LRFU	Least Recently & Frequently Used	Evict the file with the lowest weight based on recency and frequency
EXD	Exponential Decay (Big SQL [10])	Evict the file with the lowest weight based on recency and frequency
XGB	XGBoost-based Modeling	Evict the file with the lowest access probability in the distant future

TABLE II: Cache admission policies

Acronym	Policy Name	Description
OSA	On Single Access	Cache a file into memory upon access (if not there already)
LRFU	Least Recently & Frequently Used	Cache a file if its weight is higher than a threshold
EXD	Exponential Decay (Big SQL [10])	Cache a file if its weight is higher than the weight of to-be-evicted files
XGB	XGBoost-based Modeling	Cache files with high access probability in the near future

Parameter H represents the "half life" of W, i.e., after how much time the weight is halved. Hence, files that are recently accessed multiple times will have a large weight, as opposed to files accessed a few times in the past.

EXD (Exponential Decay) explores the tradeoff between recency and frequency in data accesses in Big SQL [10]. In particular, it selects the file with the lowest weight W computed using the following formula:

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

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

XGB (XGBoost-based Modeling) utilizes an XGBoost model (recall Section III) for predicting which file will not be accessed in the distant future. Specifically, XGB computes the access probability for the k LRU files and selects the file with the lowest probability to evict. 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.

V. Cache Admission Policies

In a traditional cache setting, all data accesses must go through the cache first: if the accessed object is not found, then it will be inserted into the cache. Cache admission policies are rare in such a setting as they complicate the read process without major benefits [10]. In our case, however, placing a file into the cache is costlier as it may involve accessing a large amount of data. Hence, the decisions of when and what to cache are as important as when and what to evict. The admission process is invoked (i) every time a file is accessed and (ii) periodically in case a policy wants to make a proactive decision. For comparison purposes, we have implemented two conventional admission policies, one related policy from recent literature, and one new policy, listed in Table II.

OSA (On Single Access) implements the common approach of caching each file when it is accessed and not already present in the cache.

LRFU (Least Recently & Frequently Used) caches an accessed file when its computed weight (recall Formula 1) is greater than a threshold value, which is empirically set to 3 for favoring files that are accessed recently multiple times.

EXD (Exponential Decay) is used in Big SQL [10] for selecting which files to insert into the cache. If there is enough space to fit the accessed file f, then f will get cached. Otherwise, EXD will cache f only if its weight (computed using Formula 2) is higher than the sum of weights of the files that will need to be evicted to make room for f.

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

VI. Experimental Evaluation

The evaluation is conducted on a 12-node cluster running CentOS Linux 7.2. with 1 master and 11 slave nodes. Each node has a 64-bit, 8-core, 2.4GHz CPU, 24GB RAM, and three 500GB SAS HDDs. The available memory for HDFS caching is set to 4GB, the default replication factor is 3, and block size is 128MB. We modified HDFS v2.7.7 to include the AutoCache components. For our modeling, we used XGBoost2 v0.60 and set the learning objective to be logistic regression for binary classification.

Our goal is to evaluate (i) the prediction accuracy and efficiency of our XGBoost models, and (ii) the effectiveness of our approach in improving performance for a workload derived from real-world production traces from Facebook. The traces were collected over a period of 6 months from a 600-node Hadoop cluster and contain data about MapReduce jobs (e.g., arrival times, data sizes) [14]. We used SWIM [15] to generate and replay a realistic and representative workload that preserves the original workload characteristics such as distribution of input sizes and skewed popularity of data [9].

TABLE III: Job size distributions, binned by their data sizes

Bin	Data size	% of Jobs	% of Resources	% of I/O
A	0-128MB	74.4%	25.0%	3.2%
B	128-512MB	16.2%	12.2%	16.1%
C	0.5-1GB	4.0%	7.3%	12.0%
D	1-2GB	3.0%	13.4%	19.3%
E	2-5GB	1.6%	20.8%	21.9%
F	5-10GB	0.8%	21.4%	27.5%

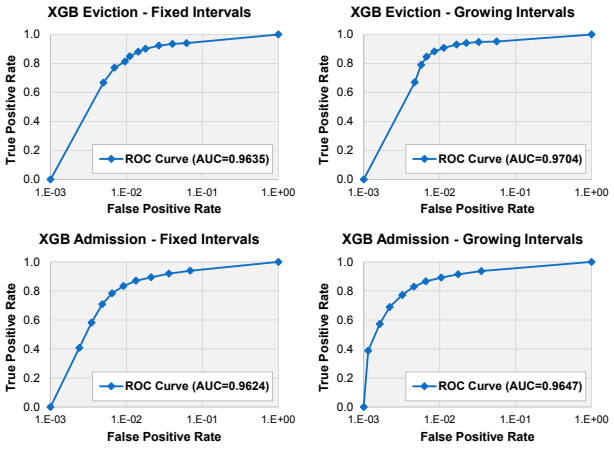

Fig. 3: ROC curves for XGB eviction/admission models

The workload consists of 1000 jobs scheduled for execution over a 6-hour period, processing 1380 files with a total size of 92GB. To separate the effect of caching on different jobs, we split them based on their input data size into 6 bins. Table III shows the distribution of jobs by count, cluster resources they consume, and amount of I/O they generate. The jobs exhibit a heavy-tailed distribution of input sizes, also noted in [9], [12]. Specifically, the workload is dominated by small jobs (74.4%) that process <128MB of data. Yet, they only account for 25% of the resources consumed and perform only 3.2% of the overall I/O. On the contrary, jobs processing over 1GB of data account for over 54% of resources and over 68% of I/O. The popularity of files is also skewed, with a small fraction of the files accessed very frequently [9], [12]. For example, 5.7% of files are accessed more than 5 times. This repeatability must be exploited by ensuring such files are present in the cache.

A. 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) [16]. The ROC curve takes as input the probabilities of file accesses 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 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 sequential data set into training (first 4 hours), validation (5th hour; used for early stopping during training to avoid overfitting), and test (6th hour) sets. For the fixed-intervals approach, we discretized time into twelve 30-minute intervals before the reference point, while for the growing-intervals approach we used 6 intervals, starting from a 15-minute interval and doubling it each time (recall Section III-A). The class window sizes for the eviction and admission policies were set to 20 minutes and 90 minutes, respectively.

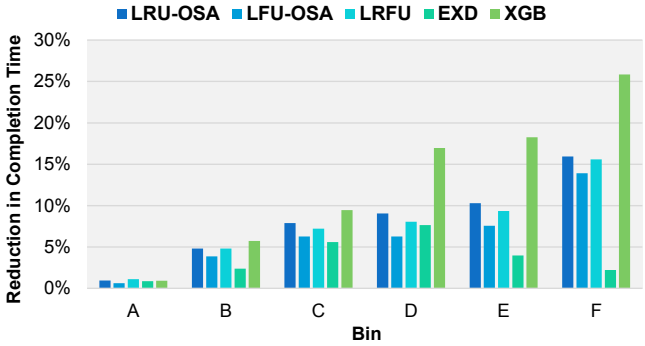

Fig. 4: Percent reduction in completion time over HDFS

Figure 3 shows the 4 ROC curves for the eviction and admission models for our two approaches: using fixed and growing time intervals (note the logarithmic scale of the x-axis). In all four cases, the curves are near point $(0, 1)$ with AUC values higher than 0.96 (1 is the max), which indicate the very high prediction performance of our models. The growing-intervals models are slightly better compared to the fixed-intervals models, offering $\sim$1% and $\sim$2% better accuracy and precision, respectively.

Training an XGBoost model given a 5-hour workload trace takes about 5.3 seconds in total. Using the model to make a single prediction takes 1.8ns. Overall, during the entire workload run, selecting a file to evict or admit into the cache amounts to 0.49 CPU seconds; showcasing the negligible CPU overhead caused by XGBoost for both training and prediction.

B. Workload Performance Evaluation

We executed the derived workload over default HDFS and the modified HDFS with the eviction and admission policies listed in Tables I and II. Since LRU and LFU do not have corresponding admission policies, we paired them with the OSA policy. For XGB, we only present results using the growing intervals models as they behave very similarly to the fixed intervals models. We compare executions 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 [9]. Even though cache hit ratio is a popular metric for evaluating caching policies, we did not use it as previous work has shown that maximizing hit ratio neither minimizes job completion time nor maximizes cluster efficiency [9].

Figure 4 shows the reduction percentage in job completion time compared to HDFS for each bin (recall Table III). Small jobs (Bins A, B) experience only a small improvement (<5%) in completion time for all policies. This is not surprising since time spent in I/O is only a small fraction compared to CPU and scheduling overheads. The gains in job completion time

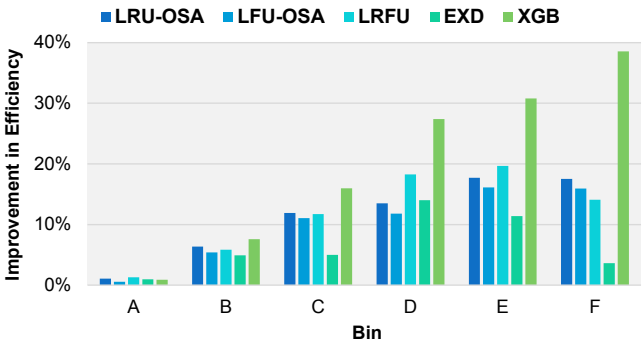

Fig. 5: Percent improvement in cluster efficiency over HDFS

increase as the input size increases, while we start observing *different behavior across the policies*. Specifically, LRU-OSA and LRFU are performing well as the workload exhibits good temporal locality of reference, resulting in up to 15% reduction in completion time for large jobs (Bin F). LFU-OSA trails them with only 2% difference. EXD performs well only for jobs in Bin D with 8% gains in completion time, while it performs poorly for larger jobs. Recall that EXD explores the tradeoff between recency and frequency without taking file size into account, which is an important factor in caching. Finally, our XGB policy is able to provide the *highest reduction in average completion time across all job bins*, with 16%-25% gains for large jobs, almost double compared to the second-best policy. Overall, XGB is able to effectively learn the different access patterns and detect data reuse across jobs.

With each cache access, the cluster efficiency improves. Figure 5 shows how this improvement 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 III). Across different policies, the trends for efficiency improvement are similar to the trends for completion time reduction discussed above: the conventional policies generally offer good benefits; EXD offers poor gains; and *XGB offers the best benefits*. Hence, improvements in cluster efficiency are often accompanied by lower job completion times, doubling the benefits. For example, XGB is able to reduce completion time of large jobs by 26% while consuming 39% less resources. Results from further analysis (not shown due to lack of space) reveal that XGB (i) results in the highest percentage of cache accesses and (ii) is the most selective in terms of admission.

VII. Related Work

The Google File System (GFS) [17] explicitly states that it relies on the local file system's buffer cache to keep frequently accessed data in memory. However, the local cache is not aware of the files that are distributedly stored and accessed in the cluster, and hence is missing out on potential cluster-wide performance optimizations. Other distributed file systems, such as *HDFS* [3] and *OctopusFS* [7], now support storing files in memory, but they do not offer any support for automatically adding or removing files from it. In-memory file systems (e.g., *Alluxio* [5], *GridGain* [6]) can also be used for storing or caching data in clusters, while Spark enables jobs to persist a specified dataset in memory [2]. However, these system only use conventional cache eviction policies (e.g., LRU) and rely on the user to manually cache the data.

PACMan [9] is a memory caching system designed for data-intensive parallel jobs. PACMan implements two eviction policies, one that prioritizes small inputs and one that evicts less frequently accessed files. However, PACMan does not allow jobs to specify hot data in memory and does not implement cache admission policies. *Big SQL* [10] is an SQL-on-hadoop system that uses HDFS cache for caching table partitions. Big SQL proposes two algorithms, namely *SLRU-K* and *EXD*, that explore the tradeoff between 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.

Caching is a well-studied problem that appears in various contexts and discussed extensively in several surveys [8], [18]. In the context of CPU caches and database buffer caches, there is extensive work on *cache eviction* policies such as LRU, LFU, ARC, and MQ. Unlike our approach, these policies operate on fixed-size pages and assume that every accessed page will be inserted into the cache. Web caching policies (e.g., SIZE, Hyper-G, Greedy-Dual-Size [8]) operate on variable size objects but are designed to improve hit ratio, which does not necessarily improves performance in a large-scale cluster environment [9].

Machine learning techniques such as logistic regression and artificial neural networks have also been used for developing better web caching policies [8], [19]. 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) [11]. However, such relationships and associations are not expected to be present in big data analytics workloads and, hence, are not applicable in our setting. LeCaR [20] models cache eviction as an online learning problem involving regret minimization between LRU and LFU, but it was designed for caches that are significantly smaller than the working set. More recently, deep learning has been used in eviction policies for content caching [21] and admission policies for object store caching [22], which try to learn temporal relationships between requested objects. Finally, none of the aforementioned approaches take advantage of access patterns for automatically managing a DFS cache.

VIII. Conclusions

This paper describes AutoCache, a framework for automatically admitting and evicting files from a cache in a distributed file system (DFS). Our cache policies employ light-weight gradient boosted trees for learning file access patterns and predicting which files should be cached or uncached. The framework and policies have been implemented in a real DFS and successfully evaluated over real-world workload traces.

REFERENCES

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. of the 4th Symp. on Cloud Computing (SoCC)*. ACM, 2013, pp. 5–21.

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

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. of the 26th IEEE Symp. on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

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

[5] "*Alluxio: Open Source Memory Speed Virtual Distributed Storage*," 2018. [Online]. Available: https://www.alluxio.org/

[6] "*GridGain In-Memory Computing Platform*," 2018. [Online]. Available: https://www.gridgain.com/

[7] 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)*. ACM, 2017, pp. 65–78.

[8] W. Ali, S. M. Shamsuddin, and A. S. Ismail, "A Survey of Web Caching and Prefetching," *International Journal of Advances in Soft Computing and its Applications (IJASCA)*, vol. 3, no. 1, pp. 18–44, 2011.

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

[10] 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)*. ACM, 2016, pp. 321–333.

[11] M. Mesnier, E. Thereska, G. R. Ganger, and D. Ellard, "File Classification in Self-* Storage Systems," in *Proc. of the 1st Intl. Conf. on Autonomic Computing (ICAC)*. IEEE Computer Society, 2004, pp. 44–51.

[12] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads," *Proc. of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[13] 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)*. ACM, 2016, pp. 785–794.

[14] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance using Workload Suites," in *Proc. of the 19th Intl. Symp. on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2011, pp. 390–399.

[15] "*SWIM: Statistical Workload Injector for MapReduce*," 2016. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki

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

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[18] S. Podlipnig and L. Böszörmenyi, "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[19] P. Venketesh and R. Venkatesan, "A Survey on Applications of Neural Networks and Evolutionary Techniques in Web Caching," *IETE Technical review*, vol. 26, no. 3, pp. 171–180, 2009.

[20] 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, pp. 928–936.

[21] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A Deep Reinforcement Learning-based Framework for Content Caching," in *Proc. of the 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 2018, pp. 1–6.

[22] E. Ofer, A. Epstein, D. Sadeh, and D. Harnik, "Applying Deep Learning to Object Store Caching," in *Proc. of the 11th ACM Intl. Systems and Storage Conference (SYSTOR)*. ACM, 2018, pp. 126–126.