future internet

MDPI

Article

Employing Streaming Machine Learning for Modeling Workload Patterns in Multi-Tiered Data Storage Systems †

Edson Ramiro Lucas Filho [1], George Savva [1], Lun Yang [2], Kebo Fu [2], Jianqiang Shen [2] and Herodotos Herodotou [1,*]

[1] Department of Electrical Engineering and Computer Engineering and Informatics, Cyprus University of Technology, Limassol 3036, Cyprus; edson.lucas@cut.ac.cy (E.R.L.F.); gec.savva@edu.cut.ac.cy (G.S.)
[2] Huawei Technologies Co., Ltd., Shenzhen 518100, China
* Correspondence: herodotos.herodotou@cut.ac.cy
† This paper is an extended version of our paper published in Lucas Filho, E.R.; Yang, L.; Fu, K.; Herodotou, H. Streaming Machine Learning for Supporting Data Prefetching in Modern Data Storage Systems. In Proceedings of the First Workshop on AI for Systems (AI4Sys '23), ACM, 2023, pp. 7–12. https://doi.org/10.1145/3588982.3603608.

Abstract: Modern multi-tiered data storage systems optimize file access by managing data across a hybrid composition of caches and storage tiers while using policies whose decisions can severely impact the storage system's performance. Recently, different Machine-Learning (ML) algorithms have been used to model access patterns from complex workloads. Yet, current approaches train their models offline in a batch-based approach, even though storage systems are processing a stream of file requests with dynamic workloads. In this manuscript, we advocate the streaming ML paradigm for modeling access patterns in multi-tiered storage systems as it introduces various advantages, including high efficiency, high accuracy, and high adaptability. Moreover, representative file access patterns, including temporal, spatial, length, and frequency patterns, are identified for individual files, directories, and file formats, and used as features. Streaming ML models are developed, trained, and tested on different file system traces for making two types of predictions: the next offset to be read in a file and the future file hotness. An extensive evaluation is performed with production traces provided by Huawei Technologies, showing that the models are practical, with low memory consumption (<1.3 MB) and low training delay (<1.8 ms per training instance), and can make accurate predictions online (0.98 F1 score and 0.07 MAE on average).

Keywords: multi-tiered data storage systems; streaming machine learning; workload patterns

Academic Editor: Michael Sheng

Received: 13 March 2025
Revised: 8 April 2025
Accepted: 9 April 2025
Published: 11 April 2025

Citation: Lucas Filho, E.R.; Savva, G.; Yang, L.; Fu, K.; Shen, J; Herodotou, H. Employing Streaming Machine Learning for Modeling Workload Patterns in Multi-Tiered Data Storage Systems. *Future Internet* **2025**, *17*, 170. https://doi.org/10.3390/fi17040170

1. Introduction

Modern data storage systems, such as Huawei OceanStor Dorado, Dell Unity XT, and HPE Alletra Storage, optimize data access by distributing data through a hybrid composition of caches and storage tiers composed of different storage media devices (e.g., NVRAM, SSD, HDD). Each action of a cache (e.g., admit, evict, prefetch) or tier (e.g., migrate up, migrate down) is governed by different policies, which individually influence the performance of the entire system. Such policies are used, for instance, to mitigate the latency of retrieving data, increase the lifespan of flash memory devices, and ensure enough space is available in a cache or tier to store new data. Therefore, caching and tiering policies are fundamental for building modern multi-tiered storage systems. However, the

performance of such policies is susceptible to the storage workload, often generated by numerous applications accessing thousands of files in parallel. While there have been several efforts to understand and exploit file access patterns on data storage systems to benefit caching and tiering policies [1–8], the growing number of storage workloads with crescent complexity, such as machine learning, scientific workflows, and data science, poses a challenge to keep these policies up-to-date.

One solution to this challenge is to enable data storage systems to learn workload patterns. In recent years, there have been many efforts to support caching and tiering policies using different machine-learning algorithms. Even though such policies can be used at various levels of the storage's software stack and operate on different data abstractions (e.g., memory pages, disk blocks, files), the employment of Machine Learning (ML) is similar across them. For instance, algorithms such as Neural Networks, Decision Trees, Random Forests, and Support Vector Machines are commonly trained with a set of data storage I/O requests to learn the workload patterns and then predict the value of a certain attribute of a page, block, file, or object (e.g., access frequency, next offset) that is required by a policy to make its decisions [9–15]. Even though modern data storage workloads reflect continuous and dynamic streaming of data requests, current approaches learn workload patterns by training their models offline in a batch-based fashion. This drives the need for systems to retrain their models regularly; a process that demands extra space to store new training data as well as additional processing to update the model, thereby hurting system performance. Deep Learning (DL) [16–18] further exacerbates the issue as it requires significant computational resources at scheduled intervals, leading to inefficient resource utilization. In addition, the models can become outdated between training cycles, reducing predictive accuracy for dynamic environments [19].

In this work, we investigate the use of streaming machine learning (as opposed to traditional batch-based models) to learn workload patterns that are useful for caching and tiering policies in multi-tiered, file-based storage systems. Streaming ML involves training models online by processing one training instance at a time and introduces various advantages, including (i) high efficiency during both training and using the models, (ii) high accuracy in predictions, and (iii) high adaptability to changing workload patterns [20]. We perform our analysis over a set of production traces provided by Huawei Technologies. First, we identify and exploit file access patterns, including temporal, spatial, length, and frequency patterns at different granularity (i.e., file, directory, and file format), for extracting data features reflecting various spatiotemporal and semantic associations. These features are extended with other basic file and request information (e.g., file size, request offset) typically used for this purpose by prior work. The features are then used in real-time to build streaming ML models within a new online learning framework. To show the generality of the framework, models are developed for making two types of predictions. The first one involves predicting the next file offset to be read by a future file request in a particular file. The second one involves predicting the future file hotness of a particular file (i.e., a metric that determines the file's future access rate and is defined concretely in Section 4.1). For evaluation purposes, one regression and one classification model are built for each type of prediction, totaling four models.

In summary, this paper makes the following contributions:

- We show that file access patterns calculated per file, parent directory, and file format are strong features for ML models that predict the presented target variables.
- We present and evaluate a set of representative and robust features to be used with streaming ML models for predicting the next offset and future file hotness.
- We describe an architecture for an online learning framework for generating and testing training instances efficiently and in real time while the system is running.

- We show, using real production workloads, that streaming machine-learning models are vital in supporting the high and continuous number of I/O requests with evolving access patterns present in modern data workloads.

In our preliminary work [21], we explored using one streaming classification model to predict the next file offset to be read in a file. In this work, we (1) generalize the application of streaming ML to multiple targets (i.e., next offset and future file hotness) using both classification and regression; (2) formalize the file access patterns as features; (3) present the conceptual architecture of an online learning framework; and (4) provide an extensive experimental evaluation with comparisons to other rule-based, machine-learning, and deep-learning models.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents the file access patterns, while Section 4 discusses feature extraction and selection. Section 5 presents the online learning framework. Section 6 presents model selection and hyperparameter tuning. Section 7 presents the experimental evaluation. Section 8 discusses the potential limitations of the proposed approach along with future directions. Finally, Section 9 concludes the paper.

2. Related Work

There have been several efforts to support caching and tiering policies by learning workload patterns from data storage workloads using Machine Learning (ML). Even though these approaches are applied at different levels of the storage's software stack, the usage of ML is similar across them. In particular, it is used to predict a certain attribute of a page [14,17,22–32], block [12,15,18,33–38], file [9–11,39–46], or object [13,47,48], where the target attribute varies according to the strategy of the policy (e.g., next offset, access frequency, next access time, associated items). In order to predict such targets, each approach employs a set of features for representing the workload patterns, commonly derived from different sources such as metadata, data requests, and historical information.

Notably, a subset of features and attributes to be predicted are common across systems. For instance, caching and tiering policies that operate on blocks are generally aimed at predicting the next offset [37,49], next block [12,15,18,24,26,27,36], or correlated blocks [35] to be accessed in the near future. The most common features used include the requested *offset*, *length*, *last access time*, and the *access frequency* of blocks. The most common ML techniques applied are Markov Models and Neural Networks. Policies that operate over files are usually aimed at predicting the access frequency. The most common features are the attributes of files and objects such as *name*, *permissions*, *type*, *creation time*, *size*, and the requested *offset* and *length* [11,14,16,31,39–42,50–54]. The most common ML algorithms employed are Reinforcement Learning, Decision Trees, and Neural Networks.

We model the prediction of the next offset to be accessed and future hotness of a file as both regression and classification to demonstrate the generality of the proposed approach. To this end, we built a set of representative features that mix attributes commonly used by the related work (e.g., file name, creation time, requested offset, request length) and new attributes such as the access patterns presented in Section 3.2 that can be identified online (e.g., sequential reads). In our feature evaluation (Section 4.3), we demonstrate that access patterns are representative features that help ML models in their predictions.

The ML models employed by prior work, such as Markov Models, Neural Networks, and Decision Trees, are batch-based models that can provide high-accuracy solutions but require offline training and testing. Some systems retrain their models from time to time, or employ incremental learning, in an effort to meet an online demand [11,32,40,48]. However, retraining and updating the models periodically can lead systems to underperform in the presence of new workloads, due to models being outdated for certain time periods.

Systems that perform offline training and testing must also store massive historical datasets. On the other hand, streaming ML can provide high-accuracy models by training, testing, and evaluating models online. Streaming ML models are able to capture changes in the underlying data distribution they model (i.e., data access patterns) and naturally adjust to workload changes over time with minimal overhead. Also, they significantly lower the need for historical datasets.

Most of the related work attempts to learn and predict the next action in the workload (e.g., next block to be read, next offset to be accessed, the probability of a file to be accessed) and commonly uses similar features to perform these predictions. Similar to the related work, we aim to predict similar targets such as the *next offset* to be accessed and the *hotness* of a file to be read in the near future. However, we identify an effective set of features to predict both target variables that includes access patterns (e.g., sequential reads, uniform lengths) to be used with streaming ML models. To the best of our knowledge, we are the first to systematically use access patterns as features, as well as streaming ML to support caching and tiering policies.

Streaming ML is becoming popular for continuously training models for various industrial applications, including cyber security, IT infrastructure maintenance (AIOps), anomaly scoring, and drift detection [55]. One work utilized steaming ML to develop a data mining-based horizontal fragmentation method in Data Warehouses to optimize query response time and system efficiency [56], while another work studied the combined problem of the system configuration and hyperparameter tuning of ML applications over distributed stream processing engines such as Apache Spark Streaming [57]. Streaming ML has also been deployed in other domains, such as in social computing for detecting online aggression on social media [58], in urban management for IoT-enabled waste management in smart cities [59], and in electronic health for online estimation and inference of treatment effects [60].

Several streaming ML libraries are available, each with unique features. Massive Online Analysis (MOA) [20] offers a collection of well-established online algorithms for streaming classification, clustering, and change detection. Vowpal Wabbit [61] is a streaming ML framework built on the perceptron algorithm with a strong emphasis on reinforcement learning. Jubatus [62] provides stream mining capabilities by tightly integrating its ML library with a custom-built Distributed Stream Processing Engine (DSPE). streamDM [63] is a data mining framework that implements streaming ML algorithms specifically for Spark Streaming. Lastly, Apache SAMOA [64] supports distributed ML computation across multiple DSPEs, including Storm, Flink, Samza, and Apex.

3. File Access Patterns

In this section, we present the input trace characteristics used in our analysis and discuss how we model a set of representative access patterns that are used as features for the streaming ML models.

3.1. Input Traces

We perform our analysis over a set of 17 different production traces provided by Huawei Technologies and collected from hybrid (i.e., multi-tiered) data storage systems of the Huawei OceanStor Dorado and Pacific Series. Each input trace consists of a list of file I/O requests to be processed by the storage system. Each request consists of a *process id* of the application submitting the request, an epoch *timestamp* of submission, a file *operation* (e.g., open, read, write, close), a full *file path name*, a request *offset* (in bytes), a request *length* (in bytes), the *file size* (in bytes), and the *duration* of the operation (in microseconds). The traces exhibit a very high degree of variability in their characteristics: the number of file

requests ranges from 3 K up to 6.7 M, originating from 1 up to 44 K applications running in parallel, accessing from 17 files up to 860 K files, with individual file sizes ranging between 0 bytes and 32 GB. In addition, 7 out of 17 traces are read-heavy (containing >90% read requests), while 4 traces are write-heavy (<50% read requests). After investigating the read requests, we observe that only 5 traces perform predominately sequential reads (>90%), while 7 traces perform a majority (>50%) of random reads. Table 1 summarizes the information about the traces, including the number of files accessed, the total number of requests, and a breakdown per request type (i.e., open, close, read, write). Overall, the traces exhibit very diverse and complex access patterns. Given that the proposed features and models work well across all 17 traces (as shown in Section 6), we believe they are generalizable to other workloads and systems.

Table 1. Breakdown of file request operations per trace, including the number of files and applications, the number of file requests per request type, and the distribution of random and sequential read/write requests.

Trace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
# Total Files	66	26	27	38	3376	17	3764	4498	2725	4464	4591	79	21	72	860,465	26,684	56,321
# Applications	1	24	2	60	18	1	66	131	43	849	104	40	6	1	341	3077	43,928
# File Requests	3328	5622	49,923	41,696	40,372	71,528	126,849	213,983	197,161	217,004	217,076	1,496,468	4,050,788	3,216,929	4,422,753	6,093,193	6,767,326
# Open Requests	111	47	20	7524	12,127	16	39,513	79,755	32,672	76,042	38,421	151	19	276	1,469,542	43,559	822,047
# Close Requests	111	47	20	7524	12,127	16	39,513	79,755	32,672	76,042	38,421	151	19	276	1,469,542	43,559	822,047
# Read Requests	3106	5513	113	25,448	14,435	48,859	40,825	52,346	65,979	64,772	64,592	404,013	45	3,196,567	1,473,466	5,128,388	3,908,841
# Write Requests	0	15	49,770	1200	1683	22,637	6998	2127	65,838	148	75,642	1,092,153	4,050,704	19,810	10,203	877,687	1,214,391
Read Ratio (%)	100	99.73	0.23	95.50	89.56	68.34	85.37	96.10	50.05	99.77	46.06	27.00	0.00	99.38	99.31	85.39	76.30
Seq. Reads (%)	97.30	96.90	89.38	21.15	23.47	99.81	25.87	16.49	44.89	26.59	62.17	99.97	71.11	76.49	0.09	97.65	85.28
Seq. Writes (%)	0.00	60.00	100	98.17	91.74	99.97	0.00	73.95	99.99	0.00	99.82	100	100	98.25	98.53	97.93	95.42
Duration (min)	14.0	84.6	3.8	122.6	11.3	3.7	1.8	37.2	23.5	46.1	19.0	3.1	8.4	14.0	17.1	61.3	43.4

3.2. File Access Patterns

A sequence of file requests can exhibit various access patterns, which we organize into four major groups: (i) *temporal*, (ii) *spatial*, (iii) *length*, and (iv) *frequency* patterns. For every new file request, we extract the required information and compute each listed pattern along with how frequently each access pattern appears.

The *temporal pattern* represents the frequency in which a file receives read/write I/O requests over time. Files can be accessed based on an *intense* frequency (i.e., each request is submitted almost immediately after the previous one) or a *mild* frequency (i.e., there is some small delay between request submissions). The temporal pattern requires storing the last n request times per file to be calculated. Consider $\mathbb{T}_f$ as the set of n previous request times for a file f. Thus, the temporal pattern is given by:

$$\delta(f) = \begin{cases} \text{intense}, & \forall t_i \in \mathbb{T}_f : |t_i - t_{i-1}| \leq \Delta, \Delta > 0 \\ \text{mild}, & \text{otherwise} \end{cases} \tag{1}$$

where Δ is a configurable length gap between consecutive requests. Whenever $|\mathbb{T}|$ reaches the size of n, we remove the last item of $\mathbb{T}$.

The *spatial pattern* represents the order in which the data are accessed in a file. Files can receive *sequential* and *non-sequential* (or *random*) I/O requests over their data. A sequential spatial pattern represents a file that has its data accessed contiguously, without any gap in the accessed data requested between file requests. Consider $\mathbb{H}_f$ as the access history of file f, containing the pairs (o, l), where o is an offset and l is a length. Also consider n as the size of H. The spatial pattern of f is given by:

$$\omega(f) = \begin{cases} \text{sequential}, & \forall (o_i, l_i) \in \mathbb{H}_f : o_i = o_{i-1} + l_{i-1} \\ \text{random}, & \text{otherwise} \end{cases} \tag{2}$$

When $|\mathbb{H}|$'s size reaches n, the last (oldest) pair in $\mathbb{H}$ is removed.

The *length pattern* represents the variability of the length size requested to be read from or written to a file. This pattern determines whether or not requests over a file are accessing equal-sized blocks of data. Thus, it can be of *uniform* length size or *variable* length size. $\mathbb{H}_f$ holds the history of the previous accessed *offset* (o) and *length* (l) of each file. The length interval is given by:

$$\lambda(f) = \begin{cases} \text{uniform}, & \forall (o_i, l_i) \in \mathbb{H}_f : |l_i - l_{i-1}| = 0 \\ \text{variable}, & \text{otherwise} \end{cases} \tag{3}$$

The *frequency patterns* provide an estimate of the number of times a file f has been accessed. We differentiate between two forms of frequency patterns. The first form is the *file access frequency*, represented by a number between 0 and 1, amortized by a logarithmic function. It requires storing a counter c_f per file f to count the number of times the file is open for reading and is given by:

$$\rho_1(f) = 1 - \left(\frac{1}{\log_2 (c_f + 2)} \right) \tag{4}$$

The intuition for $\rho_1(f)$ is to have the shape of a logarithmic function that starts from point $(0, 0)$, increases quickly for small c_f, and then converges to the value 1 as c_f goes to infinity. The second form is the *fully read frequency* metric, which calculates an approximation of how many times this file was fully read, normalized by the size of the file. It requires the total number of bytes accessed from a file (b_f) and the file size (s_f), calculated by:

$$\rho_2(f) = \left\lfloor \frac{b_f}{s_f} \right\rfloor \quad (5)$$

We aggregate these file access patterns into two additional sets of patterns to provide a greater context to the learning models. We aggregate by files under the same *directory* and by files with the same *file format* (e.g., text, audio, image, video, compression, executable, scientific, source code, etc.). For the temporal, spatial, and length patterns, we maintain counters for the access frequency of each pattern for each file under a parent directory. Then, we compute the ratio of each pattern over all patterns identified for that directory. For the frequency patterns, we count the number of distinct files in the directory and aggregate the files' fully-read frequencies. In the traces, we did observe some interesting patterns for some directories. For instance, files under the `/etc` directory are often very small files that are open and closed, or open, read once, and closed. We also noticed during our analysis that files with the same format sometimes tend to follow the same trends.

4. Feature Extraction and Selection

The objective for feature extraction and selection is to build a set of representative features that can be used by ML models for making various types of predictions, such as the *next offset* and *hotness* of a file. Based on the analysis of file access patterns discussed in Section 3, we extracted a set of 37 features, presented in Table 2 and discussed in Section 4.1, that are derived from information extracted from file requests, file metadata and patterns, directory patterns, and file format patterns. The target variables are presented next in Section 4.2, while feature evaluation is presented in Section 4.3.

Table 2. Combined score of *p*-value, Chi-squared, and Gini Importance for all the features across all traces. ✓indicates selected features.

Feature	**Next File Hotness**		**Next Offset**		**Next File Hotness Class**		**Next Offset Class**	
	Score	**Selected**	**Score**	**Selected**	**Score**	**Selected**	**Score**	**Selected**
Request								
Request Operation	0.77		2.20	✓	1.36		1.00	
Request Offset	2.24	✓	2.73	✓	1.71	✓	2.64	✓
Request Length	0.99		1.35	✓	1.33		1.72	✓
File								
File ID	1.33	✓	1.33	✓	1.73	✓	1.61	✓
File Size	1.89	✓	1.40	✓	2.05	✓	2.07	✓
Time Since Creation	0.90		1.11	✓	1.53	✓	1.21	
Time Since Last Access	1.03		1.25	✓	1.11		1.25	
Time Since Last Update	0.55		0.66		1.07		0.83	
History of Open Times	1.53		1.10		1.23		1.13	
File Hotness	1.90	✓	1.09	✓	2.13	✓	1.59	✓

Table 2. *Cont.*

Feature	Next File Hotness		Next Offset		Next File Hotness Class		Next Offset Class	
	Score	Selected	Score	Selected	Score	Selected	Score	Selected
File Access Patterns								
Length Pattern	0.99	✓	1.62	✓	1.59	✓	2.56	✓
Length Pattern Frequency	1.53	✓	1.56	✓	1.53	✓	2.11	✓
Spatial Pattern	1.10	✓	1.59	✓	1.60	✓	2.00	✓
Spatial Pattern Frequency	1.91	✓	1.72	✓	1.66	✓	2.19	✓
Temporal Pattern	1.17	✓	1.64	✓	1.47	✓	1.59	✓
Temporal Pattern Frequency	1.79	✓	2.17	✓	1.90	✓	2.00	✓
Access Frequency	1.83	✓	1.38	✓	1.84	✓	1.64	✓
Open Frequency	1.28	✓	1.10	✓	1.99	✓	1.47	✓
Fully Read Frequency	2.47	✓	2.00	✓	2.92	✓	2.51	✓
Directory Access Patterns								
Directory ID	1.59	✓	1.51	✓	2.15	✓	1.86	✓
Temporal Intense Ratio	1.88	✓	1.48	✓	1.88	✓	1.77	✓
Temporal Mild Ratio	2.05	✓	1.69	✓	2.08	✓	1.93	✓
Spatial Sequential Ratio	1.50	✓	1.11	✓	1.58	✓	1.56	✓
Spatial Random Ratio	1.65	✓	1.39	✓	1.81	✓	1.91	✓
Length Uniform Ratio	1.39	✓	0.90	✓	1.50	✓	1.42	✓
Length Variable Ratio	1.68	✓	1.46	✓	1.79	✓	1.61	✓
File Count	1.26	✓	1.18	✓	1.85	✓	1.70	✓
Access Frequency	1.56	✓	1.04	✓	1.36	✓	1.21	✓
File Format Access Patterns								
File Format ID	1.03	✓	1.13	✓	1.67	✓	1.46	✓
Temporal Intense Ratio	1.66		1.24		1.39		1.13	
Temporal Mild Ratio	1.77		1.36		1.52		1.13	
Spatial Sequential Ratio	1.32		0.71		1.17		1.01	
Spatial Random Ratio	1.47		1.08		1.42		1.30	
Length Uniform Ratio	1.32		0.76		1.27		1.05	
Length Variable Ratio	1.65		1.24		1.57		1.43	
File Count	1.10	✓	0.94		1.60	✓	1.36	✓
Access Frequency	1.40		0.86		1.12		0.90	

4.1. Feature Extraction

As file requests are executed on a storage system, the file metadata information is updated, and the various patterns discussed in Section 3 can be computed and stored at the level of files, directories, and file formats. This aggregated information is then used for generating the features presented in Table 2 in an online fashion, as explained later in Section 5. All features are normalized to values between 0 and 1 but with different normalization approaches. Next, we discuss the set of extracted features and their normalization, grouped by their granularity.

Request Features: These are collected at the level of individual file requests and include the *Request Operation*, *Request Offset*, and *Request Length*. The Request Operation is encoded with a specific number assigned per operation type: 1 for Open, 2 for Write, 3 for Read, 4 for Close, and 5 for Delete, normalized by dividing it by 5. The Request Offset and Request Length are normalized using the size of the file accessed by the request.

File Features: These are extracted from basic file metadata on a per-file basis, and they include the *file ID*, *file size*, *time since creation*, *time since last access*, *time since last update*,

history of open times, and *current file hotness*. They identify the file and aim to determine how frequently this file is accessed and/or updated. The file size is normalized by the fifth root, over the fifth root of the biggest file size found in the system (defaults to 32 GB). Different approaches to normalization were also considered. With min–max normalization, smaller file offsets and lengths would acquire very small normalized values that would be almost indistinguishable from each other. For instance, 80% of the files have a size of less than 8 GB, but they would be normalized to less than 0.25 with min–max normalization, while the remaining 20% would acquire values between 0.25 and 1. The fifth root normalization spreads the normalized values in a more representative fashion. In the above example, those 80% of the files are normalized between 0 and 0.76.

The history of open times reveals how frequently the file is opened. The size of the history is configurable and set to 10 by default. Timestamps are well-known to be bad features because they continuously grow over time. Hence, instead of using them directly, we use the time difference between each consecutive pair of open timestamps as features. All time differences are normalized using min–max normalization, where max is a configurable time window.

The *hotness* of a file represents the file's access rate over a defined time interval. Intuitively, the more a file is accessed and more recently, the higher its hotness value. The file hotness h is calculated as:

$$h(f) = \frac{\sum_{\forall t} read(f,t) \times weight(t)}{filesize(f)} \tag{6}$$

where $read(f,t)$ is the amount of data read from file f during time period t, $weight(t)$ is the weight of t, which is inversely proportional to the time difference between the current time and the beginning of t, and $filesize(f)$ is the size of f.

File Access Patterns Features: The current temporal, spatial, and length patterns are maintained as a class with a specific number representation (e.g., mild temporal is 0 and intense temporal is 1). In addition to the pattern itself, we also maintain the *pattern frequency*, which represents the frequency of the intense, sequential, and uniform identification of the temporal, spatial, and length patterns, respectively. The pattern frequencies are normalized with the following:

$$n_{pattern\_frequency}(c) = 1 - \frac{1}{\max(\log(c), 1)} \tag{7}$$

where c is the frequency of the intense, sequential, or uniform pattern. Finally, three more features are extracted: the *file access frequency*, *file open frequency*, and *fully read frequency*, which aim to quantify the frequency of access. These are normalized using min–max normalization.

Directory and File Format Features: These features are similar and hold an aggregated view of the access patterns for each directory and each file format. We hypothesize that there is often a predominant pattern in the directory or file format, and that this pattern would help the model in its predictions. Each directory and each file format maintains the same set of nine features: an *ID* representing the directory or file format; the *ratio* of each pattern identified over all files under the specified directory or file format; the *files count*, representing the total number of distinct files under the same directory or with the same file format; and the *access frequency*, counting the total access frequency of the files under the same directory or with the same file format. The file, directory, and file format IDs are normalized by

$$n_{mod}(x, M) = \frac{|x|\%M}{M} \tag{8}$$

where x is the hash value of the file name, directory, or file format and M is a configurable max number of files, root directories, and file formats, respectively. We set M = 1,000,000 for file IDs and $M = 100$ for directories and file formats. The counts and access frequency features are normalized by the fifth root, as was done for the file size discussed above.

4.2. Target Variables

Target variables represent the values to predict, such as the next offset to be read and the hotness of a file in the near future. Accurate prediction of these two targets can drive caching and tiering policies into making important decisions that optimize the performance of multi-tier storage systems. Next offsets can be used for prefetching data into the cache proactively, thereby reducing the access latency of future read requests. File hotness is a proxy of how popular a file will become, which is crucial when making cache eviction, cache admission, and tier migration decisions. The target variables are also calculated and added as a target column to the feature vectors to generate training instances.

The *Next Offset* target variable is a numeric value, and a natural solution is to model it as a regression. The advantage of doing so is that, if accurate enough, it can determine the next portion of the file to be accessed, helping a prefetching policy to make good decisions. However, a moderate or even a small error when predicting the next offset may mislead to different regions of the file. This motivates us to also model the prediction of the Next Offset as a 3-class classification problem by determining if (1) the next offset is continuous to the current offset, (2) the next offset is at a random file location, or (3) the file will not be read anymore; labeled as *sequential*, *random*, and *none*, respectively. Classification is a relaxation of the regression problem, yet provides sufficient information to help prefetching policies to take action in the presence of primarily sequential workloads.

The *Next File Hotness* is also a numeric value (computed using Equation (6)), represents the hotness of a file during a future time interval, and can be modeled using either regression or multi-class classification. The prediction of the next file hotness as a regression has the advantage of differentiating how hot (or cold) a file is, which can help cache and tiering policies to take actions, such as prioritizing files during data movement. Contrary to the prediction of the next offset, a small or moderate prediction error may not affect determining how hot the file will be in the near future. This motivates modeling the prediction of the next file hotness as a classification problem. We categorized the file hotness into six distinct classes, with the first class being the coldest and the sixth the hottest. The number of classes was determined based on the distribution of the normalized values of file hotness across all traces.

Overall, we modeled both problems as regression and classification to analyze their performance and differences, computing four target variables: *Next Offset* and *Next Offset Class* for predicting the next file offset with regression and classification, respectively, and *Next File Hotness* and *Next File Hotness Class* for predicting the future file hotness with regression and classification, respectively.

4.3. Feature Evaluation and Correlation Analysis

During feature evaluation, we analyze the impact of each feature for predicting the four target variables for each trace. Then, we extract a representative and robust subset of features that is sufficiently accurate for performing the predictions across all 17 traces, giving us confidence that these features will also work with other traces. To do so, we use four different and widely used statistical metrics: (i) the *p-value test*, (ii) the *Chi-squared test*, (iii) the *Gini Importance*, and (iv) the *Correlation Matrix*. Each test provides a different perspective on the relationship between features and the target variable, and using them together can lead to more robust and reliable feature selection. The p-value test determines

the features that have a strong relationship ($p \leq 0.05$) with the target variable, the Chi-squared test indicates the probability of predicting values closer to the testing samples, the Gini Importance metric (calculated from a Random Forest model) determines the importance of each feature in predicting the target variable, and the Correlation Matrix expresses the correlation of extracted features between them.

Next, we present the detailed results of the four metrics for the Next Offset target. The results for the other target variables are similar. According to the *p*-value metric, the *File Size*, *History of Open Times 10*, *Request Offset*, *File Fully Read Frequency*, and *File Temporal Pattern Frequency* are highly important to predict the Next Offset. The connection between some of these features is clearly explained by their nature. For instance, the *Request Offset* is often correlated with the next offset because of the high frequency of sequential operations. However, the connection between features such as the File Size or the History of open times with $\Delta = 10$ is not so straightforward. This happens because the *p*-value might produce false positives due to incorrect connections with the target variable. For example, *History of Open Times 10* is frequently set to 0 because many files are not opened 10 times or more, while the (normalized) next offset is often very close to zero due to the many reads that start at the beginning of a file. Thus, employing the three metrics in conjunction is valuable for diminishing the possible false-positive effect from a single metric.

Following the *p*-value, the Chi-squared test also gave a high score for the *Request Offset* and *File Fully Read Frequency*. These features have higher scores in Traces 14–17, which are traces with a high number of sequential read requests. In contrast with the *p*-value metric, the *File Size* received a lower score from the Chi-squared test. It is interesting to observe that file access patterns received high scores from both metrics. For instance, the *File Length Pattern* and the *File Spatial Pattern* received a positive *p*-value score, while their weight received a high Chi-squared score. The access patterns related to files and directories received the highest scores, while the file format patterns received, in general, lower scores. This indicates that the file access patterns are correlated to the target variables and can help the prediction models. The key results from the Gini Importance are similar to the Chi-squared results and are not elaborated due to space constraints.

Finally, our correlation analysis has revealed that most of the features exhibiting a correlation are features related to the access patterns. This happens because many patterns are mutually exclusive. For instance, a file can only be currently either sequentially accessed or not. Note that the patterns for the File Format are typically correlated with the Directory patterns. This happens because many directories often host files with the same formats. The correlation is similar across all target variables.

The objective during feature selection is to decrease the number of features by removing non-representative or correlated features and leaving only a subset of the features that will yield good prediction results across all traces. After calculating the *p*-value, Chi-squared, and Gini Importance for each target variable across all traces, we combine the computed values to generate one single representation of each metric. For each tested feature, we add how many times it was selected by the *p*-value test across the 17 traces and normalize the sum to the 0–1 range (by dividing by 17). Next, we average the Chi-squared values per feature across the traces and perform min–max normalization to the 0–1 range. We repeat the latter process for the Gini Importance score. The final score is computed by summing the three normalized values per feature, and we use it to rank the most important features. Summation is used because it is simple, interpretable, and effectively balances feature importance without introducing bias. Additionally, since all scores are already normalized, summation naturally maintains a balanced aggregation without distorting the impact of high rankings. Finally, we use the correlation matrices to remove redundant features. Table 2 presents the combined scores for each feature, along with an indication

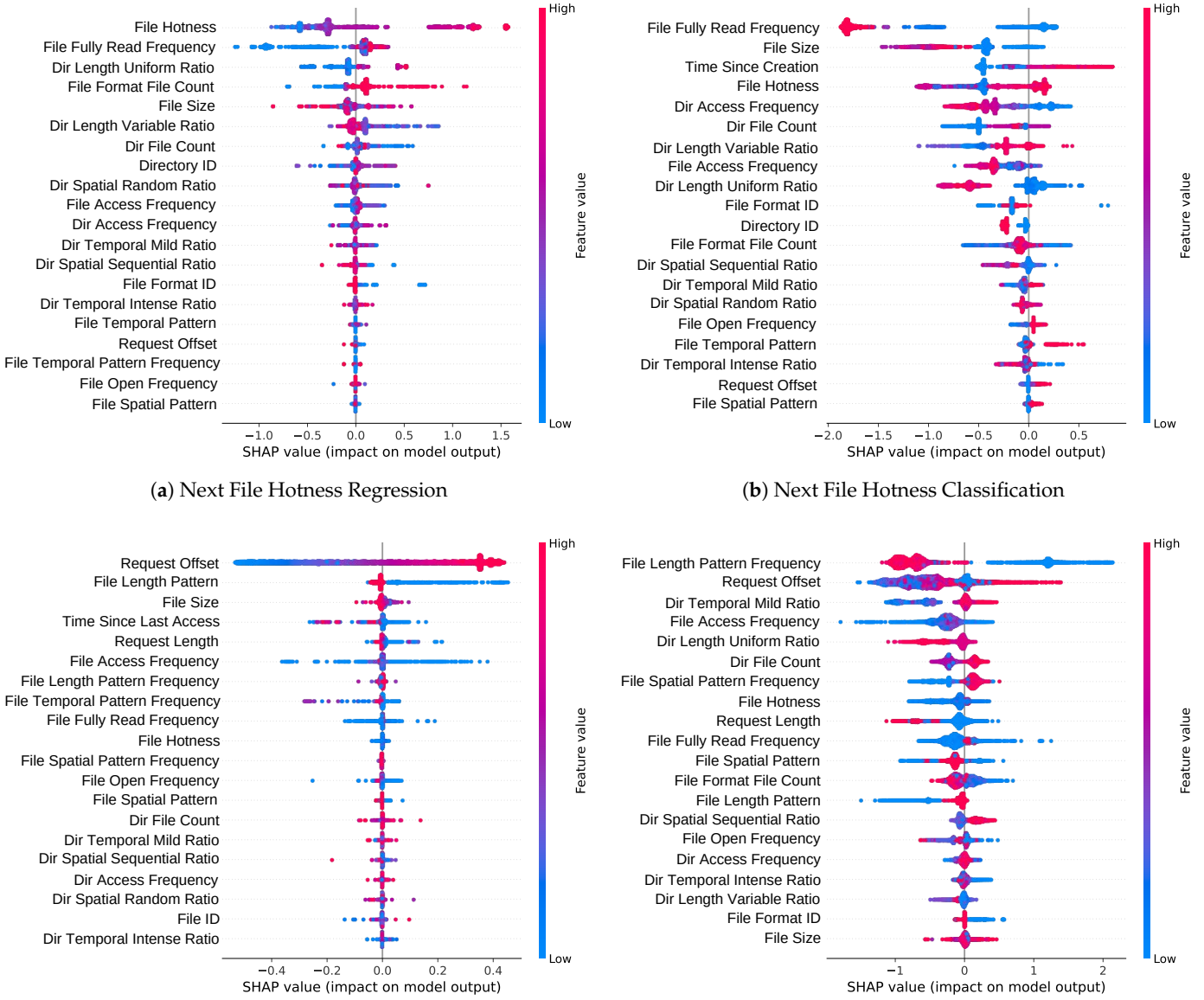
(✓) if the feature was selected during our feature selection process. Features with a final score of less than 1 exhibit low importance when predicting the target variables and thus can be discarded without loss of correctness in the prediction task. For instance, the *Time since last update* of a file has the lowest score across all target variables. On the other hand, *Fully Read Frequency* and *Request Offset* have the highest scores across all targets, indicating that they are helpful in predicting the target values.

The sets of features selected to predict the target variables have some differences, but the majority of the chosen features are shared between all sets. Based on the ranking scores and the correlation matrices, we selected all features related to the file and directory access pattern. From the File Format, we kept only the *File Format ID* and *File Count* because the patterns related to file formats either achieved a low score or had a high correlation with the selected directory pattern features. Features associated with the history of accesses are also not selected, except for the *Time since last access* and *Time since creation*. The final set of selected features has 24 features for the Next File Hotness, 27 for the Next Offset, 25 for the Next File Hotness Class, and 25 for the Next Offset Class.

To gain deeper insight into how each feature affects model prediction, we employed SHAP (SHapley Additive exPlanations) values [65] derived from tree-based machine-learning models, namely XGBoost models. XGBoost [66] is a gradient-boosting framework that builds an ensemble of trees and works well for both regression and classification problems. For each target variable and each trace, we trained an appropriate model (regressor or classifier), computed SHAP values for all features, and generated violin summary plots. In the case of classification, SHAP values were averaged over all classes to produce a single importance value per feature. Figure 1 displays the violin summary plots for the four targets for Trace 17, our longest production trace. These plots visualize the distribution of SHAP values for each of the top 20 features across a sample of 10,000 instances, capturing both the magnitude and variability of each feature's impact on the model's predictions. The spread on the X-axis indicates how much this feature changes predictions, the direction on the X-axis indicates whether the feature increases or decreases the prediction, while the vertical spread shows the variation of effect across the data points.

In the *Next File Hotness* regression setting visualized in Figure 1a, the (current) *File Hotness* is the most impactful feature, with positive SHAP values for higher feature values, indicating that hotter files typically increase the target variable. Features such as *File Fully Read Frequency* and *File Size* also exhibited strong influence, while other features have SHAP values generally concentrated near zero but showing asymmetry for certain cases. In contrast, the classification model shown in Figure 1b revealed a broader spread of SHAP values and highlighted additional influential features such as *Time Since Creation* and *Directory Access Frequency*, reflecting the model's effort to delineate class boundaries.

As shown in Figure 1c, the (current) *Request Offset* feature dominates the regression model for *Next Offset*, indicating a direct relationship to the predicted next offset, which is to be expected for sequential workloads. *File Length Pattern* and *File Size* also reveal a strong influence, while *Time Since Last Access* has an interesting bi-modal effect, according to which both high and low values can push predictions in either direction, suggesting interaction effects. On the other hand, the classification model shown in Figure 1d identifies features like *File Length Pattern Frequency* and *Directory Temporal Mild Ratio* as being more discriminative for distinguishing between *Next Offset* classes. The broader range and greater dispersion of SHAP values in the classification plot reflect more complex, nonlinear interactions.



(**a**) Next File Hotness Regression

(**b**) Next File Hotness Classification

(**c**) Next Offset Regression

(**d**) Next Offset Classification

Figure 1. SHAP summary plot (violin plot) showing the distribution of feature contributions to the model's predictions for the four target variables for Trace 17. Each violin represents a feature, with the width indicating the density of SHAP values across samples. Features are ordered by mean absolute SHAP value, reflecting their overall importance in the model.

5. Online Learning Framework

Modern data storage systems require supporting a high and continuous number of requests accessing an ever-growing number of files. These requisites demand a solution that can act online over the large stream of requests and be able to tackle the changing properties of data over time. Batch ML models can provide high-accuracy solutions, yet they require training and testing models offline. On the other hand, streaming ML can provide high-accuracy models by training models and making predictions online. The requirements supported by streaming ML models are (i) processing an instance at a time and only once, (ii) using a limited amount of time to process each instance, (iii) using a limited amount of memory, (iv) adapting to temporal changes, and (v) being ready to give a prediction at any time [20]. Overall, streaming ML models can naturally adjust and adapt

to workload changes over time with minimal overhead and significantly lower the need for massive historical datasets.

Driven by the aforementioned requirements, an online learning framework has been designed to collect information from a data storage system in real-time and use it to generate training instances online for continuously updating ML models. Figure 2 shows the conceptual architecture of the framework, which can be used along any multi-tiered file system or file-based storage system. The framework collects file requests in real-time along with corresponding file metadata from the file system and uses them to build and incrementally maintain a state, described in Section 5.1. The state and the provided information are then used to generate feature vectors and targets (i.e., training instances), which are then used to train and evaluate streaming ML models online (Section 5.2). Finally, the models can be used for making predictions while the system is running (Section 5.3). The online learning framework was developed with Java v17 and consists of around 4000 lines of code.

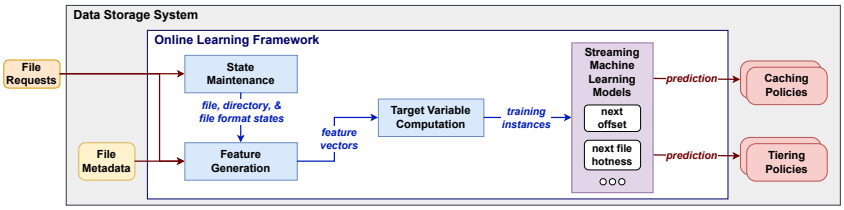


Figure 2. Conceptual architecture of the proposed online learning framework.

5.1. Online State Maintenance

The information required to translate a stream of file requests into a stream of feature vectors is tracked by a memory efficient time window comprised of *time slices*. Each time slice corresponds to a small contiguous time interval (e.g., 10 s) and holds aggregate information about the state of files, directories, and file formats accessed within that time interval. Consecutive time slices may have gaps between them, representing time periods when the storage system does not process any file requests. The time window only keeps time slices within the time length of the time window. Any time slice outside this time length is discarded. The time interval tracked by a time window and the time slices are configurable parameters. By default, the time slice holds information within 10 seconds, while the time window is set to 1 hour. Note that the time window is not implemented as a sliding window in order to avoid the large memory and computational overheads induced by maintaining a queue of the request data.

Within a single time slice, the information is aggregated at the level of files, directories, and file formats. For example, a time slice maintains, for each file, the number of times it was accessed, the time it was last accessed, how much data was read from this file, and, in general, any information necessary for computing the file features and access patterns discussed in Section 4.1. Similar information is also maintained for each directory and each file format accessed during a time slice. For optimization purposes, the time window also maintains the aggregated information currently stored across the time slices. Each time a time slice is created or updated, the aggregated information at the time window level is also updated. Similarly, when a time slice is discarded, the aggregated information is updated accordingly. Hence, the time window always maintains the most up-to-date information and is used when needed to extract features for generating a feature vector.

Overall, the time window and time slices offer several key advantages. First, they bound the amount of information stored regarding the files, directories, and file formats accessed only during the time window. Second, they enable the efficient computation of the aggregate states as time moves forward without the need to recompute any values. Finally,

old information is discarded from the system and does not influence future decisions; a property that can be very useful as access patterns change and evolve over time.

5.2. Online Training and Evaluation

Generating training instances online entails translating a stream of file requests into a stream of feature vectors and then appending target variables to them. Given a file request; the file metadata; and the current file, directory, and file format states, a feature vector can be generated (as explained in Section 4.1) containing the features required by the streaming ML models maintained by the system. The target variables are also calculated online and added to the feature vectors as target columns to generate training instances. The target generation process (and thus the training instances generation process) is different for each type of ML model maintained.

Calculating the *Next Offset* (and *Next Offset Class*) target requires using the offset of the current read request as the next offset of the previous read request of the same file, as shown in Algorithm 1. For this purpose, we maintain the feature vector computed from the latest read request per file (lines 4–7). When the file is accessed again, the current offset is set as the target of the previous feature vector corresponding to that file (lines 9–12). The offset is normalized using the file size so that the target value is between 0 and 1. In the classification case (not shown in Algorithm 1), if the current request offset equals the end offset of the previous request, then the target is set to '*sequential*'; otherwise, it is set to '*random*'. When a close request is received, the next offset is set equal to 1 to indicate that the file will not be read anymore (lines 13–15). In the classification case, the target is set to '*none*' when a close request is received.

Algorithm 1 Training Instance Generation for Next Offset

Input: fv – feature vector generated from current request

1: $fileID = fv$.getFileID()
2: $operation = fv$.getRequestOperation()
3: $previousFV =$ NULL
4: **if not** $previousFVmap$.contains($fileID$) **then**
5: **if** $operation =$ READ **then**
6: $previousFVmap$.put($fileID, fv$)
7: **end if**
8: **else**
9: $previousFV = previousFVmap$.get($fileID$)
10: **if** $operation =$ READ **then**
11: $previousFV$.setTarget(fv.getRequestOffsetNorm())
12: $previousFVmap$.put($fileID, fv$)
13: **else if** $operation =$ CLOSE **then**
14: $previousFV$.setTarget(1)
15: $previousFVmap$.remove($fileID$)
16: **end if**
17: **end if**
18: **return** $previousFV$

Calculating the *Next File Hotness* (and *Next File Hotness Class*) target requires computing and storing a set of feature vectors for a small period of time (which is equal, by default, to a time slice interval). Whenever a past feature vector v becomes older than this period of time, the current file hotness is computed using Equation (6) and set as the Next File Hotness target of v.

While a streaming ML model is trained online, it can also be evaluated online to assess its performance using the prequential evaluation [67,68] technique. In prequential evaluation, the processing of individual training instances follows their order of submission,

and each individual instance is used to test the model before training the model. This guarantees that the model is always tested on instances that it has never seen before and enforces the usage of all available data. The prediction accuracy of the model is continuously updated with each instance and changes during execution, but tends to increase and stabilize as more instances are processed. The current prediction accuracy is included along with each prediction so that the policies are informed about how accurate the predictions are at any given point in time.

5.3. Online Predictions

A streaming ML model is always available for providing a prediction given a feature vector. Generating a feature vector requires a file request, file metadata, and the current states, as discussed above. Hence, when a prediction is needed by a cache or tiering policy, the file system sends a file request and the latest file metadata to the framework, which are then used along with the latest states to generate a feature vector. The ML model is then probed with the generated feature vector and the prediction is returned to the policy.

6. Streaming Machine-Learning Models

The extracted features and target variables discussed in Section 4 are used to train ML models for predicting either the next file offset to be read or the hotness of a file in the near future. As multiple streaming ML algorithms are available, we first perform model selection to identify which algorithms to use for each model application (Section 6.1), followed by hyperparameter tuning for optimizing the prediction performance of the models (Section 6.2).

6.1. Model Selection

During model selection, the objective is to select the model with the best prediction accuracy to performance ratio. For the regression problems, we are looking for an algorithm that minimizes the Mean Absolute Error (MAE), a commonly used metric in regression, while being sufficiently fast. We selected a set of popular streaming regression algorithms available in the MOA framework [20] (v23.04.0) that are representative of different classes of regressors:

1. K-Nearest Neighbors (KNN) predicts the target value for a new data point by finding the k (e.g., 10) most similar (nearest) instances from the latest W (e.g., 1000) instances of the incoming data stream based on a distance metric (e.g., Euclidean distance).
2. Adaptive Random Forest (ARF) is an ensemble learning method that consists of multiple streaming decision trees (e.g., Hoeffding Trees), each trained on different subsets of the data stream.
3. Support Vector Regression (SVR) relies on a subset of the streaming training data, called support vectors, to incrementally define the decision function that minimizes the error within a predefined margin (epsilon).
4. Linear Regression with Stochastic Gradient Descent (LR-SGD) fits a linear relationship between input features and a target variable by minimizing the difference between predicted and actual values. The model parameters are updated incrementally using SGD.
5. Linear Regression with Adaptive Gradient Algorithm (LR-ADA) works the same as LR-SGD, but the model parameters are updated incrementally using ADA instead of SGD.

All of the models mentioned above are designed to adapt to concept drift, i.e., changes in the underlying data distribution over time, by incorporating a variety of mechanisms, such as using a fixed-size sliding window of recent data points (KNN), replacing a tree

when its performance degrades (ARF), adjusting the support vectors incrementally (SVR), and incorporating techniques like learning rate decay or forgetting mechanisms (LR-SGD, LR-ADA).

We executed these regression algorithms to predict the Next Offset and the Next File Hotness targets. Table 3 presents the MAE and execution time for Next Offset, while the results for Next File Hotness are very similar. Regarding prediction performance, KNN has the lowest MAE because it performs well in common, low-variance cases such as sequential workloads (which appear in many traces). On the other hand, ARF ranks second lowest for both targets. ARF handles outliers and complex relationships better, even with a slightly higher MAE. LR-SGD and LR-ADA have a slightly higher MAE, whereas SVR has the highest MAE and clearly fails to make good predictions. On the other hand, the execution time is much higher for KNN compared to the other algorithms, while ARF had the second highest execution time, revealing a clear tradeoff between prediction performance and execution time. Based on the above results, ARF was selected as the algorithm to use for the two regression problems. The decision was driven by two key factors. First, KNN has a significantly higher overhead (1.6–2.5×) compared to ARF. The second, and perhaps the most important reason, is related to the inner workings of KNN. Specifically, KNN computes the distance between the current sample and the available centroids, matching the sample to the nearest centroid within a time window. This drives KNN to always under-predict values for the Next Offset. For Trace 1, it under-predicts by about 16k, but for Trace 4 by typically 1–2 MB (because of larger files).

For classification, the Hoeffding Tree (HT) [20] was compared against the Adaptive Random Forest of Hoeffding Trees (ARF-HT). HT is a state-of-the-art, memory efficient decision tree designed for streaming data, capable of learning from large datasets. It exploits the fact that small samples can be enough to choose an optimal splitting attribute. It also has a unique feature: it guarantees performance as its output is asymptotically identical to non-incremental learning using an infinity of examples [20]. HT exhibited about the same predictive performance as ARF-HT but was an order of magnitude faster, and thus was selected as our classification algorithm.

Table 3. Mean Absolute Error and execution time of regression models predicting the Next Offset.

	Mean Absolute Error (MAE)					**Execution Time (CPU Time in Seconds)**				
Trace	**KNN**	**ARF**	**SVR**	**LR-SGD**	**LR-ADA**	**KNN**	**ARF**	**SVR**	**LR-SGD**	**LR-ADA**
1	0.0660	0.0899	1.3806	0.2850	0.1663	7.9667	6.4197	0.4471	0.4530	0.4273
2	0.0123	0.0248	2.8512	0.1550	0.0180	14.3740	7.0607	0.5886	0.5844	0.5566
3	0.1054	0.1822	0.5026	0.5265	0.1402	0.1263	0.3877	0.0675	0.0688	0.0673
4	0.0135	0.0378	10.5670	0.3912	0.1986	48.7526	30.5150	2.1178	2.1466	2.0745
5	0.0227	0.0475	4.8442	0.2129	0.0678	26.9410	14.5455	1.1651	1.1713	1.1596
6	0.0025	0.0083	26.8169	0.0195	0.0032	91.2398	43.4018	3.3994	3.0883	3.2911
7	0.0413	0.0637	15.7229	0.2027	0.1380	70.5418	39.5608	3.3572	3.0529	3.3041
8	0.0285	0.0574	20.8522	0.1423	0.0952	71.4809	48.9063	3.9817	4.0577	3.8280
9	0.0409	0.0725	33.4892	0.2198	0.1064	124.4128	53.3301	4.4851	4.7366	4.6960
10	0.0239	0.0457	26.3200	0.1370	0.0779	99.8426	53.7119	4.9598	4.4258	4.3624
11	0.0146	0.0297	25.1362	0.1132	0.0509	114.9402	60.9934	4.4784	5.1680	4.6234
12	0.0007	0.0048	282.3831	0.0128	0.0009	619.2878	259.6127	26.2367	25.7767	23.4651
13	0.2856	0.2772	0.6463	0.6503	0.3156	0.0520	0.2341	0.0271	0.0299	0.0304
14	0.0009	0.0038	2285.3300	0.0059	0.0021	4887.7723	1943.5129	160.7518	167.5465	170.0269
15	0.0007	0.0075	661.0170	0.0045	0.0020	1625.4702	575.0574	75.6567	69.2641	69.0651
16	0.0033	0.0102	3261.4318	0.0103	0.0041	6139.1183	1850.7011	228.6654	220.5844	223.7565
17	0.0495	0.1039	1777.0460	0.0462	0.0351	2496.2506	1360.6449	194.2261	218.9568	195.6011
Sum	0.6628	1.0668	8436.3371	3.1351	1.4222	16,438.5702	6348.5960	714.6112	731.1119	710.3354

6.2. Hyperparameter Tuning

The regression and classification algorithms are sensitive to their configuration parameters, impacting not only the accuracy but also the execution time and the size of the model. Thus, during hyperparameter tuning, the objective is to identify the best set of configurations that works for all traces, increases the accuracy, and decreases the execution time and size of the model. We performed a grid search to find the best set of parameters for the Adaptive Random Forest (ARF) and the Hoeffding Tree (HT) over the selected features.

Table 4 lists the three primary parameters that impact ARF performance, the tested parameter values, and the selected parameter values for each regression target. To define the search space, we leveraged the default values, literature, and domain knowledge [69]. We independently performed the grid search for each trace, finding (slightly) different sets of optimal hyperparameter settings per trace that minimize MAE. However, our objective is to find a single set of hyperparameter settings that performs well across all traces to ensure that the models will also work well with other traces. For this purpose, we first computed the minimum achievable MAE for each trace across all configurations. Next, we computed the difference of each MAE from the corresponding minimum MAE per trace. Finally, for each configuration, we computed the sum of the differences across all traces and ranked all configurations by this sum. The configuration with the smallest sum represents the configuration that predicted values closer to the original, with the smallest error across all traces.

The HT has six major parameters that impact performance, as listed in Table 5. We use the *F1 score* metric (i.e., the harmonic mean of precision and recall) to evaluate the classifications because it provides accurate results for both balanced and imbalanced datasets. The employed hyperparameter tuning methodology is analogous to the one used for regression. We first computed the maximum achievable F1 score for each trace across all configurations. Next, we computed the difference of each F1 score from the corresponding maximum F1 score per trace. Finally, for each configuration, we computed the sum of the differences across all traces and selected the configuration with the smallest sum.

Table 4. Tested and selected hyperparameter values for Adaptive Random Forest after hyperparameter tuning.

Parameter	Tested Values	Selected for Next Offset	Selected for Next File Hotness
Ensemble Size (num trees)	5, 10, 30, 40, 100	30	40
Features Per Tree (m)	10, 20, 30, 60	30	60
Features Mode (M=# features)	m, $M*(m/100)$, $\sqrt{M}+1$, $M-(\sqrt{M}+1)$	$M*(m/100)$	$M*(m/100)$

Table 5. Tested and selected hyperparameter values for Hoeffding Tree after hyperparameter tuning.

Parameter	Tested Values	Selected for Next Offset Class	Selected for Next File Hotness Class
Grace Period (instances between splits)	100, 200, 300, 1000	100	100
Split Criterion (used to split nodes)	Gini, Info Gain, SDR	Info Gain	Info Gain
Split Confidence (max error in splits)	0.05, 0.10, 0.15, 0.20	0.10	0.20
Tie Threshold (to break ties in splits)	0.05, 0.10, 0.15, 0.20	0.10	0.10
Leaf Prediction (method)	MC, NB, NBAdaptive	NBAdaptive	NBAdaptive
NB Threshold (to permit Naive Bayes)	0, 10, 100	0	0

7. Experimental Evaluation

This section presents the experimental evaluation of the proposed streaming ML approach. First, we discuss the performance and trade-offs of the four streaming ML models concerning their prediction accuracy, model size, and execution runtime in Section 7.1. Comparisons with batch-based ML/DL models and rule-based approaches are performed in Sections 7.2 and 7.3, respectively. Next, we present a more in-depth evaluation of the streaming ML models regarding scalability and adaptability (Section 7.4). We then repeat our experimental evaluation using an open-source workload from Google's Thesios project to show the generalizability of our proposed approach (Section 7.5). We finish the evaluation by discussing the system overheads (Section 7.6).

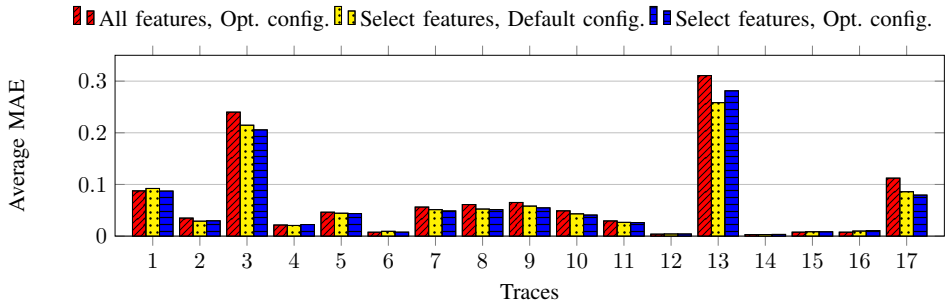
7.1. Streaming ML Model Evaluation

In this section, we compare the optimized configurations for the selected features against (i) the optimized configurations for all features and (ii) the default configurations for the selected features for each streaming ML model while using the prequential evaluation technique (recall Section 5.2). The experiments are run with DITIS [70], Huawei's distributed tiered storage simulator, on a machine with a 2.70 GHz CPU and 12 GB RAM. Each trace runs independently, and the evaluation of each trace starts with an empty model, which is trained live while the trace is running. This is one of the most significant advantages of the streaming ML paradigm, as we do not need to pre-train a model in advance.

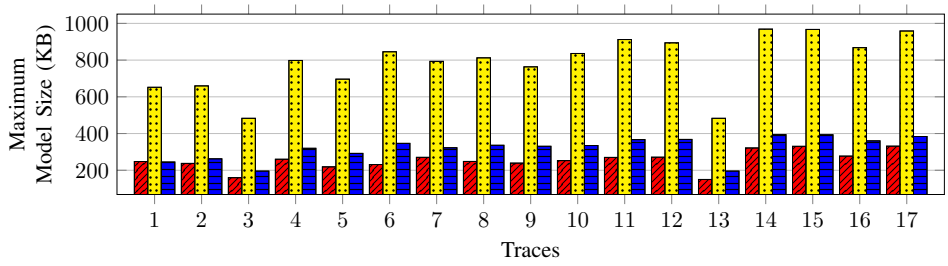
Figure 3a,b,c respectively present the average MAE (the lower, the better), the model size, and the runtime (for testing and training) for the *Next Offset* regression target. Note that Figure 3c employs an exponential scale on the y-axis to effectively represent the high variability in the runtime values across the different traces. Using the selected features with ARF improved MAE by 12.1% on average for almost all traces compared to using all features, further validating our feature selection process presented in Section 4. The average model size increased by 26.2% (around 67 KB) for the selected features because using better features leads to more frequent tree node splits and the creation of larger models. At the same time, the overall runtime decreased by 62.5% (i.e., 2.7× improvement). Compared to the default configuration for the selected features, the optimized configuration improved MAE in 11 out of the 17 traces, with a highlight to an improvement of 17.0% for Trace 6 and about 5.5% for Traces 1, 9, and 10. The same configuration, however, decreased the MAE of Trace 13 by 8.9%. Note that Trace 13 corresponds to a write-heavy workload, yielding only 45 training instances for predicting the next offset to be read. Hence, the model does not have enough data to learn any meaningful patterns. The optimized configuration did not present a significant improvement across the other traces, yielding a variation of about 1%. On the other hand, it decreased the maximum model size by 59.3% (i.e., 2.5× improvement) and the overall runtime by 74.8% (i.e., 4× improvement). The improvements in model size and runtime are observed in all traces. Hence, one of the benefits of the optimized configuration is to trade about 1% of the accuracy in some cases for faster training and predictions. This is a decision choice that needs to be evaluated when required.

Regarding the *Next Offset Class* target, the optimized against default configuration for the selected features improves the average accuracy by 1%, with a highlight to Trace 4; the F1 score improved from 0.91 with the default configuration to 0.98. The optimized configuration decreased the F1 score for only Traces 6 and 14 by negligible amounts of 0.1% and 0.02%, respectively. Note that the default configuration already produces a high F1 score, and achieving the improvement presented by the optimized configuration requires one to increase the model size by 4× and the runtime by 29%. However, the maximum model size with the optimized configuration is 282 KB (for Trace 14), and thus the model is

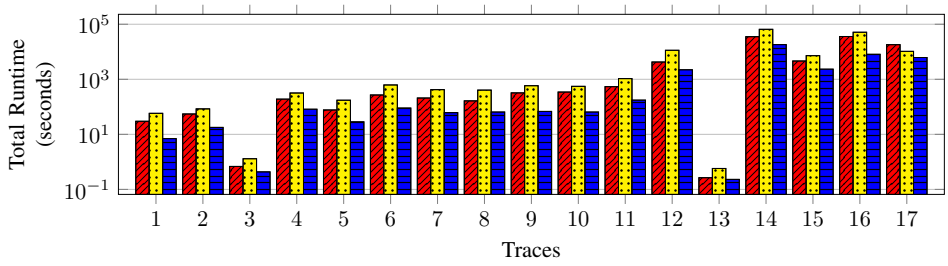
still very memory efficient. Hence, the optimized configuration improves the F1 score but increases the model size and runtime in a justifiable way.



(**a**) Mean Absolute Error (MAE)



(**b**) Maximum model size



(**c**) Runtime

Figure 3. The (**a**) Mean Absolute Error (MAE), (**b**) maximum model size, and (**c**) runtime when predicting the *Next Offset* with ARF using all features with optimized configuration, selected features with default configuration, and selected features with optimized configuration.

The results for the *Next File Hotness* and *Next File Hotness Class* follow the same trends as the next offset cases and are omitted due to space constraints. In general, we observe that both the Adaptive Random Forest and the Hoeffding Tree provide highly accurate models, which are very memory efficient, as reflected by the final size of each model, even after processing millions of requests.

Next, we drill down into the classification results for the two target variables. Figure 4 presents the confusion matrix for the Next Offset Class that shows how many predictions are correct and incorrect per class. As we can observe, the classification for the Next Offset correctly classifies most of the file requests as sequential requests, since many of the traces have large sequential reads. Even though there is an apparent imbalance problem with the training data, the model can accurately predict the class, even for the minority classes. This fact is also evident by the high Precision (0.995), Recall (0.996), and F1 score (0.996). Similarly, Figure 5 presents the confusion matrix for the Next File Hotness Class. About 63% of the file requests are classified in class *ONE* and 26% in class *TWO*. Hence, the imbalance problem is visible, but the model is still able to achieve a minimal number of miss-classifications, as well as high Precision (0.993), Recall (0.991), and F1 score (0.992).

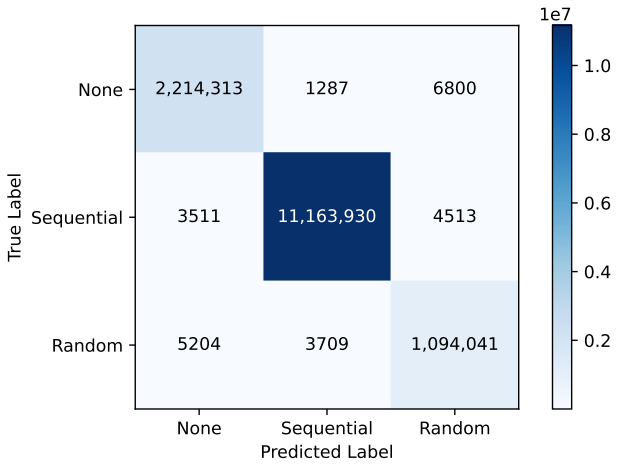


Figure 4. Confusion matrix for *Next Offset Class* target.

Further, we present two representative line graphs showing the original and predicted values for the two regression scenarios. In particular, Figure 6 presents the results for *Next Offset* for Trace 11, while Figure 7 presents the results for *Next File Hotness* for Trace 9. These figures visually demonstrate that, even in the presence of erratic and complex behavior, the predictions offered by the two models can faithfully follow the trends. At the beginning of the trace, there is a slightly higher difference between the original and the predicted values, which reduces as time goes by. This showcases the ability of the streaming ML models to learn and improve as they are trained with more instances. This is especially evident in the second half of Trace 11 in Figure 6, where the predicted values match almost exactly the original values. Trace 11 also serves as a good example of a concept drift in the access patterns of the workload. The first half of the trace is characterized by complex non-sequential access patterns, followed by a sudden switch to a sequential access pattern. The streaming ML model immediately adjusts and starts making correct sequential Next Offset predictions.

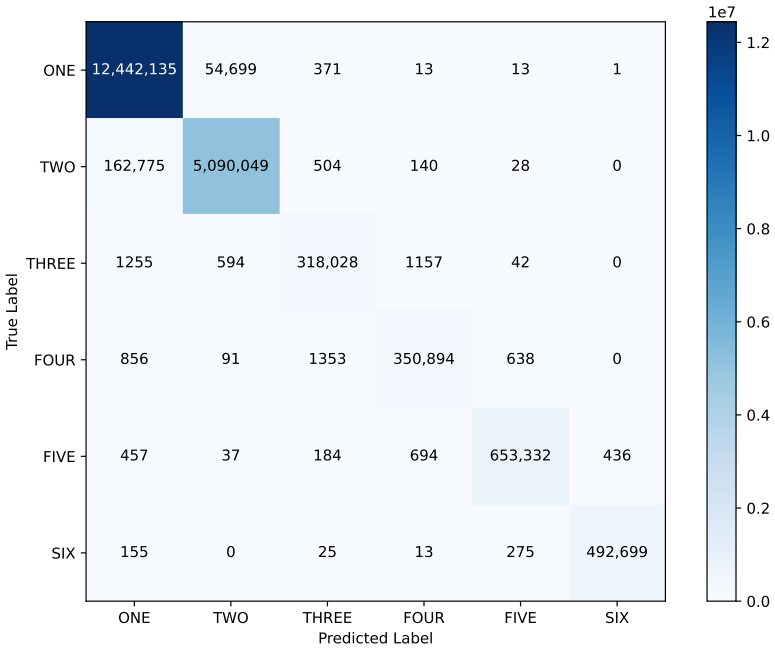


Figure 5. Confusion matrix for *Next File Hotness Class* target.

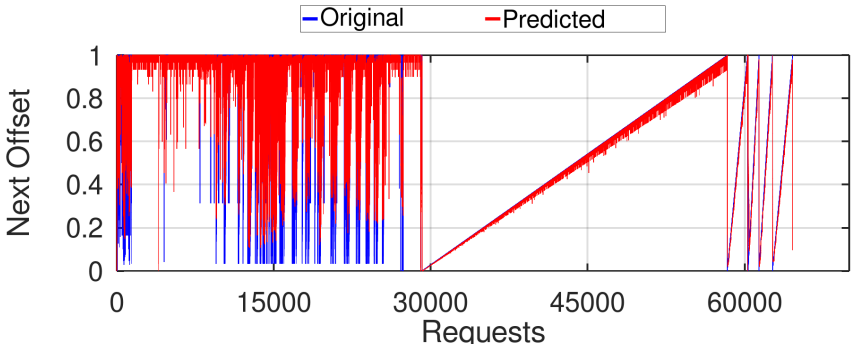


Figure 6. Original and predicted *Next Offset* values for Trace 11 with optimized configuration over selected features.

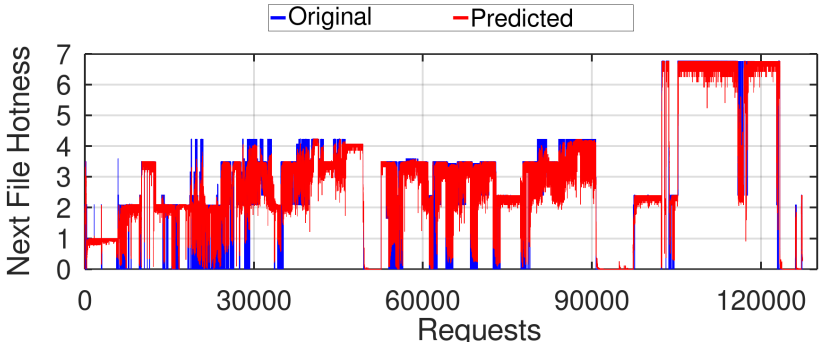


Figure 7. Original and predicted *Next File Hotness* values for Trace 9 with optimized configuration over selected features.

Key takeaways: The optimized configurations for the selected features for all four models produce highly accurate models (0.98 F1 score and 0.07 MAE on average), which are very memory efficient (<400 KB), even after processing millions of requests. The classification models work well, even in a highly imbalanced dataset. Over time, streaming ML models learn and improve as they are trained with more instances.

7.2. Streaming ML vs. Batch ML/DL

The objective of this section is to compare the proposed streaming ML approach against three common batch-based scenarios, representing how data engineers would train and update their batch models in production. The difference between the scenarios lies in how engineers perform the updates over time, where one would: (1st) train once but never update the model; (2nd) periodically retrain the model from scratch; and (3rd) retrain and incrementally update the model. To generate the scenarios, we split the data of each trace into ten equi-size segments (or *buckets*). For illustration, the buckets could represent different periods, e.g., ten consecutive hours. For the **1st scenario**, where the engineer would train once but never update the model, we train the batch model with the first bucket and test with the remaining buckets. For the **2nd scenario**, where the engineer would periodically retrain the model from scratch, we first train with the data of bucket #1 and test on the data of bucket #2. Then, we train a new model with the data of bucket #2 and test with the data of bucket #3, and so on. Thus, we train a model with bucket i and test with bucket $i+1$, without updating the model or aggregating the training data. For the **3rd scenario**, we train incrementally with the data from all the previous buckets and update the model. Thus, we first train a model with bucket #1 and test with bucket #2. Then, we retrain a model with the aggregated buckets #1 and #2 and test on bucket #3, and so on. Thus, we train a model with buckets #1 to i and test with bucket $i+1$. For the streaming scenario, prequential evaluation is used to test and train the model, with the

results averaged and reported per bucket for a fair comparison. The same training are used in all scenarios, containing the selected features described in Section 4.

We use Weka [71] (v3.8.6) to create batch ML models that correspond to our streaming models, namely *Random Forest Regressor* (*RFR*) for regression and *Decision Tree J48* (*DT*) for classification. These two batch models are used in prior related work that proposed ML-based cache policies [33], while the comparison between these streaming and batch-based algorithms has been performed in other domains as well [56,58]. Hyperparameter tuning was performed for all models, and the selected parameters are listed in Table 6. These experiments are run on a server with a 24-core 2.2 GHz CPU and 128 GB RAM. For each scenario, we collected the runtime and MAE/F1 score (for regression/classification) for every bucket used to test the model, leading to nine data points.

Table 6. Selected hyperparameter values for Decision Tree J48 (DT) and Random Forest Regressor (RFR) after hyperparameter tuning.

Decision Tree J48 (DT)	**Next Offset Class**	**Next File Hotness Class**
Pruning confidence	0.10	0.20
Num folds for reduced error pruning	3	3
Min number of instances per leaf	2	2
Random Forest Regressor (RFR)	**Next Offset**	**Next File Hotness**
Number of iterations	40	30
Size of each bag	100	100
Min variance for split	0.001	0.001
Min number of instances per leaf	1	1

Figure 8 presents the average F1 score and Figure 9 presents the total runtime (for testing and training) for all test buckets when predicting the Next Offset Class for the batch DT and streaming scenarios. The 1st scenario has the worst F1 score across most traces because the workload access patterns in many traces change drastically over time. For example, the 1st scenario achieves very low F1 scores for Trace 4. This trace begins with sequences of small random read operations and then switches to longer sequential read and write operations, driving the model to wrongly predict many sequential offsets as random. The 2nd and 3rd scenarios achieve much better F1 scores as they are (re) trained over time but at the expense of being, on average, 4× and 13× slower than the 1st scenario, respectively. The 3rd scenario exhibits a slightly better F1 score than the 2nd scenario in some cases but at the cost being 3.8× slower than the 2nd scenario due to the increased training data size for each subsequent bucket. In comparison, the streaming models exhibit almost the same high F1 scores as the 3rd scenario across the traces. Interestingly, the 3rd batch scenario achieves a higher F1 score for 7 out of the 17 traces (2.6% on average), while streaming achieves a higher F1 score for the other 10 traces (4.9% on average). Overall, streaming ML achieves a 1.8% higher F1 score while being, on average, 7.7× faster than the 3rd scenario, affirming its high performance in predicting the next offset efficiently.

The batch experiments were repeated by replacing the RFR and DT algorithms with deep neural network models from the Weka DeepLearning4j (v1.7.2) framework [72]. In particular, we developed *Long Short-Term Memory* (LSTM) *Recurrent Neural Network* (RNN) models, as proposed in [17,30], to make the same predictions. The RNN models were configured with 25 backward and 25 forward backpropagations through time, an LSTM layer with the Sigmoid gate activation function and the Rectified Linear Unit (ReLU) activation function, and an output layer with the softmax activation function and the multi-class, cross-entropy loss function (MCXENT) as an optimization objective. The Adam

optimizer, with a mean decay of 0.9 and a var decay of 0.999, was used as the updater, while Stochastic Gradient Descent (SGD), with a learning rate of 0.001, was used as the updater for the bias. Due to the excessive running times required for training the models (and the lack of a GPU), we present only the results from the first 12 traces. Note that lacking a GPU is a realistic testing environment since storage systems are typically not equipped with GPU devices and even lack significant CPU capabilities. Hence, even inference from deep-learning models can be prohibitively expensive in storage systems [47].

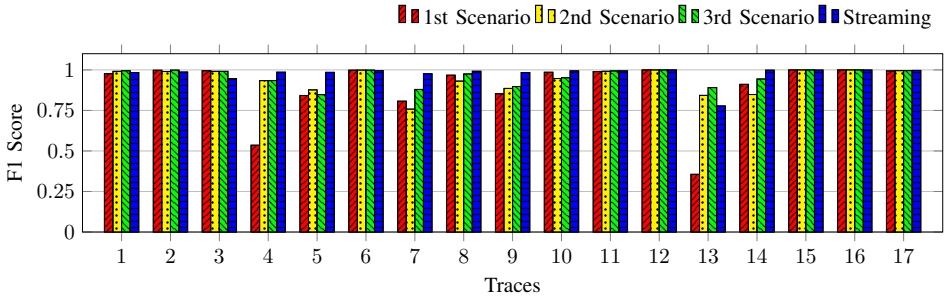


Figure 8. F1 score for the three batch scenarios using DTs against streaming ML for the *Next Offset Class*.

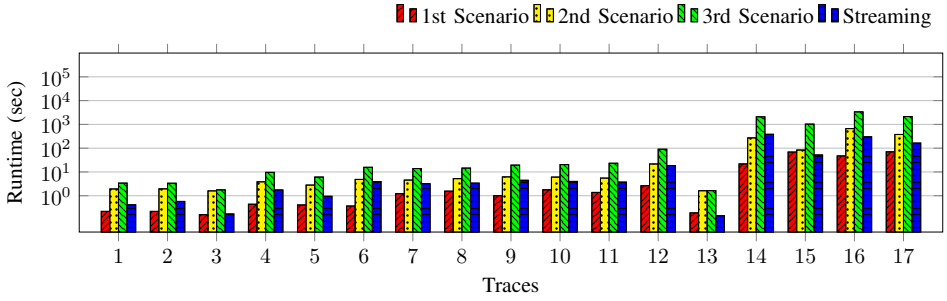


Figure 9. Runtime for the three batch scenarios using DTs against streaming ML for the *Next Offset Class*.

Figure 10 presents the average F1 score and Figure 11 the runtime when predicting the *Next Offset Class* for the batch RNN and streaming scenarios. The trends and results for the F1 score are very similar to the ones produced using DTs and discussed above: the 1st scenario can sometimes lead to very low prediction performance, while the 3rd scenario offers the best predictions for the batch-based experiments. In general, we did not observe any significant differences in F1 scores between DTs and RNNs in our experiments. In terms of runtime, training and using RNNs are 2–3 orders of magnitude slower than using DTs and, on average, 4000× slower than our streaming ML models, revealing that RNNs are impractically slow to be used within storage systems in production.

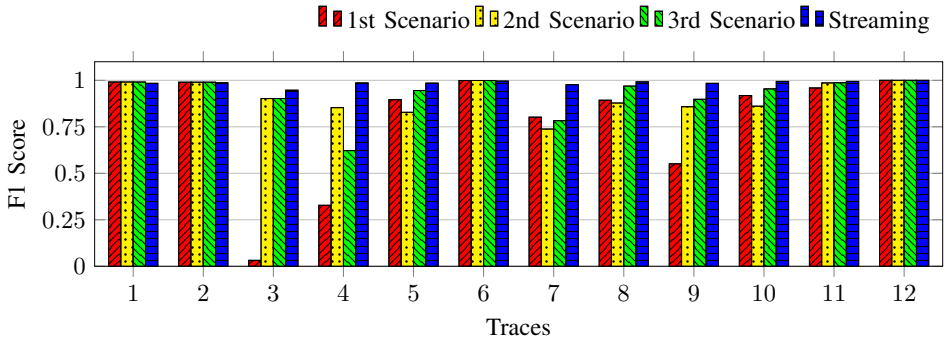


Figure 10. F1 score for the three batch scenarios using RNN (LSTM) against streaming ML for the *Next Offset Class*.

Figure 12 presents the variation of the F1 score along the buckets for Trace 8 when predicting the Next File Hotness Class. Note that each bucket corresponds to approximately 3.7 min as the duration of the entire trace is 37.2 min (recall Table 1). For the 1st scenario, we observe a high variation in the F1 score, with a sudden drop of 20% from bucket #2 to bucket #3 (from 0.97 to 0.77) for both DT and RNN models. Thus, training once but never updating the model may cause the model to underperform significantly. However, periodically updating the model or retraining it from scratch can lead the model to better performance, as observed in the 2nd and 3rd scenarios. This is due to the nature of the traces, where the workload access patterns of a bucket often have similarities to the ones of the previous bucket. The 3rd scenario has a better F1 score than the 2nd scenario because it builds upon previous knowledge. However, the streaming model always exhibits a higher and stable F1 score because it is continuously updated at every data sample, while the model from the 3rd scenario is only updated in batches. Hence, even a sudden change in the workload patterns (as observed clearly in bucket #7 where all batch models experience a sudden drop in prediction performance) can be learned quickly in a streaming environment (with a minor drop of 1% in bucket #7).

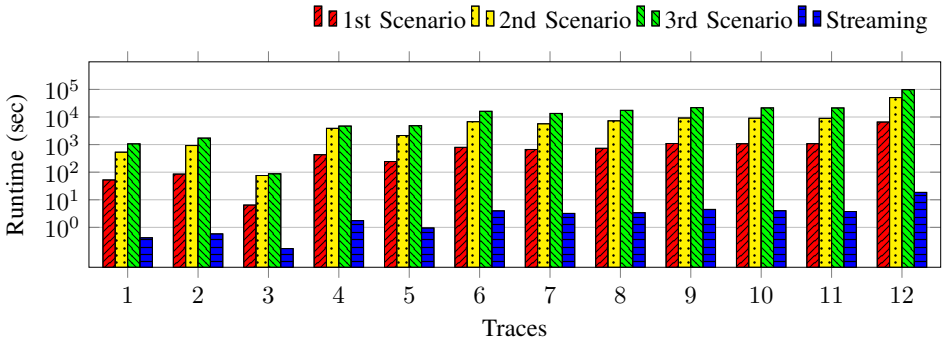


Figure 11. Runtime for the three batch scenarios using RNN (LSTM) against streaming ML for the *Next Offset Class*.

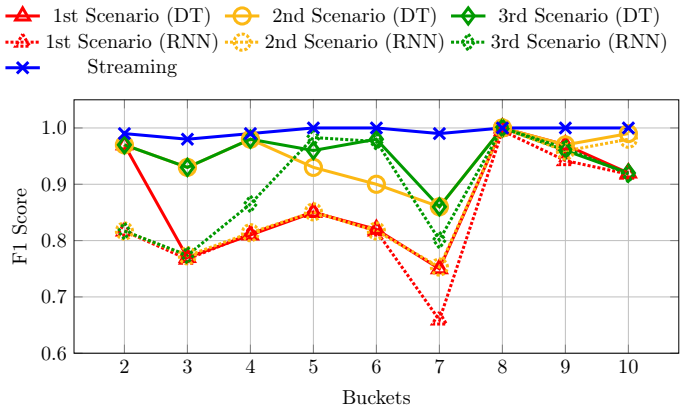


Figure 12. F1 score per test bucket for the three batch scenarios using DTs and RNNs as well as streaming ML for Trace 8 for the *Next File Hotness Class*.

Key takeaways: Periodically retraining a batch model (either from scratch or incrementally) yields higher overall predictive performance than using a fixed batch model, reducing MAE by 0.12 and increasing F1 score by 6% on average for *Next Offset*. Streaming ML further reduces MAE by 0.03 and increases F1 score by 1.8% compared to retraining the batch model incrementally (3rd scenario). At the same time, streaming ML significantly reduces retraining costs compared to the batch-based 3rd scenario, as it is an order of magnitude faster than DTs and three orders of magnitude faster than RNNs.

7.3. Streaming ML vs. Rule-Based Predictions

This section compares the proposed streaming ML methodology against non-ML, rule-based approaches that have been used in the past. In particular, for predicting the *Next Offset*, we implemented a sequential prefetching policy used in commercial storage systems, which detects the presence (or not) of the sequential access pattern for a file and makes the next offset prediction accordingly [73]. For the *Next File Hotness*, we implemented a linear interpolation approach using the most recent file hotness measurements for predicting future file hotness [11].

Figures 13 and 14 compare the F1 score achieved by the rule-based approaches against streaming ML for predicting the *Next Offset* and *Next File Hotness*, respectively. For the traces that represent predominantly sequential read workloads, such as Traces 1 and 2, the rule-based approach can make very good predictions and achieve over 90% F1 score. However, there are also several traces with more complex access patterns, for which the rule-based approach performs very poorly, with an F1 score as low as 39%. The average F1 score across all traces is 87% for the rule-based approach as opposed to 98% for the streaming ML approach. For the next file hotness case, the rule-based approach achieves an average F1 score of 70% across all traces, revealing that it is hard to predict the future file hotness based on trends from previous hotness values. On the other hand, streaming ML achieves a consistent performance, with an average F1 score of 97%.

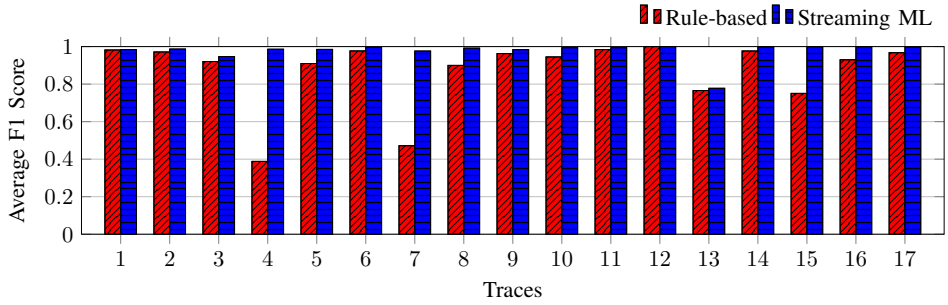


Figure 13. F1 score for rule-based approach against streaming ML for *Next Offset Class*.

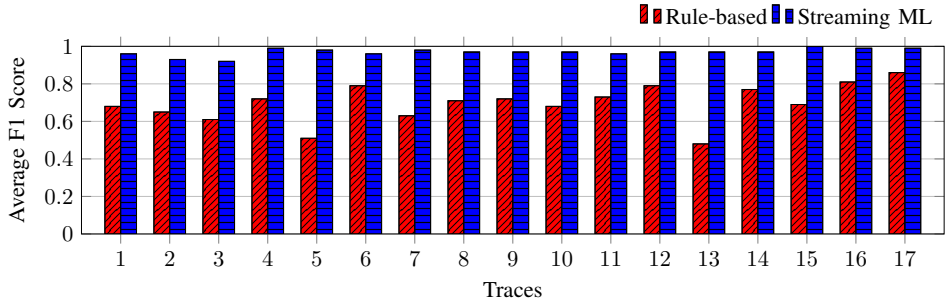


Figure 14. F1 score for rule-based approach against streaming ML for *Next File Hotness Class*.

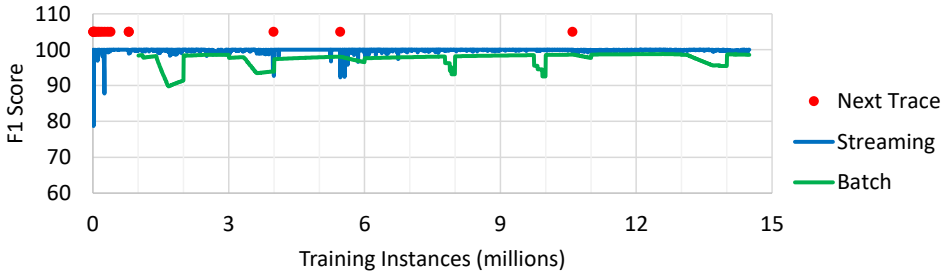
Key takeaways: The streaming ML approach consistently exhibits the highest F1 score for all traces for both targets (97% on average), as well as very robust performance, irrespective of the complexity of the workload.

7.4. Streaming ML Scalability and Adaptability

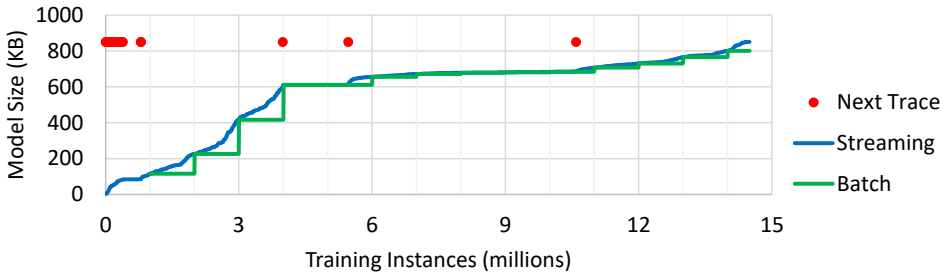
This section aims to provide an in-depth evaluation of the proposed streaming ML models regarding scalability and adaptability. For this purpose, we concatenated the 17 traces into one extensive trace containing 27.2 million file requests (testing scalability) and sudden changes to the workload access patterns at the trace boundaries (testing adaptability). The file requests are submitted by almost 50 k different applications with

varying intensity, with the underlying storage system yielding up to 20.3 k IOPS and 3.5 GB/s throughput. We extracted the selected features described in Section 4 from the trace and generated 14.5 million training instances for the Next Offset case and 9.1 million for the Next File Hotness case. We used prequential evaluation to evaluate the streaming ML models in terms of F1 score and MAE (for classification and regression, respectively), model size, and execution runtime. For comparison purposes, we also evaluated the corresponding batch-based ML models using the 3rd scenario (as it yielded the highest predictive performance among the batch scenarios; recall Section 7.2), where we retrain the model every one million training instances using all the instances up to that point.

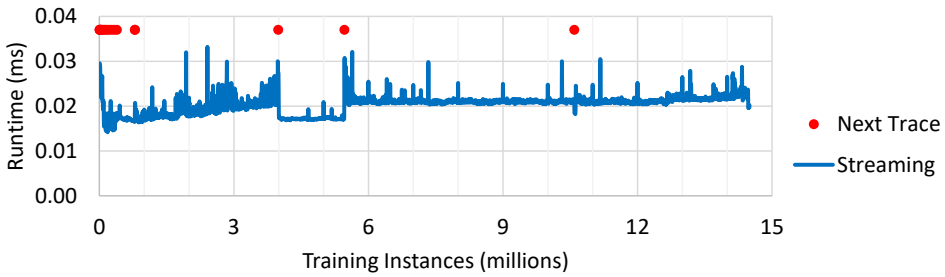
Figure 15 shows the F1 score, model size, and runtime (for testing and training per instance) for the Next Offset Class case. Regarding the F1 score, we observe that the streaming ML model is able to quickly achieve and maintain a high score of around 99.5%, with two notable exceptions: near 4 million instances (when Trace 15 begins) and near 5.4 million instances (when Trace 16 begins). As these traces have some different access patterns not seen previously, the performance of the model drops to 93% but recovers quickly (as seen in Figure 15a), highlighting the strong adaptability of the streaming ML model. On the other hand, the batch model experiences multiple performance drops down to 90–92% F1 score for longer durations and recovers after retraining the model every one million instances to achieve an F1 score of up to 98%.



(**a**) F1 Score



(**b**) Model Size

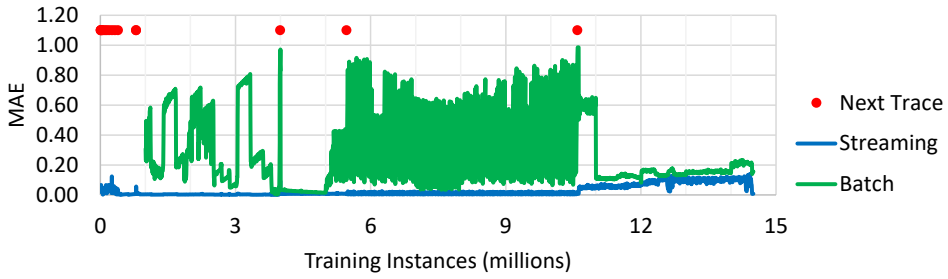


(**c**) Runtime

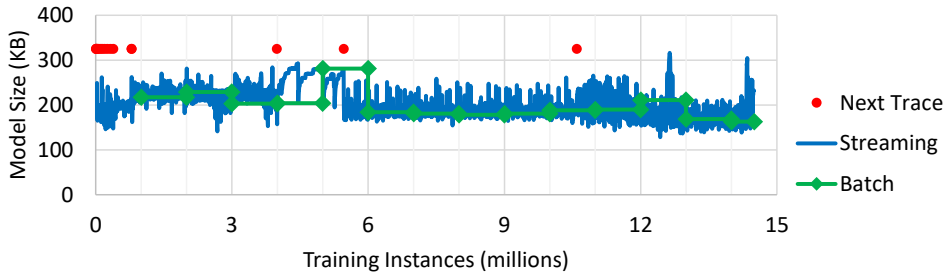
Figure 15. The (**a**) F1 score, (**b**) model size, and (**c**) runtime for the *Next Offset Class* case using all the training instances from the 17 traces consecutively. The red points show when a trace ends and the next one begins.

The size of the ML streaming model shown in Figure 15b increases linearly to 600 KB during the first 4 million training instances and sublinearly thereafter as the model grows more slowly to 800 KB. This observation applies to the batch ML model as well. Similarly, we observe some variations in the runtime in Figure 15c that are more intense at the beginning of the trace or near the trace boundaries due to the more active training and growth of the model. Afterward, the runtime stabilizes to around 0.021 ms per record. However, what is not shown in these graphs are the additional overheads of the batch-based approach: (1) 350 MB of memory is required for storing one million records between each retraining and (2) the first training takes 16.8 s but the time increases linearly and reaches 231 s for training with 14 million instances. As the streaming ML model trains whenever a new training instance is generated, there is no need for storing the instances or retraining the model periodically; hence, it is more scalable than the batch-based approach.

Figure 16 shows the MAE, model size, and runtime for the Next Offset regression case. Unlike the classification case, the batch-based regression model exhibits a very unstable predictive performance characterized by high oscillations of MAE between 0.02 and 0.93 (recall the target is normalized in the range 0 to 1), as seen in Figure 16a. Occasionally, the MAE reduces sharply after some retraining sessions, but shortly after the oscillations resume. Across the entire trace, the average MAE for the batch-based model is 0.29. On the contrary, the streaming ML model, with its per-record retraining process, achieves a low MAE consistently, with an average value of 0.027, i.e., an order of magnitude improvement, showcasing its great adaptability. The observations regarding model size (Figure 16b) and runtime (Figure 16c) are the same as with the classification case discussed above.

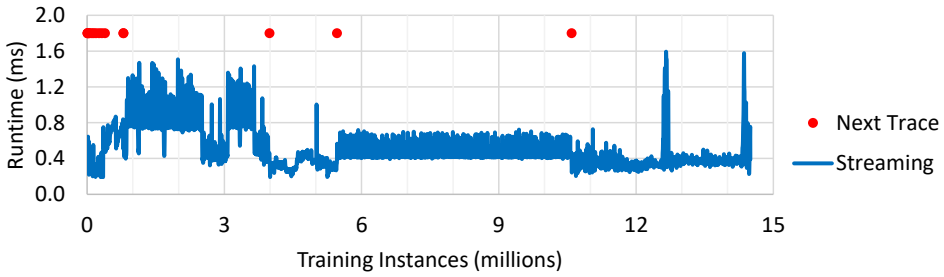


(**a**) Mean Absolute Error (MAE)



(**b**) Model Size

Figure 16. *Cont.*



(**c**) Runtime

Figure 16. The (**a**) Mean Absolute Error (MAE), (**b**) model size, and (**c**) runtime for the *Next Offset* case using all the training instances from the 17 traces consecutively. The red points show when a trace ends and the next one begins.

Key takeaways: Streaming ML demonstrates robust and stable model performance despite sudden shifts in workload access patterns (moving from one trace to another). On the other hand, batch-based approaches occasionally suffer from significant performance drops (e.g., up to 10% F1 score for *Next Offset Class*) or erratic behavior (e.g., MAE oscillations between 0.03 and 0.92 for *Next Offset*), even when retraining incrementally (3rd scenario). The streaming ML model size and runtime remain bounded and low, even after processing millions of training instances.

7.5. Evaluation with Other Workloads

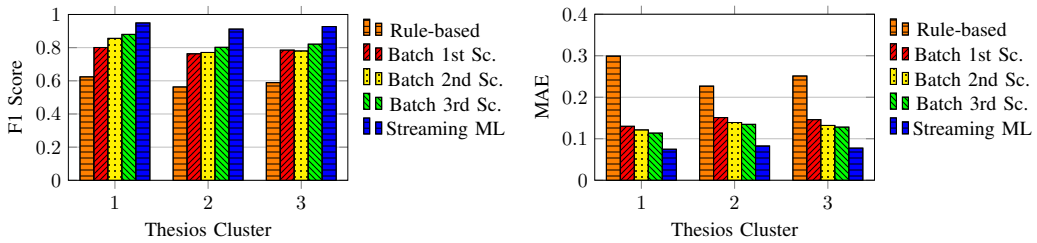
To demonstrate the generalizability of our proposed solution, we repeated our experimental evaluation with open-source workload traces provided by Google's Thesios project [74] and available in the SNIA IOTTA repository [75]. The Thesios dataset contains I/O traces from January to March 2024 from three clusters of storage servers in Google's distributed storage system with different types of workload traffic. Table 7 shows some key average statistics per trace per cluster in the Thesios dataset, revealing that each trace contains tens of thousands of read I/O requests, processing 20–42 GB of data.

Table 7. Average statistics per trace per cluster in the Thesios dataset.

Statistic	Cluster 1	Cluster 2	Cluster 3
# Applications	95	193	278
# Read Requests	60,098	42,513	54,542
Data Read (GB)	25	37	30
# Write Requests	85,923	47,791	110,099
Data Written (GB)	16	16	18

For these experiments, we evaluated the four streaming ML models against the rule-based approaches (recall Section 7.3) as well as the three scenarios of the batch-based ML approaches with bucket size 5000 (recall Section 7.2). Figure 17a shows the F1 score for *Next Offset Class* for the three clusters from the Thesios dataset. The rule-based approach yields the lowest performance, with only a 59% F1 score on average across the clusters due to the complex non-sequential access patterns present in the Thesios workload. The batch-based ML approaches achieve better performances of around 81% F1 score, with the 3rd scenario (incremental retraining) offering marginal benefits (of around 3%) over the other two scenarios. Finally, the streaming ML approach consistently achieves the highest F1 score of up to 93% across all traces and clusters, showcasing the strong predictive power and adaptability of the streaming models. The results are very similar for the regression models, as illustrated in Figure 17b, which shows MAE for the *Next File Hotness* case for the Thesios dataset. Once again, the rule-based method leads to the lowest performance

(i.e., highest MAE), followed by the three batch-based ML approaches, while streaming ML achieves the best performance.



(**d**) F1 score (**e**) MAE

Figure 17. (**a**) F1 score for *Next Offset Class* and (**b**) MAE for *Next File Hotness* for streaming ML against rule-based and batch approaches for the Thesios dataset.

Figure 18 shows a more in-depth view of how the F1 score changes over time for *Next Offset Class* for a representative trace from Cluster 1 of the Thesios dataset. For most of the trace, the rule-based approach shows a steady but low performance of around 62% F1 score because it can only detect specific sequential access patterns. The 1st batch scenario (trained once) performs well at the beginning (87% F1 score) for some time, but its performance gradually declines to 66% F1 score because new access patterns appear in the trace that the model had not seen before. Periodically retraining the model (2nd scenario) reverses the declining trend and manages to achieve up to 83% F1 score toward the end of the trace. Incrementally retraining the model (3rd scenario) yields steady performance with 86% F1 score but at the expense of collecting training data and performing costly retraining cycles. Once again, streaming ML yields the most robust and highest performance of 92% F1 score throughout most of the trace, even when new access patterns appear over time.

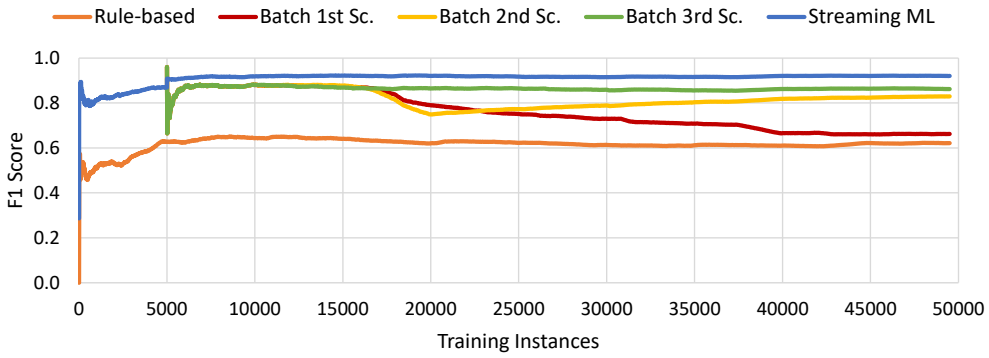


Figure 18. F1 score for streaming ML against rule-based and batch approaches for *Next Offset Class* for a representative trace from Cluster 1 of the Thesios dataset.

The results for the runtime and memory comparison of the approaches are very similar to those presented in the previous sections using the Huawei production traces and, hence, are omitted to avoid repetition.

Key takeaways: The results when using the Thesios dataset fully replicate the results observed when using the Huawei dataset, showing that the benefits of our proposed streaming ML approach generalize to other workloads from diverse storage systems.

7.6. System Overheads

Finally, we investigate the CPU and memory overheads introduced from training and using streaming ML models while the storage system is running. The overheads include both the data and processing needed to maintain the necessary state, generate feature

vectors and target variables to create training instances, train the model online, and use the model to make predictions. The training cost for a regression model for *Next Offset* is, on average, 1.8 ms per training instance, whereas for a classification model it is only 0.025 ms. The prediction cost is much lower, at 0.05 ms for regression and 0.005 ms for classification. The memory overhead is very low and less than 1.3 MB for both training and storing the models.

The training cost for a regression model for *Next File Hotness* is, on average, 1.7 ms per training instance, whereas for a classification model it is only 0.3 ms. The prediction cost is much lower, at 0.026 ms for regression and 0.004 ms for classification. The memory overheads for training are higher, with a maximum of up to 116 MB, because multiple feature vectors are created and kept in memory for some time until a target file hotness can be computed. However, the models are very memory efficient and up to 302 KB and 58 KB in size for regression and classification, respectively, for each trace.

The total computational cost for training a classification and a regression model constitutes only 0.32% and 2.27%, respectively, of the total time the file system spends serving read I/O requests. As the online learning framework proposed in Section 5 operates asynchronously within the data storage system and the CPU overhead is so low, the performance of the user-issued file requests is not affected. In addition, the computation cost for using a classification and a regression model constitutes, respectively, only 0.02% and 0.28% of the time spent serving read I/O requests, which is minuscule compared to the potential benefits these models can provide to the underlying storage system.

Key takeaways: The streaming ML approach is very practical, with low CPU and memory overheads for both training and utilizing the models while the storage system is running.

8. Discussion and Future Work

The experimental evaluation in Section 7 has demonstrated the high efficiency, high predictive performance, and high adaptability of the proposed streaming ML approach. Even though training times are low, a large storage system that processes thousands of requests per second could generate more training instances per second than the model can process. To address this issue, dynamic sampling can be used to ensure that the amount of training instances generated by the system is bounded by the maximum number of training instances the model can process. Further investigation is required to study how sampling will impact the prediction accuracy of the streaming ML models.

Another limitation of the proposed approach is that streaming ML algorithms, like most batch ML algorithms, are centralized. In a distributed system, the training instances must be transferred to a central location for training the model. Fortunately, the training instances are small (around 360 bytes each) so the network overhead will be small, even when transferring hundreds of training instances per second. However, a centralized model would also introduce a latency overhead for inference. This issue could be addressed by utilizing a federated learning approach, where each node could independently train a local streaming ML model that is periodically merged with all other models to converge towards a global model. We leave the investigation of federated learning for future work.

Another future work involves developing new caching and tiering policies that can leverage the models' predictions to make more informed decisions. The prediction of the next offset can provide prefetching policies with extra information for deciding when and which file fragments or blocks to prefetch. As shown in the experimental evaluation, the classification model can accurately detect the sequential access pattern, which can be used by a policy to prefetch the file fragments to be accessed next. Even though the classification model can also accurately detect the random sequential access pattern, this information

alone is not helpful to a prefetching policy. For this purpose, the regression model can be used for prefetching targeted data into the cache. As the regression model is not as robust as the classification one (it has a high MAE for some traces), it will be interesting to implement a hybrid prefetching policy that takes advantage of both models for making prefetching decisions.

Future cache management and tiering policies can leverage the value (or class) of the next file hotness prediction and decide to prioritize a particular file over another during caching or data migrations across storage tiers. Both the regression and the classification models performed well, but we hypothesize that the extra granularity provided by the regression model will not significantly improve the performance of the future developed policies compared to using the 6-class classification model. If this is shown to be correct, the classification model will be preferred as it is more computationally efficient than the corresponding regression model.

While the proposed methodology focuses on multi-tiered data storage systems, it is relevant and potentially applicable to other emerging technologies, such as the Cloud–Edge–IoT continuum, which integrates cloud computing, edge computing, and the Internet of Things (IoT) [76]. By intelligently caching and managing IoT data locally at the edge nodes in real-time, edge computing minimizes the need for repeated data retrievals from central cloud storage. In addition, hybrid cloud architectures are increasingly being adopted to balance cost, performance, and security for enterprise systems [77]. Such environments often employ multilevel caching or storage tiering to optimize data access speeds and reduce latency. This can be achieved by managing dynamic data movements between fast, local storage and slower, centralized storage based on real-time access patterns. Finally, emerging storage technologies such as persistent memory further enhance the capabilities of cloud storage systems and provide new storage layers that can be used for high-speed caching and tiering with new characteristics [78].

9. Conclusions

In this paper, we employ streaming ML algorithms for modeling workload patterns in data storage systems. In particular, we (i) analyzed the access patterns present in several production workloads, (ii) extracted a set of strong features from these workloads, and (iii) built a set of ML models that can help data management policies to make more intelligent data-driven decisions by providing relevant predictions.

During the workload analysis, we identified several file access patterns related to the temporal, spatial, length, and frequency aspects of the files accessed in the storage system. The extracted information is related to either a file, a directory, or even a file format. We observed that directories and file formats can provide extra context to files because they hold information about the peer files stored in the same directory or with the same file format. These patterns were found to be relevant and important features during the feature analysis.

In order to support future caching and tiering policies, we explored the problems of predicting the next offset and future file hotness as both a regression and a classification problem, resulting in four different models. We modeled each of these targets using streaming ML, benefiting from the advantages of the streaming paradigm. The resulting models have a small memory footprint, are computationally efficient, and achieve high prediction accuracy for both regression and classification, even in the presence of complex workload pattern shifts.

Author Contributions: Conceptualization, H.H. and L.Y.; methodology, E.R.L.F., K.F. and H.H.; software, E.R.L.F. and G.S.; validation, E.R.L.F., L.Y. and J.S.; investigation, E.R.L.F., G.S. and L.Y.; resources, L.Y., K.F. and J.S.; writing—original draft preparation, E.R.L.F. and H.H.; writing—review

and editing, E.R.L.F., G.S., L.Y., K.F., J.S. and H.H.; visualization, E.R.L.F. and G.S.; supervision, K.F. and H.H.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available in the SNIA IOTTA Repository at http://iotta.snia.org/ (accessed on 1 March 2025).

Conflicts of Interest: Lun Yang, Kebo Fu, and Jianqiang Shen were employed by the company Huawei Technologies. The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ARF	Adaptive Random Forest
DT	Decision Tree
HDD	Hard Disk Drive
HT	Hoeffding Tree
KNN	K-Nearest Neighbors
LR-ADA	Linear Regression with Adaptive Gradient Algorithm
LR-SGD	Linear Regression with Stochastic Gradient Descent
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
NVRAM	Non-Volatile Memory
ReLU	Rectified Linear Unit
RFR	Random Forest Regressor
RNN	Recurrent Neural Network
SSD	Sold State Drive
SVR	Support Vector Regression

References

1. Niu, J.; Xu, J.; Xie, L. Hybrid Storage Systems: A Survey of Architectures and Algorithms. *IEEE Access* **2018**, *6*, 13385–13406. [CrossRef]
2. Bang, J.; Kim, C.; Wu, K.; Sim, A.; Byna, S.; Sung, H.; Eom, H. An In-Depth I/O Pattern Analysis in HPC Systems. In Proceedings of the 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), Bengaluru, India, 17–18 December 2021; pp. 400–405. [CrossRef]
3. Paul, A.K.; Karimi, A.M.; Wang, F. Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems. In Proceedings of the IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS, Virtual Event, 3–5 November 2021.
4. Ren, Z.; Shi, W.; Wan, J.; Cao, F.; Lin, J. Realistic and Scalable Benchmarking Cloud File Systems: Practices and Lessons from AliCloud. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 3272–3285. [CrossRef]
5. Costa, L.B.; Yang, H.; Vairavanathan, E.; Barros, A.; Maheshwari, K.; Fedak, G.; Katz, D.; Wilde, M.; Ripeanu, M.; Al-Kiswany, S. The Case for Workflow-Aware Storage:An Opportunity Study. *J. Grid Comput.* **2015**, *13*, 95–113. [CrossRef]
6. Shibata, T.; Choi, S.; Taura, K. File-Access Patterns of Data-Intensive Workflow Applications and Their Implications to Distributed Filesystems. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC), Chicago, IL, USA, 21–25 June 2010; pp. 746–755.
7. Sun, H.; Dai, S.; Huang, J.; Yue, Y.; Qin, X. DAC: A dynamic active and collaborative cache management scheme for solid state disks. *J. Syst. Archit.* **2023**, *140*, 102896. [CrossRef]
8. Lin, H.; Li, J.; Sha, Z.; Cai, Z.; Shi, Y.; Gerofi, B.; Liao, J. Adaptive Management With Request Granularity for DRAM Cache Inside NAND-Based SSDs. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2023**, *42*, 2475–2487. [CrossRef]
9. Wang, H.; Yi, X.; Huang, P.; Cheng, B.; Zhou, K. Efficient SSD Caching by Avoiding Unnecessary Writes Using Machine Learning. In Proceedings of the 47th International Conference on Parallel Processing (ICPP), Eugene, OR, USA, 13–16 August 2018.
10. Eisenman, A.; Cidon, A.; Pergament, E.; Haimovich, O.; Stutsman, R.; Alizadeh, M.; Katti, S. Flashield: A Hybrid Key-Value Cache That Controls Flash Write Amplification. In Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, 26–28 February 2019; pp. 65–78.

11. Herodotou, H.; Kakoulli, E. Automating distributed tiered storage management in cluster computing. *Proc. VLDB Endow.* **2019**, *13*, 43–56. [CrossRef]
12. Chakraborttii, C.; Litz, H. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *Proceedings of the Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*; Springer International Publishing: New York City, NY, USA, 2021; pp. 427–443.
13. Yang, D.; Berger, D.S.; Li, K.; Lloyd, W. A Learned Cache Eviction Framework with Minimal Overhead. *arXiv* **2023**, arXiv:2301.11886.
14. Sun, H.; Cui, Q.; Huang, J.; Qin, X. NCache: A Machine-Learning Cache Management Scheme for Computational SSDs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2023**, *42*, 1810–1823. [CrossRef]
15. Wong, D.L.K.; Wu, H.; Molder, C.; Gunasekar, S.; Lu, J.; Khandkar, S.; Sharma, A.; Berger, D.S.; Beckmann, N.; Ganger, G.R. Baleen: ML Admission & Prefetching for Flash Caches. In Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST 24), Santa Clara, CA, USA, 26–29 February 2024; pp. 347–371.
16. Narayanan, A.; Verma, S.; Ramadan, E.; Babaie, P.; Zhang, Z.L. DeepCache: A Deep Learning Based Framework For Content Caching. In Proceedings of the 2018 Workshop on Network Meets AI & ML, Budapest, Hungary, 20–25 August 2018; pp. 48–53.
17. Shi, Z.; Huang, X.; Jain, A.; Lin, C. Applying Deep Learning to the Cache Replacement Problem. In Proceedings of the Annual International Symposium on Microarchitecture, Columbus, OH, USA, 12–16 October 2019; pp. 413–425.
18. Ganfure, G.O.; Wu, C.F.; Chang, Y.H.; Shih, W.K. DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 3311–3322. [CrossRef]
19. Marpu, R.; Manjula, B. Streaming machine learning algorithms with streaming big data systems. *Braz. J. Dev.* **2024**, *10*, 322–339. [CrossRef]
20. Bifet, A.; Holmes, G.; Kirkby, R.; Pfahringer, B. MOA: Massive Online Analysis. *J. Mach. Learn. Res.* **2010**, *11*, 1601–1604.
21. Lucas Filho, E.R.; Yang, L.; Fu, K.; Herodotou, H. Streaming Machine Learning for Supporting Data Prefetching in Modern Data Storage Systems. In Proceedings of the First Workshop on AI for Systems (AI4Sys '23), Orlando, FL, USA, 20 June 2023; pp. 7–12.
22. Pham, V.N.; Josh, M.L.; Le, D.T.; Lee, S.W.; Choo, H. A Prediction-Based Cache Replacement Policy for Flash Storage. In *Proceedings of the Future Data and Security Engineering. Big Data, Security and Privacy, Smart City and Industry 4.0 Applications*; Dang, T.K., Küng, J., Chung, T.M., Takizawa, M., Eds.; Springer: Singapore, 2021; pp. 161–169.
23. Sun, H.; Dai, S.; Cui, Q.; Huang, J. LCache: Machine Learning-Enabled Cache Management in Near-Data Processing-Based Solid-State Disks. In *Proceedings of the Network and Parallel Computing*; Springer International Publishing: New York City, NY, USA, 2021; pp. 128–139.
24. Vietri, G.; Rodriguez, L.V.; Martinez, W.A.; Lyons, S.; Liu, J.; Rangaswami, R.; Zhao, M.; Narasimhan, G. Driving cache replacement with ML-based LeCaR. In Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Co-Located with USENIX ATC 2018, USENIX, Boston, MA, USA, 11–13 July 2018.
25. Sethumurugan, S.; Yin, J.; Sartori, J. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Virtual Event, 27 February–3 March 2021; pp. 291–303.
26. Laga, A.; Boukhobza, J.; Koskas, M.; Singhoff, F. Lynx: A learning linux prefetching mechanism for SSD performance model. In Proceedings of the 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Daegu, Republic of Korea, 17–19 August 2016; pp. 1–6.
27. Xu, R.; Jin, X.; Tao, L.; Guo, S.; Xiang, Z.; Tian, T. An efficient resource-optimized learning prefetcher for solid state drives. In Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 273–276.
28. Akgun, I.U.; Aydin, A.S.; Burford, A.; McNeill, M.; Arkhangelskiy, M.; Zadok, E. Improving Storage Systems Using Machine Learning. *ACM Trans. Storage* **2022**, *19*, 1–30. [CrossRef]
29. Bera, R.; Kanellopoulos, K.; Nori, A.; Shahroodi, T.; Subramoney, S.; Mutlu, O. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, 18–22 October 2021; pp. 1121–1137.
30. Doudali, T.D.; Blagodurov, S.; Vishnu, A.; Gurumurthi, S.; Gavrilovska, A. Kleio: A hybrid memory page scheduler with machine intelligence. In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Phoenix, AZ, USA, 24–28 June 2019; pp. 37–48.
31. Sun, H.; Sun, C.; Tong, H.; Yue, Y.; Qin, X. A Machine Learning-Empowered Cache Management Scheme for High-Performance SSDs. *IEEE Trans. Comput.* **2024**, *73*, 2066–2080. [CrossRef]
32. Chang, J.; Doh, W.; Moon, Y.; Lee, E.; Ahn, J.H. IDT: Intelligent Data Placement for Multi-tiered Main Memory with Reinforcement Learning. In Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, Pisa, Italy, 3–7 June 2024; HPDC '24; pp. 69–82. [CrossRef]

33. Zhang, Y.; Zhou, K.; Huang, P.; Wang, H.; Hu, J.; Wang, Y.; Ji, Y.; Cheng, B. A Machine Learning Based Write Policy for SSD Cache in Cloud Block Storage. In Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Virtual Event, 9–13 March 2020; pp. 1279–1282. [CrossRef]
34. Liu, E.Z.; Hashemi, M.; Swersky, K.; Ranganathan, P.; Ahn, J. An Imitation Learning Approach for Cache Replacement. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, Virtual Event, 12–18 July 2020; pp. 6193–6203.
35. Dai, D.; Bao, F.S.; Zhou, J.; Chen, Y. Block2Vec: A Deep Learning Strategy on Mining Block Correlations in Storage Systems. In Proceedings of the 2016 45th International Conference on Parallel Processing Workshops (ICPPW), Philadelphia, PA, USA, 16–19 August 2016; Volume 2016-Septe, pp. 230–239.
36. Braun, P.; Litz, H. Understanding Memory Access Patterns for Prefetching. In Proceedings of the International Workshop on AI-assisted Design for Architecture (AIDArc)—In conjunction with International Symposium on Computer Architecture (ISCA), Phoenix, AZ, USA, 22–26 June 2019.
37. He, J.; Bent, J.; Torres, A.; Grider, G.; Gibson, G.; Maltzahn, C.; Sun, X.H. I/O acceleration with pattern detection. In Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC), New York City, NY, USA, 17–21 June 2013; pp. 25–36.
38. Pang, L.; Alazzawe, A.; Kant, K.; Swift, J. Data Heat Prediction in Storage Systems Using Behavior Specific Prediction Models. In Proceedings of the 2019 IEEE 38th International Performance Computing and Communications Conference, IPCCC 2019, London, UK, 29–31 October 2019; pp. 1–8.
39. Tracolli, M.; Baioletti, M.; Ciangottini, D.; Poggioni, V.; Spiga, D. An Intelligent Cache Management for Data Analysis at CMS. In *Proceedings of the Computational Science and Its Applications—ICCSA 2020*; Springer International Publishing: New York City, NY, USA, 2020; pp. 320–332.
40. Herodotou, H. AutoCache: Employing Machine Learning to Automate Caching in Distributed File Systems. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), Macao, China, 8–11 April 2019; pp. 133–139.
41. Tracolli, M.; Baioletti, M.; Poggioni, V.; Spiga, D. Effective Big Data Caching through Reinforcement Learning. In Proceedings of the 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 14–17 December 2020; pp. 1499–1504.
42. Liao, C.; Zheng, J. Boosting data access based on predictive caching. In Proceedings of the 2011 IEEE 3rd International Conference on Communication Software and Networks, Xi'an, China, 27–29 May 2011; pp. 93–97.
43. Gu, R.; Li, C.; Shu, P.; Yuan, C.; Huang, Y. Adaptive cache policy scheduling for big data applications on distributed tiered storage system. *Concurr. Comput. Pract. Exp.* **2019**, *31*, 1–25. [CrossRef]
44. Nicolas, L.M.; Thomas, L.; Hadjadj-Aoul, Y.; Boukhobza, J. SLRL: A Simple Least Remaining Lifetime File Evicition policy for HPC multi-Tier storage systems. In Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS) - In Conjunction with EuroSys 2022, Rennes, France, 5–8 April 2022; Volume 1, pp. 33–39.
45. Hsu, Y.F.; Irie, R.; Murata, S.; Matsuoka, M. A Novel Automated Cloud Storage Tiering System through Hot-Cold Data Classification. *IEEE Int. Conf. Cloud Comput. Cloud* **2018**, *2018*, 492–499.
46. Shetti, M.M.; Li, B.; Du, D.H. Machine Learning-based Adaptive Migration Algorithm for Hybrid Storage Systems. In Proceedings of the 2022 IEEE International Conference on Networking, Architecture and Storage (NAS), Philadelphia, PA, USA, 3–4 October 2022; pp. 1–8.
47. Mbunge, E.; Batani, J.; Fashoto, S.G.; Akinnuwesi, B.; Gurajena, C.; Opeyemi, O.G.; Metfula, A.; Ncube, Z.P. The Future of Next Generation Web: Juxtaposing Machine Learning and Deep Learning-Based Web Cache Replacement Models in Web Caching Systems. In *Proceedings of the Computer Science On-Line Conference*; Springer: New York City, NY, USA, 2023; pp. 426–450.
48. Wang, Z.; Hu, J.; Min, G.; Zhao, Z.; Wang, Z. Agile Cache Replacement in Edge Computing via Offline-Online Deep Reinforcement Learning. *IEEE Trans. Parallel Distrib. Syst.* **2024**, *35*, 663–674. [CrossRef]
49. Akgun, I.U.; Aydin, A.S.; Shaikh, A.; Velikov, L.; Zadok, E. A Machine Learning Framework to Improve Storage System Performance. In Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, 27–28 July 2021; HotStorage '21; pp. 94–102.
50. Zhong, C.; Gursoy, M.C.; Velipasalar, S. A deep reinforcement learning-based framework for content caching. In Proceedings of the 52nd Annual Conf. on Information Sciences and Systems (CISS), Princeton, NJ, USA, 21–23 March 2018; pp. 1–6.
51. Subedi, P.; Davis, P.E.; Parashar, M. Leveraging Machine Learning for Anticipatory Data Delivery in Extreme Scale In-situ Workflows. In Proceedings of the 2019 IEEE International Conference on Cluster Computing (CLUSTER), Albuquerque, NM, USA, 23–26 September 2019; Volume 2019, pp. 1–11.
52. Tanzil, S.M.S.; Hoiles, W.; Krishnamurthy, V. Adaptive Scheme for Caching YouTube Content in a Cellular Network: Machine Learning Approach. *IEEE Access* **2017**, *5*, 5870–5881. [CrossRef]
53. Song, J.; Sheng, M.; Quek, T.Q.; Xu, C.; Wang, X. Learning Based Content Caching and Sharing for Wireless Networks. *IEEE Trans. Commun.* **2017**, *65*, 4309–4324. [CrossRef]

54. Li, D.; Sun, H.; Qin, X. iCache: An Intelligent Cache Allocation Strategy for Multi-Tenant in High-Performance Solid State Disks. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2024**, *2024*. [CrossRef]
55. Barry, M.; Montiel, J.; Bifet, A.; Wadkar, S.; Manchev, N.; Halford, M.; Chiky, R.; Jaouhari, S.E.; Shakman, K.B.; Fehaily, J.A.; et al. StreamMLOps: Operationalizing Online Learning for Big Data Streaming & Real-Time Applications. In Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE), Anaheim, CA, USA, 3–7 April 2023; pp. 3508–3521. [CrossRef]
56. Rodríguez-Mazahua, N.; Rodríguez-Mazahua, L.; López-Chau, A.; Alor-Hernández, G.; Peláez-Camarena, S.G. Comparative Analysis of Decision Tree Algorithms for Data Warehouse Fragmentation. In *New Perspectives on Enterprise Decision-Making Applying Artificial Intelligence Techniques*; Springer: New York City, NY, USA, 2021; pp. 337–363.
57. Odysseos, L.; Herodotou, H. On Combining System and Machine Learning Performance Tuning for Distributed Data Stream Applications. *Distrib. Parallel Databases* **2023**, *41*, 411–438. [CrossRef]
58. Herodotou, H.; Chatzakou, D.; Kourtellis, N. A Streaming Machine Learning Framework for Online Aggression Detection on Twitter. In Proceedings of the 2020 IEEE International Conference on Big Data (Big Data), Virtual Event, 10–13 December 2020; pp. 5056–5067.
59. Vonitsanos, G.; Panagiotakopoulos, T.; Kanavos, A.; Kameas, A. An Apache Spark Framework for IoT-enabled Waste Management in Smart Cities. In Proceedings of the 12th Hellenic Conference on Artificial Intelligence, Corfu, Greece, 7–9 September 2022; SETN '22. [CrossRef]
60. Luo, L.; Risk, M.; Shi, X. Online Causal Inference with Application to Near Real-time Post-market Vaccine Safety Surveillance. *Stat. Med.* **2024**, *43*, 2734–2746. [CrossRef] [PubMed]
61. Vowpal Wabbit. Available online: https://github.com/VowpalWabbit/vowpal_wabbit (accessed on 31 March 2025).
62. Jubatus: Distributed Online Machine Learning Framework. Available online: http://jubat.us/en/ (accessed on 31 March 2025).
63. streamDM: Data Mining for Spark Streaming. Available online: https://huawei-noah.github.io/streamDM/ (accessed on 31 March 2025).
64. Apache Samoa. Available online: https://github.com/apache/incubator-samoa (accessed on 31 March 2025).
65. Lundberg, S.M.; Lee, S.I. A Unified Approach to Interpreting Model Predictions. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1–10.
66. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794.
67. Gama, J.a.; Sebastião, R.; Rodrigues, P.P. On Evaluating Stream Learning Algorithms. *Mach. Learn.* **2013**, *90*, 317–346. [CrossRef]
68. Bifet, A.; de Francisci Morales, G.; Read, J.; Holmes, G.; Pfahringer, B. Efficient Online Evaluation of Big Data Stream Classifiers. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, Australia, 10–13 August 2015; pp. 59–68.
69. MOA Machine Learning for Data Streams. Available online: https://moa.cms.waikato.ac.nz/ (accessed on 31 March 2025).
70. Lucas Filho, E.R.; Odysseos, L.; Yang, L.; Fu, K.; Herodotou, H. DITIS: A Distributed Tiered Storage Simulator. *Infocommun. J.* **2022**, *14*, 18–25. [CrossRef]
71. The Weka Workbench. Available online: https://ml.cms.waikato.ac.nz/weka/ (accessed on 30 January 2025).
72. WekaDeeplearning4j: Deep Learning Using Weka. Available online: https://deeplearning.cms.waikato.ac.nz/ (accessed on 30 January 2025).
73. Gill, B.S.; Modha, D.S. SARC: Sequential Prefetching in Adaptive Replacement Cache. In Proceedings of the USENIX Annual Technical Conference, USENIX, Anaheim, CA, USA, 7–9 July 2005; pp. 293–308.
74. Phothilimthana, P.M.; Kadekodi, S.; Ghodrati, S.; Moon, S.; Maas, M. Thesios: Synthesizing Accurate Counterfactual I/O Traces from I/O Samples. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, San Diego, CA, USA, 27 April–1 May 2024; ASPLOS '24; pp. 1016–1032. [CrossRef]
75. Phothilimthana, P.M.; Kadekodi, S.; Ghodrati, S.; Moon, S.; Maas, M. Google Thesios I/O Traces (SNIA IOTTA Trace Set 36818). Available online: https://iotta.snia.org/traces/parallel/36818 (accessed on 30 January 2025).
76. Al-Dulaimy, A.; Jansen, M.; Johansson, B.; Trivedi, A.; Iosup, A.; Ashjaei, M.; Galletta, A.; Kimovski, D.; Prodan, R.; Tserpes, K.; et al. The Computing Continuum: From IoT to the Cloud. *Internet Things* **2024**, *27*, 101272. [CrossRef]
77. Merseedi, K.J.; Zeebaree, S.R. The Cloud Architectures for Distributed Multi-cloud Computing: A Review of Hybrid and Federated Cloud Environment. *Indones. J. Comput. Sci.* **2024**, *13*, 1644–1673. [CrossRef]
78. Shan, Y.; Tsai, S.Y.; Zhang, Y. Distributed shared persistent memory. In Proceedings of the 2017 Symposium on Cloud Computing, Santa Clara, CA, USA, 25–27 September 2017; pp. 323–337.