Raven: Belady-Guided, Predictive (Deep) Learning for In-Memory and Content Caching

Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, Zhi-Li Zhang
University of Minnesota - Twin Cities
{hu000007, ye000094, tianx399}@umn.edu & {eman, zhzhang}@cs.umn.edu

ABSTRACT

Performance of caching algorithms not only determines the quality of experience for users, but also affects the operating and capital expenditures for cloud service providers. Today's production systems rely on heuristics such as LRU (least recently used) and its variants, which work well for certain types of workloads, and cannot effectively cope with diverse and time-varying workload characteristics. While learning-based caching algorithms have been proposed to deal with these challenges, they still impose assumptions about workload characteristics and often suffer poor generalizability.

In this paper, we propose Raven, a general learning-based caching framework that leverages the insights from the offline optimal Belady algorithm for both in-memory and content caching. Raven learns the distributions of objects' next-request arrival times without any prior assumptions by employing *Mixture Density Network (MDN)-based universal distribution estimation*. It utilizes the estimated distributions to compute the probability of an object that arrives farthest than any other objects in the cache and evicts the one with the largest such probability, regulated by the sizes of objects if appropriate. Raven (*probabilistically*) approximates Belady by explicitly accounting for the stochastic, time-varying, and non-stationary nature of object arrival processes. Evaluation results on production workloads demonstrate that, compared with the best existing caching algorithms, Raven improves the object hit ratio and byte hit ratio by up to 7.3% and 7.1%, respectively, reduces the average access latency by up to 17.9% and the traffic to the origin servers by up to 18.8%.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**;
• **Computing methodologies → Machine learning**.

ACM Reference Format:
Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, Zhi-Li Zhang. 2022. Raven: Belady-Guided, Predictive (Deep) Learning for In-Memory and Content Caching. In *The 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22), December 6–9, 2022, Roma, Italy*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3555050.3569134

CoNEXT '22, December 6–9, 2022, Roma, Italy

ACM ISBN 978-1-4503-9508-3/22/12…$15.00
https://doi.org/10.1145/3555050.3569134

1 INTRODUCTION

Caches are an integral part of various computer systems and networks. With today's large-scale, geographically distributed cloud services, the performance of caches not only determines the quality of experience (QoE) for users, but also affects the operating and capital expenditures for cloud service providers. For example, by caching frequently accessed data originating from a backend data store, in-memory caching systems such as Memcached [29] and Redis [51] can significantly reduce the access latency and speed up repeated computations for web applications [2, 78]. Likewise, judiciously caching objects in content delivery networks (CDNs) not only accelerates the response time to user requests, but also reduces the WAN (wide-area network) bandwidth consumption between CDN nodes and the origin content servers [1, 68]. With the rise of 5G networks and edge computing, the role of caching will become ever profound.

Optimizing cache performance has been a long standing and widely studied problem. Various heuristic caching policies from simple ones such as least recently used (LRU) and least frequently used (LFU), to more sophisticated ones (see, *e.g.,* [4, 5, 11, 13, 14, 26, 45, 46, 55, 56]) have been proposed and developed over the years. Challenges in the design of effective caching algorithms lie in that the workload characteristics (*e.g.,* object access patterns or request processes) are unknown, cannot be neatly modeled mathematically, and often vary over time. As a result, an effective heuristics designed for one type of workload may not work well when applied to other types of workloads, or when the workload characteristics change. Coping with such challenges has led to *learning-based* caching algorithms. One popular approach is to apply machine learning to predict object popularities over time [16, 43, 69, 70, 77]. These algorithms are essentially analogous to the LFU policy, and hence are no longer the "optimal" policy for recency-pattern workloads. Another popular approach is to apply online learning or reinforcement learning to directly learn a caching policy by optimizing a certain reward or regret function [41, 49, 58, 64, 79]. These learning algorithms either still impose explicit or implicit strong assumptions about workloads, or resort to "black-box" deep reinforcement learning which has significant complexity and suffer from poor stability, generalizability, and explainability.

A particularly promising approach for designing cache algorithms is to leverage Belady algorithm [8] for cache decision making [9, 37, 39, 50, 60, 68]. Given a workload, Belady always evicts the object whose *next request arrival time is the farthest in the future among all objects currently in the cache*. This strategy is *provably optimal*, however it is an *offline* algorithm, *which assumes the knowledge of all future requests*. Hence, the key issue here is how to leverage the insight of Belady in cache algorithm designs without

the knowledge of future request arrival processes. One line of pursuit is to directly *learn and imitate* the decision making of Belady algorithm, *e.g.,* via reinforcement learning [50], or directly *learn and predict* the objects' *next request arrival times* [68] and apply Belady algorithm in one way or another. A major issue all these Belady-based algorithms overlook is the fact that the decisions made by Belady are *deterministic* and *specific* to a given workload. In other words, given that the real-world caching environment is inherently *stochastic, time-varying*, and often *nonstationary*, the decisions made by Belady are *sample-path specific*. The existing Belady-based algorithms fundamentally ignore the *stochasticity*, *time-varying*, and *nonstationary* nature of object arrival processes.

In this paper, we present Raven, a Belady-guided, learning-based caching framework for both in-memory and content caching. Different from existing algorithms that are also inspired by the Belady algorithm (see §2), Raven explicitly accounts for the *stochastic*, *time-varying*, and *nonstationary* nature of object arrival processes while making caching decisions. For this, we first design a mixture density network (MDN) to learn arrival processes of objects based on their historical requests. Then, we propose an estimated version of *order statistics* to rank objects for eviction decisions based on the estimated arrival processes. To efficiently and practically learn distributions of object next-arrival times, we judiciously design our MDN to address four major **challenges**: i) scaling to millions of objects, ii) estimating any unknown distribution without any prior assumptions, iii) modeling the nonstationarity of the distribution, and iv) handling data scarcity of distribution estimation for infrequent objects. As for the rank order statistics of objects in cache, we employ two sampling techniques to estimate the probability that an object's "residual time" to the next request given the current time is the farthest in the future than all other objects. Raven selects the object with the largest estimated probability for eviction and has a constant eviction time. To further cope with objects with variable sizes, Raven explicitly incorporates the object size in caching decision making to achieve different goals, *e.g.,* to optimize the object hit ratio for improved QoE performance, or to optimize the byte hit ratio to reduce the WAN bandwidth consumption.

Compared to previous Belady-based caching algorithms [38, 40, 50, 61, 68], our method has three major benefits. First, it is more general and can be applied to any object arrival process and cache system setting, as it does not rely on any feature engineering, only the objects' past request times. Second, our method is more stable in its decision making by explicitly accounting for the stochasticity of the arrival processes and learning the distribution (see evaluation results §3.5). Last but not the least, we add explainability to the performance of our framework by approximating the full decision-making process of the Belady algorithm. For example, apart from the (unavoidable) errors in the estimation of (unknown) next-arrival time distributions, the gap between the optimal Belady decisions and those by Raven can be largely attributed to the uncertainty and randomness inherent in the object (request) arrival processes.

We evaluate Raven[1] on synthetic workloads with various arrival processes to prove the concepts of Raven, as well as large-scale real-world traces, which have more complicated request patterns and object size distributions, to validate the performance improvement and scalability of Raven. On three CDN production cache traces and three in-memory cache traces from Twitter, Raven consistently outperforms the state-of-the-art caching algorithms across all experiments. Overall, Raven improves the object hit ratio and byte hit ratio by up to 7.3% and 7.1%, respectively, reduces the average access latency by up to 17.9% and backend traffic by up to 18.8%.

[1]The source code of Raven is available at https://github.com/RavenCaching.

2 BACKGROUND, RELATED WORK, AND MOTIVATION

Caching mechanisms have been studied extensively since 1960s. Most earlier designs have relied on various heuristics that often impose strong assumptions on the object request arrival processes, such as the object arrivals following a Poisson process or more generally an Independent Reference Model (IRM). The classical examples are LRU (least recent used) and many of its variants, which have been widely used in existing systems. The other example is LFU (least frequently used) policy, which yields the optimal object hit ratio under the (stationary) Poisson or IRM assumption. Most cache heuristics are improvements upon these two basic policies, more recent examples include TinyLFU [26], ARC [55], and Hyperbolic [13]; the literature is too numerous to list here. We refer the reader to several survey papers and relevant recent research papers (and references therein) [5, 6, 13, 27, 31, 42, 45, 53, 55–57, 65][2]. In the following, we focus instead on *learning-based* cache policies. This provides the background and motivation for the design of our proposed Raven framework.

2.1 General Learning-based Adaptive Caching Algorithms

A key problem that plagues heuristics-based caching algorithms in practice is that i) cache workloads, or object arrival processes, come from an unknown distribution or process that cannot be neatly modeled mathematically; and ii) they often change over time. This has led to a flurry of interest in learning-based caching mechanism designs utilizing historical data, see *e.g.,* [16, 43, 49, 64, 69, 70]. Many of the learning-based caching algorithms focus on predicting object popularity. One line of research still makes explicit model assumptions about the object popularity distribution or arrival processes, and relies on machine learning methods, *e.g.,* Bayesian inference, to estimate model parameters. For example, the most recent caching algorithm LHR [77] estimates hazard rates to represent object popularity by assuming request processes are Poisson. The main drawback of these popularity-estimation-based learning algorithms is that they are essentially an adaptive version of LFU, namely cache most *popular* objects only. In other words, they ignore the temporal dynamics of the object arrival processes; in contrast, a cache oracle in fact may not always cache the most popular objects as we discuss in §2.3.

Another popular line of research casts the caching problem as learning how to make caching decisions to optimize predefined cache utility (aka rewards or regret). They either apply online learning [58] or reinforcement learning [41, 64, 79] algorithms to dynamically learn the *optimal* policy for decision-making. Unfortunately, due to the large state-action space, they are significantly

[2]For more details; this GitHub repository [73] also contains a list of popular cache policies and their implementations.

more complex, sensitive to hyper-parameters, and often suffer from delayed rewards leading to slow reaction times in dynamic environments [9]. In fact, these issues have led the authors in [44] to conclude that caching is "not amenable to training good policies." In addition, the models learned through these methods often lack stability, generalizability, and explainability.

2.2 Learning-Augmented Caching Algorithms

In general, caching algorithms based on ML models work well with accurate predictions, but can perform poorly when prediction errors are large. To be robust to prediction errors, another line of research, called learning-augmented algorithms [3, 17, 52, 63, 74], treats the ML algorithm as an "oracle" and focuses on redesigning online caching algorithms to leverage ML predictions with an emphasis on deriving bounds on worst-case performance (with respect to optimal offline algorithms, or bounds on competitive ratios. For example, Lykouris *et al.* [52] proposed PredictiveMarker, which augments the classic MARKER algorithm [28] with ML predictions of object reuse distance. PredictiveMarker achieves good competitive ratio and is robust even when the predictions are completely wrong. While learning-augmented algorithms provide theoretically provable upper bounds on competitive ratio, competitive analysis fails to distinguish between practical and theoretical algorithms [13, 52].

2.3 Belady-based Learning Algorithms

For a given workload where all objects are of *fixed* size, we in fact know the *cache oracle* for making *optimal* eviction decisions to *maximize object hit ratio*. This is Belady's MIN algorithm [8], known since 1966, which is an *offline* algorithm assuming that all *future* object request arrival times are known. Upon a cache miss, Belady algorithm always evicts the object in the cache whose *next request arrival time* is the *farthest* in the future among all objects in the cache. This insight has given rise to several papers which employ Belady algorithm to guide eviction decisions. The earliest work [61], as well as more recent ones [38, 40], directly *emulate* Belady algorithm for caching and prefetching decisions using a window of *past* memory accesses in hardware cache designs.

More closely related to our work, Berger [9] first advocates directly learning from the optimal caching decisions using machine learning, and proposes LFO for CDN caching which trains a gradient boosting decision tree using *manually-designed features* as a binary classifier for caching decisions. Designed for memory cache systems, Parrot [50] uses a deep reinforcement learning (DRL) framework to imitate Belady algorithm for caching decisions based on past requests and current cache state. Unlike LFO, Parrot avoids feature engineering, but the heavy DRL framework makes it difficult to scale to systems with millions of objects. The recent learning-based caching algorithm, LRB [68], also leverages Belady algorithm for caching decisions. However, in contrast to LFO and Parrot, instead of *imitating* the "optimal decisions" made by Belady (via either a learned binary classifier or via a DRL framework), LRB aims to directly learn and *predict* the *next arrival times* of object requests, and then resorts to a *relaxed* Belady algorithm for caching evictions. Similar to LFO, LRB employs a gradient boosting machine (GBM) [30] based on *manually-designed features*. While LRB has shown superior performance over existing caching algorithms in terms of byte hit ratios for CDN caching, the manual feature engineering limits its applicability and generalizability to more general object arrival processes and non-CDN application scenarios (*e.g.,* in-memory cache systems). The *ad hoc* manner in which features are selected and constructed also obscures its explainability.

2.4 Casting Arrival Process Estimation as the Key to Belady-Guided Caching Design

A key challenge in directly applying Belady algorithm to caching systems lies in that the optimal decisions made by Belady are (workload or) *sample-path specific*. Hence, directly imitating the decisions made by Belady algorithm as in [9, 50] to learn a cache policy may not be the best approach (see the evaluation results in §3.5). A perhaps more effective approach is to learn and predict the objects' future next-arrival times, and then apply the same optimal strategy used in Belady algorithm for making caching decisions, namely, evicting the object with the farthest next arrival time. The problem then boils down to effectively predicting object next-arrival times given the inherent *stochasticity*, *uncertainty*, and *non-stationarity* in the time-varying object arrival processes.

Unlike LRB which directly predicts a *deterministic estimation* of object next-arrival time based on hand-crafted features, our proposed framework, Raven, utilizes *probabilistic estimation* of next-arrival time. It models an object's next-arrival time as a non-stationary distribution and predicts the next-arrival time as a list of possible values by sampling from the distribution. This probabilistic estimation not only provides the expectation, but also the variance of the next-arrival time. On the other hand, LRB predicts next-arrival time as a single value (typically around the mean value). This deterministic estimation does not convey any variability or stochasticity of the arrival process. For example, considering a scenario where the next-arrival time distributions of N objects have the same mean but different variances. It will be challenging for LRB to differentiate the N objects as the predictions of LRB are around the mean value. However, Raven can differentiate them by considering their variances, and preferably evicts the object with the largest variance. By utilizing the power of probabilistic estimation, Raven accounts for the inherent stochastic and time-varying nature of object arrival processes, and *probabilistically* approximate Belady. Therefore, Raven is more stable, generalizable, and explainable.

3 LEARNING OBJECT ARRIVAL PROCESSES TO APPROXIMATE BELADY

Belady algorithm has been proven to be the optimal caching policy on traces where objects have identical size [8]. Based on knowledge of future requests, Belady evicts the object whose next request is the farthest. To approximate Belady, this section introduces an online caching algorithm, named Raven, which utilizes machine learning to learn arrival processes of objects. Then, evict the object with the largest probability of being the selected victim by Belady. We will discuss the design details of Raven in §4.

3.1 Raven Framework

We design a powerful density estimation neural network to learn and predict object next-arrival times, considering the inherent

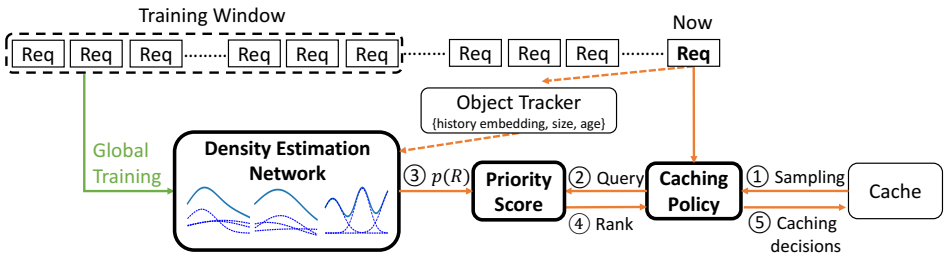


Figure 1: Overview of the Raven framework. It estimates the arrival processes of objects using Density Estimation Neural Network. The priority score of an object is derived from the estimated arrival process and represents the probability that the object arrives farthest in the future compared to other cached objects. When a cache miss happens, the object with the largest priority score is evicted.

stochasticity, *uncertainty*, and *nonstationarity* in the time-varying object arrival processes. Upon a cache miss, based on the predicted arrival processes of cached objects, Raven evicts the object with the largest probability that its next arrival is the farthest. We further extend Raven to be aware of variable object sizes in §3.4.

The framework of Raven is depicted in Fig. 1. It consists of three major components: *i) Density Estimation Network* which uses a memory window of past requests as training data and estimates arrival processes of objects based on their historical inter-arrival time information, ages, and sizes (§4.2); *ii) Priority Score* which computes the probability that an object will arrive farthest (§3.3 & §3.4); *iii) Caching Policy* which ranks the cached objects based on their priority scores, then evicts the one with the largest score to approximate Belady.

The caching process in Fig. 1 works as follows. For each object in cache, Raven maintains a feature vector, namely, object size, age, and history embedding automatically extracted by the density estimation network (§4.2.1). Upon a new request for object O_i, Raven records its information to construct training dataset to later update the density estimation network (§4.1). Upon a cache miss, a number of cached objects are randomly sampled as eviction candidates, then their next arrival processes are predicted based on their feature vectors via the density estimation network (§4.2.2 & §4.2.3). The priority scores of eviction candidates are derived from the estimated arrival processes, and the object with the largest score is evicted.

3.2 Learning Arrival Processes

Problem formulation. We represent the overall request arrival process generated by a large population of users as the superposition of many independent random processes, each referring to an individual object [70]. The arrival process of an object is characterized by a sequence of increasing arrival times $\{t_1, ..., t_N\}$ or equivalently by a sequence of inter-arrival times $\{\tau_1, ..., \tau_{N-1}\}$, where $\tau_i = t_{i+1} - t_i$. Inferring the arrival process is equivalent to estimating $p(\tau)$ from the observed inter-arrival times, where $p(\tau)$ is the density distribution of the inter-arrival time τ, *priorly unknown*, and *time-varying* in practice.

Universal Arrival Process Approximation. Learning object arrival processes for caching has four major *challenges*: 1) *scaling* to millions of objects; 2) estimating *arbitrary* arrival processes without any prior assumptions; 3) modeling *nonstationarity* of arrival processes; and 4) solving *data scarcity* caused by infrequent objects while estimating distributions. To address these challenges, we design a mixture density network (MDN) (*i.e.,* a specific type of density estimation network) [12, 54, 66]. Unlike previous works which assume a pre-specified arrival model (*e.g.,* Poisson process [70, 77], renewal process [43], self-exciting process [69]), the universal approximation capability of our MDN enables Raven to estimate arbitrary arrival processes. To account for the nonstationarity of arrival processes, the proposed MDN uses a recurrent neural network (RNN) layer to automatically extract temporal dependency features from objects' historical inter-arrival times. Therefore, an object's estimated arrival process depends on its history. Our MDN maps objects with different arrival histories to different arrival processes, and hence scales to millions of objects. As for data scarcity, we solve it by using survival probability and data from a large number of infrequent objects. Our MDN is described in detail in §4.2.

3.3 Predicting the Farthest Next (Request) Arrival of an Object

Based on the estimated object arrival processes, to approximate Belady which evicts the object with the largest residual time, Raven calculates the *order statistics* (denoted as priority scores) of objects to rank objects in cache and find the farthest arrival object.

Priority Score. The priority score of an object is defined as the probability that its residual time[3] is greater than the residual time of any other object in cache. Let's denote the age of an object O_j as a_j and its residual time as R_j. The priority score p_j of O_j is calculated as:

$$p_j = Pr\{R_j > R_1, ..., R_j > R_k, ..., R_j > R_C\} \tag{1a}$$

$$= \int_0^\infty p_{R_j}(t) \prod_{k \neq j} F_{R_k}(t) dt \tag{1b}$$

$$\approx \frac{\sum_{m=1}^{M} I\{r_{jm} > r_{km}, \forall k, k \neq j\}}{M} \tag{1c}$$

where C is the maximum number of objects in cache. $p_{R_j}(t)$ is object O_j's residual time density distribution, and $F_{R_k}(t)$ is object O_k's cumulative distribution function. Both $p_{R_j}(t)$ and $F_{R_k}(t)$ depend on object's history and can be easily obtained by our MDN in closed-form. r_{jm} is the m-th sample drawn from O_j's residual time distribution $p_{R_j}(t)$. M is the total residual time sample number. $I\{r_{jm} > r_{km}, \forall k, k \neq j\}$ is 1 if the m-th residual time sample of O_j is greater than the m-th residual time sample of any other objects and is 0 otherwise. Appendix A shows the full proof.

[3]Residual time is defined as given an object's current age, the time it takes for its next-arrival.

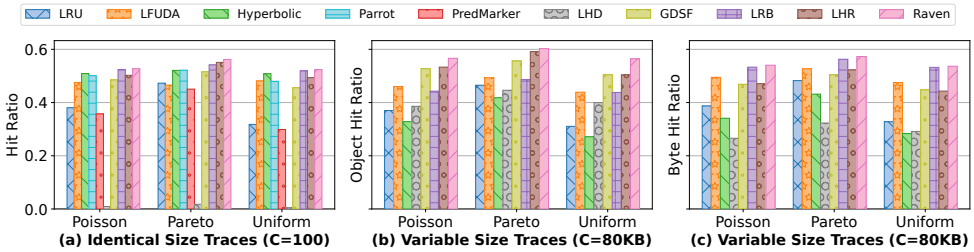


Figure 2: Hit ratios on synthetic traces (C: Cache Size).

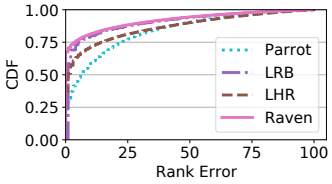


Figure 3: CDF of rank-order errors on the Uniform trace.

The proposed priority score represents the probability that an object is the selected victim by Belady. It approximates Belady while taking the randomness and uncertainties of arrival processes into consideration. While the exact priority score in Equation 1b is optimal, it is too complicated and computationally expensive. To reduce runtime during eviction, Raven approximates the exact priority score with residual time samples as in Equation 1c. The complexity of this approximation is $O(M)$, where M is the residual time sample number. As M increases, the approximated priority score converges to the exact priority score (see more discussion of residual time sample number M in §4.3.2).

3.4 Dealing with Variable Object Sizes

In reality, object sizes can span several orders of magnitude, especially in CDNs and web applications. The goal of a caching algorithm can be to maximize either object hit ratio (OHR) or byte hit ratio (BHR), depending on the penalty of a cache miss. Toward these two different goals, two Belady variants are widely used as optimal algorithms [10, 68]. This subsection describes how Raven is extended to be size-aware based on the Belady variants.

OHR goal. OHR goal treats the cost of each cache miss equally, and aims at caching as many objects as possible with limited cache space. To maximize OHR, a widely-used extension of Belady is to evict the object with the highest cost = object size × next-use distance. A more accurate upper bound on OHR is provided by practical flow-based offline optimal (PFOO) algorithm [10]. Overall, these two algorithms imply that a caching algorithm should keep small objects that will result in a cache hit quickly to maximize OHR. Based on this insight, Raven augments the original priority score (see Equation 1) with object size: $\hat{p}_i = s_i \times p_i$ and ranks objects based on the new priority scores.

BHR Goal. BHR goal values the cost of each cache miss with regard to its object size, and aims at reducing the traffic and expense of fetching missed objects. The original Belady algorithm is believed to be a near-optimal algorithm in this case, and has been used as a guideline in LRB [68] to maximize BHR. Therefore, Raven uses the original priority score to achieve the BHR goal.

3.5 Simulation Results

This subsection uses three synthetic traces to show that Raven can adapt to different workloads, and consistently outperforms the state-of-the-art caching algorithms. Each trace contains 10M requests of 1000 objects. To represent different workloads, the inter-arrival time distributions in the three traces are Poisson, Uniform, and Pareto distributions, respectively. We compare Raven with heuristics algorithms (*e.g.,* Hyperbolic [13], LFUDA, LRU, GDSF, and LHD [7]), as well as learning-based algorithms (*e.g.,* LRB [68], LHR [77], Parrot [50], and PredictiveMarker[4] [52]). More details about the experiment setup are described in Appendix C.1. The results on real-world traces are shown in §5.

Fig. 2a shows the hit ratio results on the three traces where all objects have the same size, and the cache size is configured to hold 100 objects. Raven consistently achieves the best hit ratio across all traces. We further use rank-order error to compare the four best-performing learning-based algorithms. Rank-order error of an evicted object is the difference between its ranking using a caching algorithm and its true ranking using the future request times. Fig. 3 shows the CDF of rank-order errors on the Uniform trace. The average and variance (see details in Table 6) of rank-order errors of Raven are consistently the smallest, which indicate that Raven has more accurate and stable predictions of objects' future arrivals, because it explicitly accounts for the stochastic nature of object request arrival processes. More results of hit ratio and rank-order errors are shown in Appendix C.2.

Fig. 2b and Fig. 2c show the OHR and BHR performance on the three traces where objects have variable sizes, and cache size is configured to be 80 KB (*i.e.,* 10% of the total unique bytes). Parrot and PredictiveMarker are excluded as they cannot handle variable object sizes. Results on more cache size settings can be found in Appendix C.3. This evaluation shows that Raven consistently achieve the best object hit ratios, best byte hit ratios, and outperforms the other learning-based algorithms.

4 DESIGN OF RAVEN

Raven uses a well-designed ML model to learn arrival processes of objects and approximates Belady in a practical manner. Accomplishing this requires addressing the following key design issues: i) **Training data.** How much past information is needed and how to use this historical data for training? ii) **ML architecture.** How to efficiently learn arbitrary arrival processes for millions of objects while considering the nonstationarity nature and the scarcity of data? iii) **Eviction rule.** How to utilize the estimated arrival processes to quickly calculate priority scores and rank objects to evict? This section describes the key components of Raven that help address these design issues.

4.1 Training Data

Overall, Raven keeps information about objects that have been previously requested within the training window. This historical data

[4]The comparison with [52] on the dataset used in [52] is shown in Appendix B.

is used for training the ML model to estimate the arrival processes of objects at the end of every training window. We now describe how to choose the window size and construct the training data.

As arrival processes of objects may be nonstationary and have diurnal or time-of-day patterns [15, 71, 78], Raven considers the size of a window measured in terms of the elapsed time since the last training or the beginning of the trace. Choosing the window size is important to the performance of Raven. If the window size is too small, the request sequences of all objects may be too short, and Raven will not have enough training data for its ML model to learn arrival processes and nonstationarity. If the window size is too large, it may increase the memory overhead, processing time, and training time. To account for diurnal patterns and reduce the training frequency, we set the window size to be 1 day.

In reality, the historical data in a window of 1 day can be more than hundreds of millions due to the high request rate. To address this "data explosion", Raven periodically takes a random sample of objects, and only records information of the sampled objects. Each object has the same probability to be sampled, because we want to avoid biasing the training data towards popular objects which has many requests. The sampling rate is determined such that the number of unique bytes of the sampled objects is no more than 5× the cache size. We choose 5× the cache size as upper bound, because it is large enough to characterize the workload patterns in 1 day, and similar window setting has also been used in [77] [5]. In our experiments, we find that larger window size further increases the training time and overhead, without a noticeable improvement on hit ratios. The typical numbers of objects and numbers of request samples in training datasets are shown in the Appendix D.

We remark that the training data can be streamed to another dedicated machine to train the ML model. At the beginning of a trace, we use LRU as a fallback until the ML model is trained.

4.2 ML Architecture

This subsection introduces the ML model used by Raven to learn arrival processes and how we design it to handle 1) **scalability**, *i.e.,* learning the arrival processes of millions of objects at scale; 2) **universal estimation**, *i.e.,* estimating arbitrary arrival processes without any prior assumptions; 3) **nonstationarity**, *i.e.,* capturing the nonstationarity of arrival processes; 4) **data scarcity**, *i.e.,* coping with infrequent objects whose request sequences are short in nature; The architecture of the proposed ML model is illustrated in Fig. 4. Next, we describe its components and design rationale.

4.2.1 ***Input data: inter-arrival times, object size, and age.*** Raven uses 3 inputs to its ML model: past inter-arrival times, object size, and age. Unlike previous learning-based caching algorithms (*e.g.,* LRB [68], LHR [77]) which rely on manually-designed features, Raven uses raw data and employs neural networks (NN) to automatically extract features.

History embedding. To extract temporal dependency features, Raven uses a single-layer RNN [25] to process an object's historical inter-arrival times, and embed the arrival history into a fixed-dimensional vector $\boldsymbol{h}$ (*i.e.,* the hidden state of RNN). The RNN unit can be either vanilla RNN, or LSTM, or GRU. The type of RNN unit is a parameter that can be configured and we use GRU in our experiments. According to our experimental experiences, different RNN units do not make a significant difference, and a 32-128 RNN hidden-state size works well in most cases. A larger or smaller hidden-state size would have slightly worse performance. An object's history embedding is continuously updated as its new arrival time is observed. Unlike LRB and LHR which consider fixed-length short-term relationships (*e.g.,* LRB uses past 32 inter-arrival times), Raven takes both short-term and long-term temporal relationships into consideration by utilizing RNN. This allows Raven to solve the *nonstationarity* issue caused by time-varying access patterns.

[5] Instead of using 4× the cache size as in [77], Raven uses 5× because the MDN is more complicated than GBM used in [77].

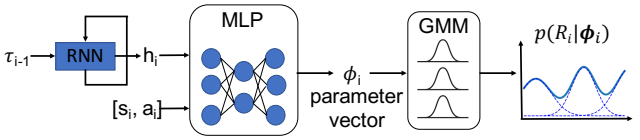


Figure 4: The architecture of mixture density network (MDN). MDN outputs mixture distribution parameters to represent any object's residual time distribution conditional on its size, age, and history embedding.

In addition to temporal dependency features, we also use object size, denoted as s, and age (*i.e.,* time since the latest request), denoted as a, to infer its future arrival process. These two pieces of information are easy to obtain and intuitively correlate with different access patterns.

4.2.2 ***Model: Mixture Density Network.*** Raven achieves *scalability* by using a global mixture density network (MDN) [12, 54, 66] to learn arrival processes for millions of objects. Our MDN is built upon the Gaussian mixture model (GMM), and approximates an arbitrary arrival process as a mixture of log-normal distributions, since inter-arrival times are positive. MDN scales GMM by employing a simple feedforward neural network to automatically extract features from input data, then map them to the parameters of GMM. Therefore, different objects will have different features and are mapped to different arrival processes.

In this work, the feedforward neural network is a three-layer fully connected multilayer perceptron (MLP) with the ReLU activation function used in hidden layers. The input of MLP is object size, current age, and history embedding. Denote the input at time t as $\boldsymbol{x} = [\boldsymbol{h}_t, s, a]$, where $\boldsymbol{h}_t = RNN(\tau_{t-1}, \boldsymbol{h}_{t-1})$. The output of MLP is the parameters of GMM, *i.e.,* the mixture weights $\boldsymbol{\omega}$, the mixture means $\boldsymbol{\mu}$, and the standard deviations $\boldsymbol{s}$. Denote the output as $\boldsymbol{\phi} = [\boldsymbol{\omega}, \boldsymbol{\mu}, \boldsymbol{s}]$) and it is obtained by

$$\boldsymbol{\omega} = \text{softmax}(\boldsymbol{W}_\omega \boldsymbol{c} + \boldsymbol{b}_\omega) \quad \boldsymbol{\mu} = \boldsymbol{W}_\mu \boldsymbol{c} + \boldsymbol{b}_\mu \quad \boldsymbol{s} = \exp(\boldsymbol{W}_s \boldsymbol{c} + \boldsymbol{b}_s) \quad (2)$$

where $\boldsymbol{c}$ is the output of MLP's last hidden layer,

$$\boldsymbol{c} = ReLU(\boldsymbol{W}_2 ReLU(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2) \quad (3)$$

To enforce the constraints on the distribution parameters $\boldsymbol{\phi}$, softmax and exp transformation are applied. $\boldsymbol{W}_*$ and $\boldsymbol{b}_*$ are the learnable parameters of MLP.

4.2.3 ***Prediction Target: residual time distribution.*** Raven uses the predicted GMM parameter vector $\boldsymbol{\phi}$ to estimate residual

time distributions for objects:

$$p(R|\boldsymbol{\phi}) = \sum_{k=1}^{K} \omega_k \frac{1}{Rs_k\sqrt{2\pi}} \exp(-\frac{(\log R - \mu_k)^2}{2s_k^2}) \tag{4}$$

where K is the number of mixture components, ω_k, μ_k, and s_k, respectively, represent the weight, mean, and the standard deviation of the k-th Gaussian component. Note that the residual time distribution is a conditional distribution that depends on object size, age, and arrival history, *i.e.,* $p(R|\boldsymbol{\phi}) = p(R|s, a, \boldsymbol{H})$, where $\boldsymbol{H}$ represents object historical inter-arrival times.

Since arbitrary stationary distributions can be approximated by a mixture of Gaussian distributions [12] and the nonstationarity of distributions can be captured by RNN, Raven achieves *universal arrival process estimation* by combining RNN and GMM. The detailed proof of universal arrival process estimation by combining RNN and MDN is shown in [66].

4.2.4 *Loss Function: log-likelihood and survival probability.*

Raven trains MDN by maximizing log-likelihood and survival probability of object request sequences which happened during the training window. To be more specific, we assume that at the end of the current training window time t, the training dataset consists of N objects, and each object has an (m_i+1)-length arrival time sequence $[t_1, t_2, ..., t_{m_i+1}]$ (*i.e.,* m_i-length inter-arrival time sequence $[\tau_1, \tau_2, ..., \tau_{m_i}]$) and a survival time $t - t_{m_i+1}$. To obtain an age associated with a past inter-arrival time sample τ_i, we randomly sample age a_i from the uniform distribution $U[0, \tau_i]$ and the corresponding residual time R_i is $\tau_i - a_i$. Same logic applies to obtaining an age associated with a survival time. Let's denote the parameters of MDN as $\boldsymbol{\theta}$. We find the optimal parameters by maximizing the following loss function:

$$\boldsymbol{\theta}^{\star} = \max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{O_i} (\sum_{i}^{m_i} \log p(R_i|\boldsymbol{\phi_i}) + \log Pr\{R_{m_{i+1}} > t - t_{m_i+1} - a_{m_i+1}|\boldsymbol{\phi}_{m_{i+1}}\}) \tag{5}$$

where $\boldsymbol{\phi}$ is the output of our MDN and is determined by MDN parameters $\boldsymbol{\theta}$, $p(R_i|\boldsymbol{\phi_i})$ is the likelihood of observing residual time R_i, and $Pr\{R_{m_{i+1}} > t - t_{m_i+1} - a_{m_i+1}|\boldsymbol{\phi}_{m_{i+1}}\}$ is the survival probability that the residual time is greater than the survival time minus age.

Typically people utilize log-likelihood to learn distributions of sequences [18, 24, 66]. In this paper, we additionally include survival probability [66] to help model short sequences, since the object popularity distribution is "long tailed" in production traces, where a small portion of objects have massive requests, while others have much less or even a few requests. By considering survival probability and learning from a large number of short sequences, MDN is able to map the history embeddings of infrequent objects with large survival times to distributions where large residual times have higher probabilities. Hence, MDN solves the *data scarcity* issue caused by infrequent objects. To demonstrate the data scarcity in the six production traces (see §5.1.1 for details of real-world traces), Table 8 in the Appendix E shows the number of one-hit wonders [6] under different cache size settings. The impact of survival probability is shown in Fig. 5, which compares the performance of Raven with and without the survival probability. We can notice that the performance of Raven improves on all traces, with significant increases on Wiki 18 and Wiki 19 traces.

[6] One-hit wonders refer to cached objects that are never accessed before they are evicted due to their lack of popularity.

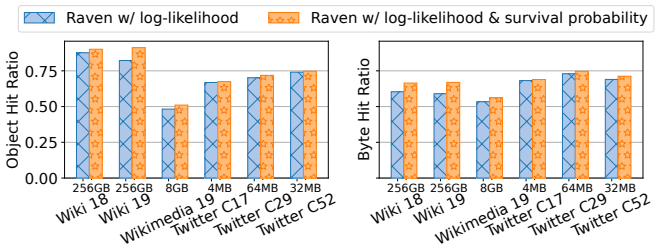


Figure 5: Impact of survival probability in loss function.

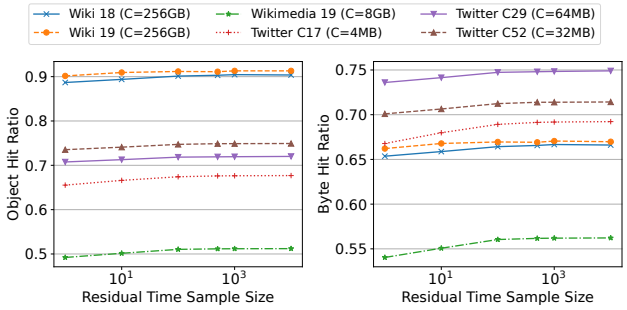


Figure 6: Residual sample size impact on hit ratios.

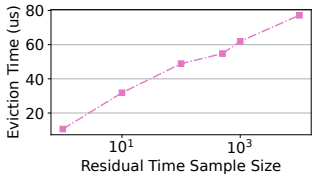


Figure 7: Residual sample size average eviction time.

4.3 Eviction Rule

In this subsection, we describe how Raven approximates the priority score defined in Equation 1 by two sampling techniques to quickly rank objects to evict.

4.3.1 *Samples from cached objects.*

Raven randomly samples cached objects to get eviction candidates, then runs a batch prediction for all candidates to estimate their residual time distributions and calculate their priority scores. Raven evicts the object with the highest priority score, which means that it has the highest probability of its next arrival request being the farthest among the other candidates. This sampling technique is widely used in recent caching policies [7, 13, 59, 68]. Similar to previous works, we choose random sample size as 64 samples. Randomly sampling 64 cached objects to evaluate reduces inference time and achieves constant ranking time with respect to the number of cached objects.

4.3.2 *Samples from residual time distributions.*

Raven uses samples from candidates' residual time distributions to approximate their priority scores as defined in Equation 1c. The complexity of this approximation is $O(M)$, where M is the residual time sample number. As M increases, the approximated priority score converges

Table 1: Key properties of the six production cache traces used throughout our evaluation.

		Wiki 18	Wiki 19	Wikimedia 19	Twitter C17	Twitter C29	Twitter C52
Total Requests		2.8 billion	2.7 billion	208 million	9.7 billion	3.8 billion	12.5 billion
Total Bytes		90 TB	99 TB	6.4 TB	5.1 TB	1.7 TB	2.9 TB
Unique Objects		38 million	51 million	49 million	29 million	326 million	728 million
Unique Bytes		5.5 TB	8.3 TB	1.3 TB	5.2 GB	115 GB	100 GB
Duration		15 days	21 days	21 days	8 days	5 days	6 days
Request Object Size	Mean	34 KB	40 KB	33 KB	575 B	482 B	258 B
	Max	1.2 GB	1.3 GB	6.6 MB	1.4 KB	712 KB	9.2 KB

to the exact priority score defined in Equation 1b. The theoretical selection of minimum sample size varies for different use cases. Generally 30-100 sample sizes are sufficient to conduct significant statistics [36] or get a meaningful result [21]. Based on our empirical experiences, the impact of residual time sample number on hit ratios is shown in Fig. 6, and its impact on the average eviction time is shown in Fig. 7. To achieve a reasonable tradeoff between approximation accuracy and computation time, Raven chooses the residual time sample size M as 100.

4.4 Putting It All Together

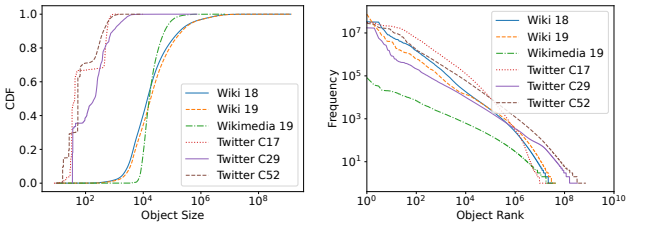
Putting all of the above components together, we have the complete design of Raven as shown in Fig. 1. Raven learns from the requested objects in a training window. A sampling process continuously samples requested objects to generate the training dataset. At the end of the training window, Raven starts training an MDN model and empties the training dataset. After that, whenever a training window is reached again, Raven repeats the process and replaces the old model with the new one. If the current requested object is not cached, we need to evict an object. Raven randomly samples 64 eviction candidates from the cached objects, runs MDN to predict their residual time distributions, draws 100 residual time samples from each predicted distributions, and calculates the priority scores. Then, Raven evicts the candidate with the highest priority score, which represents the candidate with the highest probability of having the largest residual time.

4.5 Implementation

Raven is implemented in Python and uses PyTorch [22] for ML related tasks. The whole pipeline contains 3600 lines of code. We expose the same interfaces as LRB, namely, `lookup()` for cache lookup and `admit()` for admission upon a cache miss. To allow direct comparison with Apache Traffic Server (ATS) which powers CDN servers, we integrate Raven with ATS, based upon LRB prototype, which integrates LRB caching algorithm with ATS [68]. Since the ATS is implemented in C++, Raven is integrated by employing a Python/C++ wrapper in-between Raven and ATS. When the interface from ATS, *e.g.,* `lookup()` or `admit()`, is called, the wrapper redirects the call to Raven sub-system for further execution. By this means, ATS can directly utilize Raven with minor modifications.

5 EVALUATION

We evaluate Raven on real-world cache traces to explore the following questions: 1) What is the performance of Raven compared to the state-of-the-art (SOTA) caching algorithms in terms of OHR, BHR, traffic reduction, latency, and throughput under various cache size settings? 2) What is the gap between Raven and optimal algorithms (OPTs)? 3) What is the performance of Raven prototype compared to the ATS production system in terms of OHR, BHR, traffic reduction, and what is the cost of its implementation overhead?



(a) CDF of object size **(b) Object popularity rank and frequency**

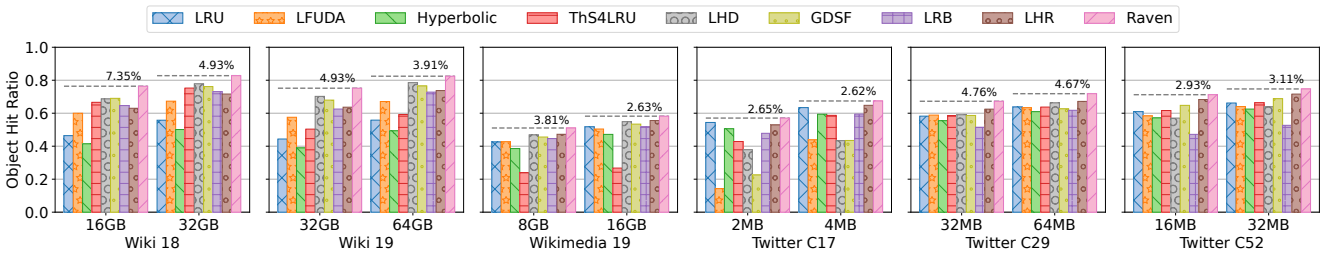
Figure 8: The object size and popularity distributions of the six production traces.

5.1 Methodology

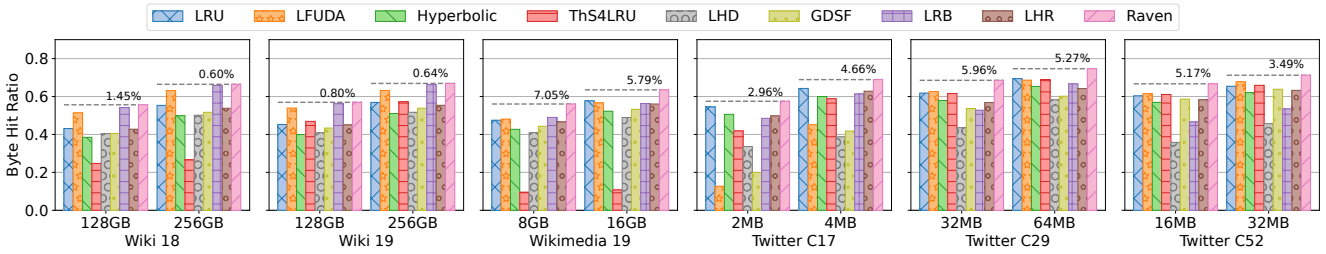
This subsection describes the real-world cache traces, the experiment setup of our simulation and testbed, the state-of-the-art algorithms, and the parameter settings of Raven.

5.1.1 ***Real-world Traces.*** We use three public CDN production traces from Wikipedia 2018, 2019 [68], Wikimedia 2019 [75] and three public in-memory production traces from Twitter in 2020 [78]. The Wikipedia traces are collected from nodes in a metropolitan area and serve a mixture of web, photo, and other media content for Wikipedia pages. The Wikimedia dataset is a restricted public snapshot of the *wmf.webrequest* table intended for caching research, which contains data on all the hits to Wikimedia's servers. As for Twitter traces, we use three miss-ratio-related traces from Twitter in-memory cache cluster 17, cluster 29, and cluster 52. We convert the in-memory trace format to our format by simply summing key size and value size to be the object size.

We summarize key properties of the traces in Table 1. Additionally, we show the characteristics and request patterns of the traces in Fig. 8. The object size distribution in Fig. 8a shows that object sizes in CDN traces can span more than eight orders of magnitude, whereas for in-memory traces (*i.e.,* Twitter) the variance of object sizes is much smaller. The object popularity distribution in Fig. 8b shows that all workloads approximately follow a Zipf distribution. Appendix F further analyzes the distributions of request numbers and requested bytes over object size and frequency.



(a) Object hit ratios



(b) Byte hit ratios

Figure 9: Object hit ratios and byte hit ratios on production traces with different cache sizes. Raven consistently outperforms all SOTAs in all traces and cache sizes combinations.

5.1.2 ***State-of-the-Art Algorithms.*** We compare Raven with 14 state-of-the-art eviction algorithms. To improve readability, we only show the eight best-performing algorithms which can be categorized into: 1) learning-based algorithms: LRB [68], LHR [77][7]; and 2) heuristics-based algorithms: LHD [7], GDSF, Hyperbolic [13], LFUDA, LRU, and ThS4LRU. We exclude Parrot [50] and PredictiveMarker [52] from comparison because they cannot handle variable object sizes and do not scale to production traces.

5.1.3 ***Raven Settings.*** Unless otherwise noted, we use the following default values to evaluate Raven. With respect to the hyperparameters of our mixture density network, the number of Gaussian mixture components K is 64, the RNN unit is GRU, the RNN hidden state (history embedding) size is 32 for CDN traces and 16 for Twitter traces to reduce the memory overhead in the case of in-memory traces. The learning rate is 0.001, and 20% of the training dataset are with-held as validation dataset. The training process is stopped early if the validation error is no longer decreasing in the previous consecutive 200 epochs. We use the first 20% of every trace to estimate the request rate and tune the sampling rate to construct the training dataset.

5.1.4 ***Testbed and Simulation Settings.*** In our prototype testbed and CDN caching simulation, three components are involved, namely, client, caching server, and origin server. Similar to the settings used in LRB [68] and LHR [77], we assume the trace-based simulation runs in an ideal environment where the network transmission rate is 8 Gbps. For latency measurements, to simulate network RTTs, we add a 10 ms delay to the link between client and caching server, and a 100 ms delay to the link between origin and caching server [68]. Similarly, in-memory experiments involve client, memory, and database. We add a 100 μs delay to the memory access and a 10 ms delay to the database access [19]. Cache sizes used for different traces are selected based on the total active bytes of each trace. All experiments are carried out on a server, which has one AMD Ryzen Threadripper PRO 3995WX with 1 TB RAM and one NVIDIA A6000 GPU with 48 GB RAM.

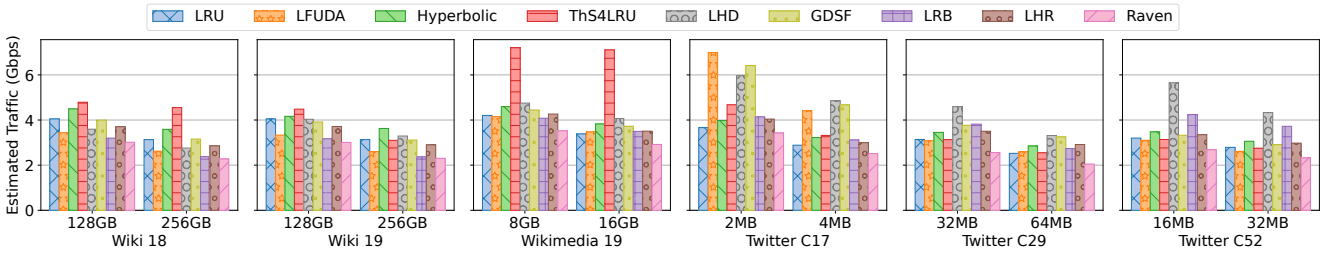
5.2 Raven vs. State-of-the-Art Algorithms

We compare Raven to 14 state-of-the-art cache eviction algorithms using simulations with various cache size settings on the six production traces described earlier. To improve readability, we show only the best eight algorithms.

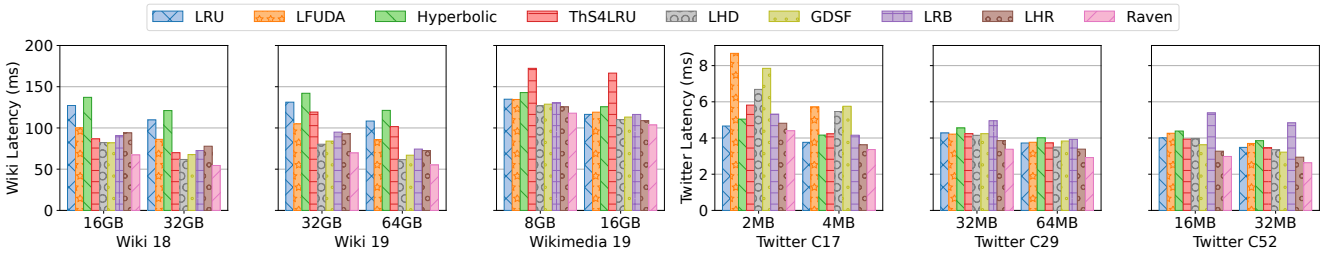
5.2.1 ***Hit Ratios.*** Fig. 9 shows the object hit ratios (OHR) and byte hit ratios (BHR) for each eviction algorithm with different cache sizes using the six traces. Raven consistently outperforms the best state-of-the-art algorithms. Across all experiment settings, Raven improves OHR by 2.6%-7.3% with an average of 4.0%, and improves BHR by 0.5%-7% with an average 3.7%. Note that the improvement of BHR on Wiki 2018 and Wiki 2019 traces are not significant compared to LRB. But, such small BHR improvement help reduce WAN traffic by 4.5% compared to LRB, as shown later. More hit ratio results over 5 cache sizes and the full comparison to the 14 baselines can be found in Appendix H.

To better understand the results, we zoom into the caching performance of different algorithms on different traces. In general, heuristic algorithms perform well on certain type of workload, but

[7]For fairness, we remove the admission control of LHR and use its eviction algorithm, since all other SOTAs and Raven don't do admission control. Appendix G compares Raven to SOTA admission algorithms, *i.e.,* AdaptSize [11] and the original LHR. The results show that Raven without admission control improves OHR up to 4.8% and BHR up to 12.1%, compared to the best performing admission algorithm.



(a) Traffic



(b) Latency

Figure 10: Simulated traffic and latency on production traces with different cache sizes. Raven consistently outperforms all SOTAs in all traces and cache sizes combinations. Small improvements in hit ratios correspond to a high improvement in traffic and latency reduction.

may perform poorly on other types of workloads. For example, GDSF achieves high OHR on Wikipedia traces, but low OHR on Twitter cluster 17 trace. LRU obtains high BHR on Twitter traces, but low BHR on Wikipedia traces. On the Wikipedia traces, compared to existing learning-based algorithms, namely LHR and LRB, Raven improves OHR by 10% and BHR by 1%. The difference between OHR and BHR improvements on the Wikipedia traces is due to the distributions of object requests and requested bytes, as shown in Fig. 18 (Appendix F). Overall, most of the requests in the Wikipedia traces are from objects with frequency greater than 200 (*i.e.,* relatively popular objects). However, most of the requested bytes are from objects with frequency less than 200 (*i.e.,* unpopular objects). Since modeling unpopular objects is more challenging than modeling popular objects, the BHR improvement on the Wikipedia traces is less than the OHR improvement.

5.2.2 ***Traffic, Latency, and Throughput.*** Fig. 10a shows the wide-area network traffic and the database read traffic for each algorithm. Raven robustly outperforms the best state-of-the-art algorithms (SOTAs). Overall, it reduces the backend traffic between the caching server and the origin server by 2.6%-18.8% with an average of 10.4%. Note that Raven simultaneously achieves the largest hit ratio and the least traffic whereas none of the SOTAs does so. For example, while LRB achieves the highest byte hit ratio and least traffic among SOTAs on CDN traces, it exhibits high hit ratios, but large database traffic on Twitter traces.

Fig. 10b shows the average latency for each algorithm. Compared to the SOTAs, Raven reduces the average latency by 4.7%-17.9% with an average of 10.1%. These results show that although the ML inference time of Raven (*i.e.,* 50 μs) is larger than SOTAs, the hit ratio improvement of Raven dramatically reduces latency, as the backend fetch time is significantly greater than the ML inference time. Table 2 summarizes the average throughput of LRU (the default algorithm in production systems), LRB, LHR, and Raven. We can notice that Raven consistently improves throughput over SOTAs across all traces. The memory overhead and running time of the machine learning based algorithms are shown in §6.1.1.

Table 2: Simulated average throughput on a cache of 128GB, 128GB, 16GB, 4MB, 64MB, and 32 MB for Wiki 18, Wiki 19, Wikimedia 19, Twitter 17, Twitter 29, and Twitter 52 respectively.

	Raven	LHR	LRB	LRU
wiki 18 (Gbps)	6.46	5.91	5.83	5.32
wiki 19 (Gbps)	6.00	5.56	5.55	5.04
wikimedia 19 (Gbps)	3.50	3.33	3.10	3.11
twitter 17 (K Requests/s)	18.09	16.87	14.46	16.23
twitter 29 (K Requests/s)	19.95	17.13	14.78	15.54
twitter 52 (K Requests/s)	21.97	19.62	12.21	16.57

5.3 Raven vs. OPT

We have seen that Raven significantly outperforms SOTAs. Now, we compare Raven with the offline optimal algorithm Belady[8] and an online optimal algorithm HRO [77]. Fig. 11 compares their object hit ratios (OHR) and byte hit ratios (BHR) on all six production traces with different cache sizes. We also include the best performing SOTA on each trace and cache size. We find that Raven indeed is

[8]Here we refer to both Belady and Belady-Size as Belady.

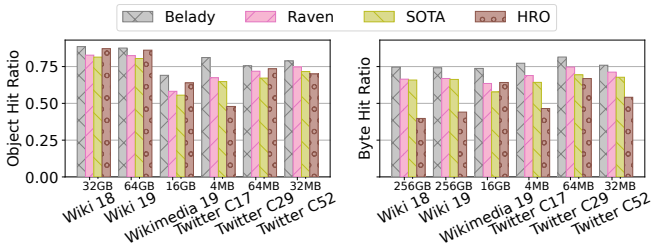


Figure 11: Raven vs OPT on hit ratios.

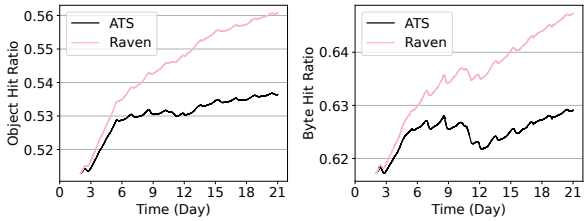


Figure 12: Raven vs unmodified ATS.

closer to Belady. Raven reduces the OHR gap between SOTAs and Belady by 37.2% on average, and reduces the BHR gap between SOTAs and Belady by 29.2% on average. One interesting observation is that HRO performs well on certain traces, but poorly on Twitter C17 and C52, which shows that HRO is workload specific. The poor generalizability of HRO is due to its similarity to LFU policy and its simplified assumptions of arrival processes. HRO estimates hazard rates to represent object popularity based on the assumption that request processes are Poisson. The remaining gap between Raven and Belady is due to the distribution estimation error of our ML model and the randomness of realization of objects' arrival processes. We leave the improvements of distribution estimation for future work.

5.4 Performance of ATS with Raven

In this section, we compare Raven prototype and an unmodified Apache Traffic Server (ATS) with respect to hit ratios, WAN traffic, and implementation overhead. The results are measured using Wikimedia 2019 trace with a cache of 32 GB.

Hit Ratios. Fig. 12 compares the object hit ratio and byte hit ratio of Raven and unmodified ATS. Overall, Raven achieves both higher object hit ratio and byte hit ratio than ATS. We can notice that Raven reaches higher OHR and BHR after its neural network is trained in the warm-up stage, and continues to improve hit ratios with a quicker rate than ATS.

Implementation Overhead. Table 3 compares the overhead of Raven against unmodified ATS in terms of throughput, latency, and traffic. We also measure the max throughput, the 90^{th} and 99^{th} percentile latency, and the 95^{th} percentile bandwidth, as these metrics are the basis of some CDN contracts [1, 68]. The OHR and BHR of Raven is 4.5% and 3% greater than ATS. This hit ratio improvement allows Raven to improve the average latency by 9.4% and the 90^{th} percentile latency by 8.9% compared to ATS. As for the 99^{th} percentile latency, Raven and ATS are the same, because the latency to the origin server dominates. With respect to traffic, Raven reduces the average WAN traffic by 9.9% and reduces the 95^{th} percentile traffic by 7.6% over ATS. Lastly, Raven has no measurable throughput overhead.

Table 3: Resource usage for Raven and unmodified ATS in production-speed experiments.

Metrics	**Raven**	**ATS**
P90 Latency (ms)	145.64	160.00
P99 Latency (ms)	220	220
avg. Latency (ms)	97.85	107.97
P95 Traffic (Gbps)	4.67	5.06
avg. Traffic (Gbps)	2.58	2.86
max Throughput (Gbps)	10.46	10.46
avg. Throughput (Gbps)	4.13	3.76

6 DISCUSSION

In this section, we show the memory overhead and running time of the three machine learning based caching algorithms. Then we discuss the methods to amortize and reduce the machine learning overhead of Raven. To study Raven's implications on IT economics, we present some examples of cache cluster costs with the assumption of allocating the simple heuristics algorithm more caching capacity to achieve the same hit ratio as Raven. Besides, we highlight limitations and future directions of arrival process estimation.

6.1 Machine Learning Overhead & Economics

6.1.1 ***Machine Learning Overhead.*** We compare the memory overhead and running time of the three learning-based algorithms: Raven, LHR, and LRB. For each object in cache, the metadata used by Raven to infer its residual time distribution is its history embedding, current age, and object size. Therefore, the metadata memory of each object in cache is 136 bytes on CDN traces, and 72 bytes on Twitter traces according to the different RNN hidden state size settings. The metadata used by LHR and LRB to make an inference for an object is the size of their manually defined features which takes 84 bytes and 176 bytes, respectively. Overall, Raven's metadata memory overhead is less than LRB, but greater than LHR.

We further summarize the average running time of cache lookup, eviction, and training ML models. Across all experiments, the average lookup time of the three algorithm is about 50 ns and is negligible. The average eviction time of LRB, LHR, and Raven is about 3 μs, 6 μs, and 50 μs, respectively. The inference of neural networks used by Raven is slightly slower than gradient boost machines used by LRB and LHR. Nevertheless, all these three algorithms are highly efficient in terms of prediction time. As for ML training, through a trace-based simulation, the total training time of LRB, LHR, and Raven is 14 hours, 110 hours, and 26 hours, respectively. The total training time of Raven is less than LHR, but greater than LRB. Although the gradient boost machine adopted by LRB and LHR is more lightweight than the neural network utilized by Raven, LRB and LHR need to constantly train their ML models to attain good performance. In contrast, Raven is highly generalizable, and trains the neural network way less frequently, as it depends on deep learning to learn general distributions. Moreover, due to the generalizability of deep learning, the neural network of Raven

Table 4: Simplified estimation of AWS VM cost for caching clusters which use Raven or LRU to achieve the same hit ratio.

	In-memory Cluster [34]		CDN Cluster with EBS [33]		CDN Cluster with SSD [35]	
	Raven	**LRU**	**Raven**	**LRU**	**Raven**	**LRU**
RAM Config	32 GB	128 GB	NA	NA	NA	NA
Disk Config	NA	NA	12.8 TB	25.6 TB	12.8 TB	25.6 TB
VM Type	t4g.micro	(t4g.small, t4g.medium)	t3.medium	t3.medium	g4dn.2xlarge	g4dn.2xlarge
# VM	64	(41, 23)	100	100	57	114
GPU Server	g4ad.xlarge	NA	g4ad.xlarge	NA	g4ad.xlarge	NA
# GPU	1	NA	1	NA	1	NA
Monthly Price	$1,240	$2,631	$6,225	$7,872	$54,322	$108,099

can be trained on a dedicated server, and then applied to 20s or 1000s of cache servers, since the size of production cache clusters typically ranges from 20 to thousands instances [78]. Therefore, the computation overhead and cost of training neural networks can be amortized over multiple cache servers. Whereas, methods using "classical" machine learning, such as LRB and LHR, need to be trained on a per-trace/server basis.

In addition, the training process of Raven can be optimized to further reduce the cost. For example, the RNN module can be replaced with the SRU [47, 48], which is an efficient implementation of the recurrent neural units, and can reduce 28.1% of the training time without performance reduction based on our empirical results. We leave the module optimization and fine-tuning for future work. Finally, the training cost can be further reduced by sharing the GPU server among multiple clusters and minimizing the retraining frequency. Currently, Raven retrains its neural network every training window, which can be reduced by retraining only when request patterns change significantly between two consecutive windows. However, this could degrade the model's accuracy due to errors of pattern change detection.

6.1.2 ***Impacts on IT Economics.*** To study deep learning's implications on IT economics, we use AWS VMs as an example to estimate the cost of in-memory and CDN cache clusters, which use Raven or LRU to achieve the same hit ratio, in a simplified manner. The cost is estimated for simplified clusters which are biased towards using many smaller nodes to limit the blast radius of node failures, whereas real deployments of production cluster are far more complicated. However, our cost comparisons are still valid, as we are interested in comparing the different caching policies under the same settings of these simplified clusters. Based on the evaluation results in §5.2 and Appendix H, LRU needs 4× the cache size as Raven to achieve the same object hit ratio on the in-memory traces, and needs 2× the cache size as Raven to achieve the same byte hit ratio on the CDN traces. We assume the servers of in-memory cache clusters are AWS ElastiCache [34] instances, the servers of CDN cache clusters are Elastic Compute Cloud (EC2) instances in AWS Wavelength zone (*i.e.,* 5G edge infrastructure) [35], and the GPU server is an EC2 instance in the AWS Region zone [33]. The number of VM/instances for each cluster is determined to meet its allocated cache size configuration, as well as to have the same number of CPU cores in the two clusters[9]. For Raven's cache cluster consisting of multiple instances, the additional GPU cost used for training its neural network gets amortized, as Raven requires a smaller cache size to achieve the same hit ratio, hence reducing the overall cluster expenses. As shown in Table 4, Raven reduces 52.9% of the cost for the in-memory cache cluster, 20.9% of the cost for the CDN cache cluster which uses Elastic Block Store (EBS) to cache contents, and 49.7% of the cost for the CDN cache cluster which uses SSD to cache contents, compared to the corresponding clusters using LRU with larger capacity to achieve the same hit ratio.

[9]CDN cluster with SSD is an exception, because AWS does not support SSD size customization for this type of instance.

6.2 Improving Arrival Processes Estimation

Raven assumes that the collected training data characterizes future request arrival processes relatively well. If the next-arrival time distributions of new objects have significant differences, it will be challenging for Raven to accurately estimate their arrival processes. Besides, as the cache capacity increases to hold more objects, differentiating between unpopular objects becomes more important and also challenging for Raven, as estimating arrival processes through a few samples is difficult. We leave the improvement of distribution estimation for future work. We also want to point out that continual learning [62, 67] and meta learning [76] are promising ways to overcome these two challenges.

7 CONCLUSION

Raven is a Belady-guided, learning-based caching framework for both in-memory and content caching. Compared to state-of-the-art caching algorithms, Raven is more general, stable, and explainable. These advantages stem from the key design choices of Raven which explicitly account for the stochastic and nonstationary nature of object arrival processes by learning the (unknown) distributions of object next-request arrival times and estimating the rank order statistics of objects in the cache for decision-making. Evaluation on production CDN and in-memory traces demonstrates that Raven can adapt to different workloads and consistently outperforms the state-of-the-art caching policies in terms of object and byte hit ratios, traffic, and latency reductions.

ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their valuable feedback and our shepherd Nikolaos Laoutaris for his insightful comments and guidance. This research was in part supported by NSF under Grants CNS-1814322, CNS-1836772, CNS-1901103, CCF-2123987 and CNS-2106771.

REFERENCES

[1] Micah Adler, Ramesh K Sitaraman, and Harish Venkataramani. 2011. Algorithms for optimizing the bandwidth cost of content delivery. *Computer Networks* 55, 18 (2011), 4007–4020.

[2] Mehmet Altinel, Christof Bornhoevd, Chandrasekaran Mohan, Mir Hamid Pirahesh, Berthold Reinwald, and Saileshwar Krishnamurthy. 2008. System and method for adaptive database caching. US Patent 7,395,258.

[3] Antonios Antoniadis, Christian Coester, Marek Elias, Adam Polak, and Bertrand Simon. 2020. Online metric algorithms with untrusted predictions. In *International Conference on Machine Learning*. PMLR, 345–355.

[4] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review* 27, 4 (2000), 3–11.

[5] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/fast-04/car-clock-adaptive-replacement

[6] Sorav Bansal and Dharmendra S Modha. 2004. CAR: Clock with Adaptive Replacement.. In *FAST*, Vol. 4. 187–200.

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. {LHD}: Improving cache hit rate by maximizing hit density. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 389–403.

[8] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[9] Daniel S Berger. 2018. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 134–140.

[10] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 1–38.

[11] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. 2017. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 483–498.

[12] Christopher M Bishop. 1994. Mixture density networks. (1994).

[13] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 499–511.

[14] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[15] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking {RocksDB}{Key-Value} Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[16] Livia Elena Chatzieleftheriou, Merkouris Karaliopoulos, and Iordanis Koutsopoulos. 2017. Caching-aware recommendations: Nudging user preferences towards better caching performance. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[17] Jakub Chłędowski, Adam Polak, Bartosz Szabucki, and Konrad Tomasz Żołna. 2021. Robust learning-augmented caching: An experimental study. In *International Conference on Machine Learning*. PMLR, 1920–1930.

[18] Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C Courville, and Yoshua Bengio. 2015. A recurrent latent variable model for sequential data. *Advances in neural information processing systems* 28 (2015).

[19] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. 2017. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 321–334.

[20] CitiBike. 2022. Citi bike trip histories. https://ride.citibikenyc.com/system-data

[21] Ronán Conroy. 2015. Sample size A rough guide. *Retreived from http://www.beaumontethics. ie/docs/application/samplesizecalculation. pdf* (2015).

[22] Pytorch Contributors. 2022. Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/pytorch

[23] Renato Costa and Jose Pazos. 2017. *Mlcache: A multi-armed bandit policy for an operating system page cache*. Technical Report. Technical report, University of British Columbia.

[24] Ruizhi Deng, Bo Chang, Marcus A Brubaker, Greg Mori, and Andreas Lehrmann. 2020. Modeling continuous stochastic processes with dynamic normalizing flows. *Advances in Neural Information Processing Systems* 33 (2020), 7805–7815.

[25] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. 2016. Recurrent marked temporal point processes: Embedding event history to vector. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1555–1564.

[26] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage* 13, 4, Article 35 (Nov. 2017), 31 pages. https://doi.org/10.1145/3149371

[27] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/2038916.2038939

[28] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. 1991. Competitive paging algorithms. *Journal of Algorithms* 12, 4 (1991), 685–699.

[29] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 124 (2004).

[30] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29, 5 (2001), 1189–1232.

[31] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 57–69. https://www.usenix.org/conference/atc15/technical-session/presentation/hu

[32] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 167–181.

[33] Amazon Inc. 2022. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/

[34] Amazon Inc. 2022. Amazon ElastiCache pricing. https://aws.amazon.com/elasticache/pricing/

[35] Amazon Inc. 2022. AWS Wavelength Pricing. https://aws.amazon.com/wavelength/pricing/

[36] Mohammad Rafiqul Islam. 2018. Sample size and its role in Central Limit Theorem (CLT). *Computational and Applied Mathematics Journal* 4, 1 (2018), 1–7.

[37] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 78–89.

[38] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 78–89. https://doi.org/10.1109/ISCA.2016.17

[39] Akanksha Jain and Calvin Lin. 2018. Rethinking belady's algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 110–123.

[40] Akanksha Jain and Calvin Lin. 2018. Rethinking Belady's Algorithm to Accommodate Prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 110–123. https://doi.org/10.1109/ISCA.2018.00020

[41] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. 2020. RL-Cache: Learning-Based Cache Admission for Content Delivery. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2372–2385. https://doi.org/10.1109/JSAC.2020.3000415

[42] Elias Koutsoupias and Christos H Papadimitriou. 2000. Beyond competitive analysis. *SIAM J. Comput.* 30, 1 (2000), 300–317.

[43] K Kylkoski and J Virtamo. 1998. Cache replacement algorithms for the renewal arrival model. In *Fourteenth Nordic Teletraffic Seminar, NTS*, Vol. 14. 139–148.

[44] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. 2017. Harvesting Randomness to Optimize Distributed Systems. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (Palo Alto, CA, USA) *(HotNets-XVI)*. Association for Computing Machinery, New York, NY, USA, 178–184. https://doi.org/10.1145/3152434.3152435

[45] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 1999. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 134–143.

[46] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers* 50, 12 (2001), 1352–1361.

[47] Tao Lei. 2021. When attention meets fast recurrence: Training language models with reduced compute. *arXiv preprint arXiv:2102.12459* (2021).

[48] Tao Lei, Yu Zhang, Sida I Wang, Hui Dai, and Yoav Artzi. 2017. Simple recurrent units for highly parallelizable recurrence. *arXiv preprint arXiv:1709.02755* (2017).

[49] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. 2016. Trend-aware video caching through online learning. *IEEE Transactions on Multimedia* 18, 12 (2016), 2503–2516.

[50] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. PMLR, 6237–6247.

[51] Redis Ltd. 2021. Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. https://redis.io/

[52] Thodoris Lykouris and Sergei Vassilvtiskii. 2018. Competitive Caching with Machine Learned Advice. In *International Conference on Machine Learning*. PMLR, 3296–3305.

[53] Bruce M. Maggs and Ramesh K. Sitaraman. 2015. Algorithmic Nuggets in Content Delivery. *SIGCOMM Comput. Commun. Rev.* 45, 3 (July 2015), 52–66. https://doi.org/10.1145/2805789.2805800

[54] Osama Makansi, Eddy Ilg, Ozgun Cicek, and Thomas Brox. 2019. Overcoming limitations of mixture density networks: A sampling and fitting framework for multimodal future prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7144–7153.

[55] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *Fast*, Vol. 3. 115–130.

[56] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[57] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: A Replacement Algorithm for Flash Memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (Seoul, Korea) *(CASES '06)*. Association for Computing Machinery, New York, NY, USA, 234–241. https://doi.org/10.1145/1176760.1176789

[58] Georgios S Paschos, Apostolos Destounis, Luigi Vigneri, and George Iosifidis. 2019. Learning to cache with no regrets. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 235–243.

[59] Konstantinos Psounis and Balaji Prabhakar. 2002. Efficient randomized web-cache replacement schemes using samples from past eviction times. *IEEE/ACM transactions on networking* 10, 4 (2002), 441–454.

[60] Kaushik Rajan and Govindarajan Ramaswamy. 2007. Emulating optimal replacement with a shepherd cache. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 445–454.

[61] Kaushik Rajan and Govindarajan Ramaswamy. 2007. Emulating Optimal Replacement with a Shepherd Cache. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 445–454. https://doi.org/10.1109/MICRO.2007.25

[62] Dushyant Rao, Francesco Visin, Andrei Rusu, Razvan Pascanu, Yee Whye Teh, and Raia Hadsell. 2019. Continual unsupervised representation learning. *Advances in Neural Information Processing Systems* 32 (2019).

[63] Dhruv Rohatgi. 2020. Near-optimal bounds for online caching with machine learned advice. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1834–1845.

[64] Alireza Sadeghi, Fatemeh Sheikholeslami, and Georgios B Giannakis. 2017. Optimal and scalable caching for 5G using reinforcement learning of space-time popularities. *IEEE Journal of Selected Topics in Signal Processing* 12, 1 (2017), 180–190.

[65] Ketan Shah, Anirban Mitra, and Dhruv Matani. 2010. An O (1) algorithm for implementing the LFU cache eviction scheme. *no* 1 (2010), 1–8.

[66] Oleksandr Shchur, Marin Biloš, and Stephan Günnemann. 2019. Intensity-Free Learning of Temporal Point Processes. In *International Conference on Learning Representations*.

[67] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. 2017. Continual learning with deep generative replay. *Advances in neural information processing systems* 30 (2017).

[68] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544.

[69] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. 2017. Popularity prediction of facebook videos for higher quality streaming. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 111–123.

[70] Stefano Traverso, Mohamed Ahmed, Michele Garetto, Paolo Giaccone, Emilio Leonardi, and Saverio Niccolini. 2013. Temporal locality in today's content caching: why it matters and how to model it. *ACM SIGCOMM Computer Communication Review* 43, 5 (2013), 5–12.

[71] Stefano Traverso, Mohamed Ahmed, Michele Garetto, Paolo Giaccone, Emilio Leonardi, and Saverio Niccolini. 2015. Unravelling the impact of temporal and geographical locality in content caching systems. *IEEE Transactions on Multimedia* 17, 10 (2015), 1839–1854.

[72] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[73] Webcachesim2. 2021. A simulator for CDN caching and web caching policies. https://github.com/sunnyszy/lrb

[74] Alexander Wei. 2020. Better and Simpler Learning-Augmented Online Caching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[75] Wikitech. 2019. This dataset is a restricted public snapshot of the wmf.webrequest table intended for caching research. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching

[76] Yujia Xie, Haoming Jiang, Feng Liu, Tuo Zhao, and Hongyuan Zha. 2019. Meta learning with relational information for short sequences. *Advances in neural information processing systems* 32 (2019).

[77] Gang Yan, Jian Li, and Don Towsley. 2021. Learning from optimal caching for content delivery. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 344–358.

[78] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 191–208.

[79] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. 2018. A deep reinforcement learning-based framework for content caching. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 1–6.

A PROOF OF PRIORITY SCORE EQUATION 1

Proof. Priority score p^j of an object O_j is defined as the probability that the object will arrive farthest, *i.e.,* the residual time R^j of the object is greater than the residual time of any other objects in cache:

$$
\begin{aligned}
p^j &= Pr\{R^j > \max_k R^k, k \neq j\} \\
&= Pr(R^j > R^1, R^j > R^2, \cdots, R^j > R^k, \cdots, R^j > R^C) \\
&(k \neq j, C \text{ is cache size}) \\
&= \int_0^\infty \cdots \int_0^\infty p_{R^1 R^2 \cdots R^C}(r^1, r^2, \cdots, r^C) dr^1 dr^2 \cdots dr^C \\
&(R^j > R^k, k \neq j) \\
&= \int_0^\infty \cdots \int_0^\infty \Big[p_{R^1 \cdots R^{C-1}}(r^1, \cdots, r^{C-1}) dr^1 \cdots dr^{C-1} \\
&\qquad \int_0^{r^j} p_{R^C}(r^C) dr^C \Big] \\
&(R^j > R^k, k \neq j, C, \text{ independence assumption}) \\
&= \int_0^\infty \cdots \int_0^\infty p_{R^1 R^2 \cdots R^{C-1}}(r^1, r^2, \cdots, r^{C-1}) \\
&\qquad Pr(R^C \leq r^j | \tau^C > a^C) dr^1 dr^2 \cdots dr^{C-1} \\
&(R^j > R^k, k \neq j, C) \\
&\vdots \\
&= \int_0^\infty p_{R^j}(r^j) \prod_{k \neq j} Pr(R^k \leq r^j | \tau^k > a^k) dr^j \\
&= \int_0^\infty p_{R^j}(t) \prod_{k \neq j} Pr(R^k \leq t | \tau^k > a^k) dt \\
&= \int_0^\infty p_{R^j}(t) \prod_{k \neq j} F_{R^k}(t) dt
\end{aligned}
$$

where

$$
\begin{aligned}
F_R(t) = Pr(R \leq t | \tau > a) &= \frac{Pr(R \leq t, \tau > a)}{Pr(\tau > a)} \\
&= \frac{Pr(\tau - a \leq t, \tau > a)}{Pr(\tau > a)} \qquad (R = \tau - a) \\
&= \frac{Pr(a < \tau \leq t + a)}{Pr(\tau > a)} \\
&= \frac{F_\tau(t + a) - F_\tau(a)}{1 - F_\tau(a)}
\end{aligned}
$$

□

B RAVEN VS. PREDICTIVEMARKER ON CITI DATASET

We compare Raven with PredictiveMarker [52] on the Citi dataset that is used in [52]. The Citi dataset comes from a bike sharing platform operating in New York City. For each month of 2017, the first 25,000 bike trips are extracted and a request in each trace corresponds to the starting station of a trip. The source code of PredictiveMarker and processed Citi dataset are available at [3]. The raw Citi dataset is publicly available at [20].

Table 5: Competitive ratios and average miss ratios on the Citi dataset.

Metric	LRU	PredictiveMarker	Raven
Competitive ratio	1.971	1.942	**1.799**
Average Miss ratio	0.665	0.658	**0.616**

Table 6: Statistics of Rank Order Errors on synthetic datasets.

Trace	Policy	Mean	Median	P90	Variance
Pareto	Parrot	40.54	37	81	26.87
	LRB	32.20	25	75	26.86
	LHR	31.11	24	73	26.58
	Raven	**28.44**	**20**	**70**	**26.13**
Poisson	Parrot	23.64	14	63	24.41
	LRB	17.52	6	**57**	24.12
	LHR	22.77	12	65	25.57
	Raven	**17.34**	**5**	**57**	**24.05**
Uniform	Parrot	16.01	6	49	21.70
	LRB	9.38	**1**	34	18.60
	LHR	13.07	**1**	50	22.74
	Raven	**7.92**	**1**	**31**	**18.39**

The Citi dataset used for evaluation consists of 12 traces, each of 25,000 length. We use the first 15,000 requests to train the Mixture Density Network of Raven and report evaluation results on the remaining 10,000 requests. Same as [52], we set the cache size to 100. Table 5 shows the competitive ratios and average miss ratios over the 12 bike trip traces. Raven outperforms PredictiveMarker by 7.4% in terms of competitive ratio and by 6.4% in terms of average miss ratio.

C EXPERIMENT ON SYNTHETIC TRACES

C.1 Experiment Setup

We evaluate Raven on three synthetic traces in an idealized setting. Each trace contains 10M requests of 1000 objects whose popularities follow a Zipf distribution with $\alpha = 0.8$. The inter-arrival time distributions of objects in the three traces are Poisson, Uniform, and Pareto distributions, respectively. We further assign sizes sampled from a uniform distribution $U(10, 1600)$ to the objects based upon the independence assumption of object size [10]. We use the first 5M requests to train our machine learning model to learn the inter-arrival time distributions, and evaluate the caching performance on the second 5M requests. We compare Raven with state-of-the-art heuristics algorithms (*e.g.,* Hyperbolic [13], GDSF, LHD [7], LFUDA, and LRU) and learning-based algorithms (*e.g.,* LRB [68], LHR [77], PredictiveMarker [52], and Parrot [50]).

C.2 Results on Synthetic Traces with Identical Object Size

Fig. 13 shows the object hit ratios with various cache sizes on the three synthetic traces mentioned in §3.5. The results show that

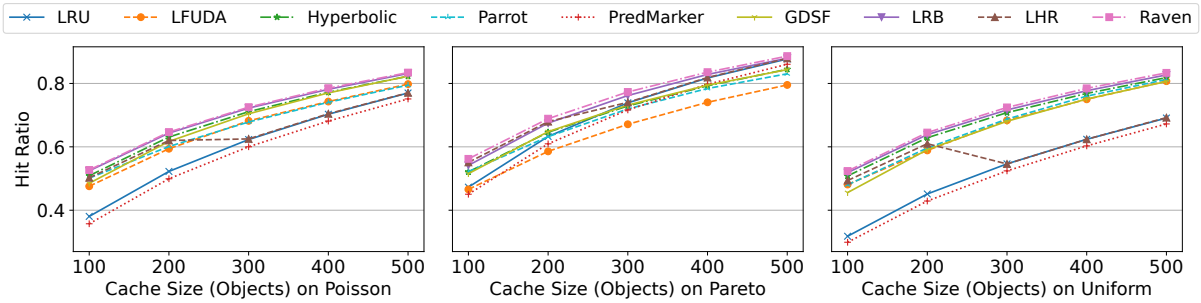
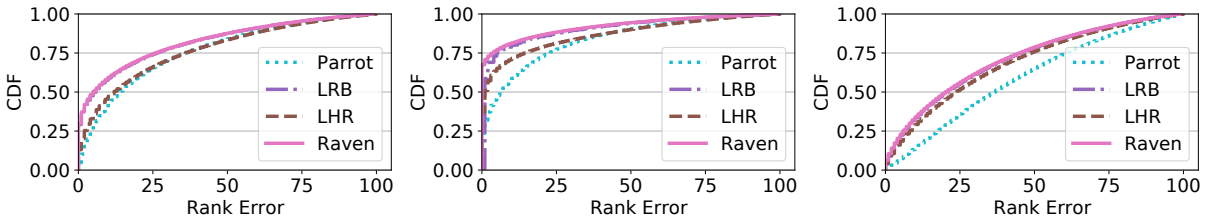


Figure 13: Object hit ratios on the three synthetic traces with identical object size.



(a) Poisson Trace **(b) Uniform Trace** **(c) Pareto Trace**

Figure 14: PDF of rank-order errors on the three synthetic traces with cache size configured to hold 100 object entries.

Raven achieves best performance across various traces and different cache size settings.

Fig. 14 shows the rank-order errors of Parrot, LRB, LHR, and Raven on the three synthetic traces with the cache size configured to hold 100 object entries. The statistics (*i.e.,* mean, median, 90% percentile, and variance) of the rank-order errors are listed in Table 6. All these results show that the average rank-order errors of Raven across traces are consistently the smallest, which indicates that its predictions of objects' future arrivals are more accurate than LRB, LHR, and Parrot. Besides, Raven has the smallest variance of rank-order errors, which indicates that Raven has more stable prediction performance when making caching decisions, because it explicitly accounts for the stochastic nature of object request arrival processes.

C.3 Results on Synthetic Traces with Variable Object Size

Fig. 15 and Fig. 16 show the object hit ratios and byte hit ratios with various cache size settings on the three synthetic traces with variable object size. The results show that Raven consistently achieves the best OHR and BHR performance across various traces on different cache size settings.

D TRAINING DATA SIZE

Table 7 shows the average object number and request sample number in training datasets for the caching settings used in §5.2. The sample rate of the six production traces on the various caching settings ranges from 0.3% to 73%. Based on the sample rate, the number of request samples in the training data ranges from 0.8 million to 65 millions. Since each object has the same probability to be sampled, the characteristics of request samples in the training data remains similar to the characteristics of the full trace in the training window. Therefore, even after sampling, the training data characterizes the original request patterns in the training window quite well.

Table 7: Average object number and request sample number in training datasets.

		# objects	# samples
Wiki 18	C=32 GB	499 K	16 M
	C=128 GB	2 M	65 M
Wiki 19	C=64 GB	1 M	25 M
	C=128 GB	2 M	51 M
Wikimedia 19	C=8 GB	455 K	3 M
	C=16 GB	910 K	6 M
Twitter C17	C=2 MB	11 K	781 K
	C=4 MB	22 K	2 M
Twitter C29	C=32 MB	186 K	4 M
	C=64 MB	373 K	8 M
Twitter C52	C=16 MB	170 K	5 M
	C=32 MB	339 K	10 M

E ONE-HIT WONDER NUMBERS

Table 8 shows the average one-hit wonder numbers within one million requests for the caching settings used in §5.2. One-hit wonders are cached objects that are accessed once and then cached yet never read again before they are evicted due to their lack of popularity. The number of one-hit wonders of a workload is related to the cache size and also caching policies. From the table, we can see that Belady, as an optimal policy, has the least one-hit wonders

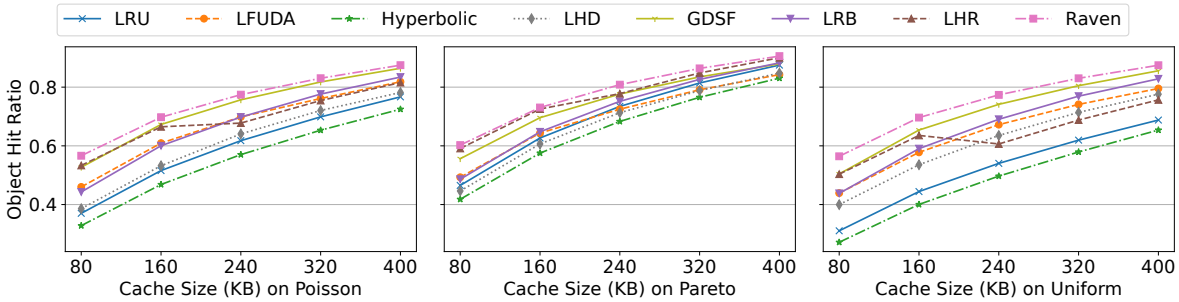


Figure 15: Object hit ratios on the three simulation traces with variable object size.

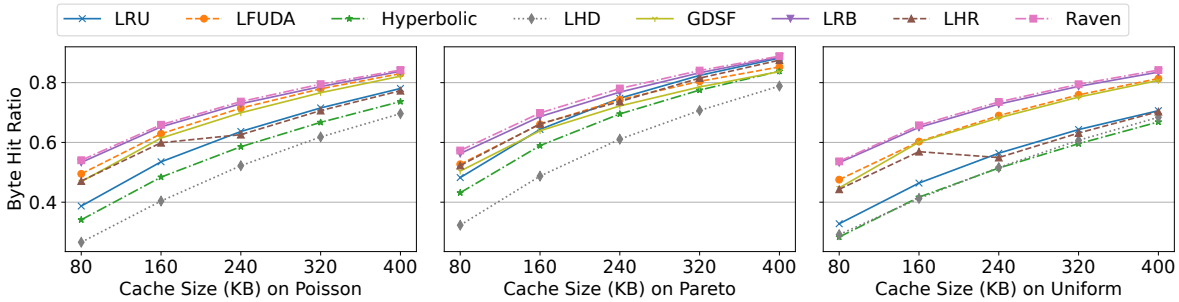


Figure 16: Byte hit ratios on the three simulation traces with variable object size.

Table 8: Average one-hit wonder numbers within one million requests in the six production traces.

		LRU	LFUDA	LRB	LHR	Raven	Belady
Wiki 18	C=32 GB	339 K	292 K	238 K	187 K	138 K	124K
	C=128 GB	172 K	159 K	134 K	77 K	65 K	44 K
Wiki 19	C=64 GB	321 K	286 K	235 K	162 K	140 K	124 K
	C=128 GB	219 K	191 K	157 K	98 K	93 K	72 K
Wikimedia 19	C=8 GB	286 K	319 K	289 K	265 K	240 K	127 K
	C=16 GB	203 K	245 K	227 K	188 K	171 K	70 K
Twitter C17	C=2 MB	290 K	840 K	349 K	367 K	296 K	188 K
	C=4 MB	217 K	524 K	265 K	258 K	238 K	130 K
Twitter C29	C=32 MB	239 K	262 K	246 K	211 K	188 K	158 K
	C=64 MB	195 K	222 K	192 K	171 K	152 K	126 K
Twitter C52	C=16 MB	216 K	272 K	313 K	168 K	150 K	129 K
	C=32 MB	177 K	222 K	216 K	138 K	123 K	105 K

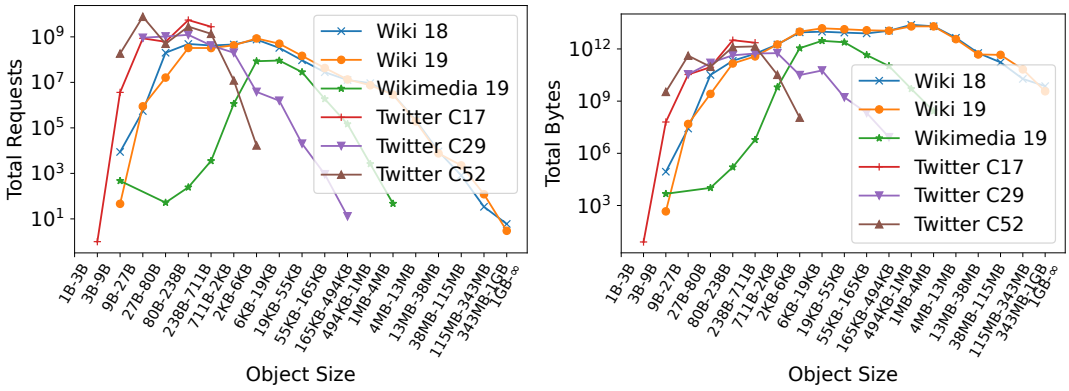
across all workloads and cache size settings. After Belady, Raven, in most cases, has the least one-hit wonders.

F DISTRIBUTION OF OBJECT REQUEST NUMBERS AND REQUESTED BYTES

Fig. 17 and Fig. 18 show the request number and the requested byte distributions over object size and object frequency, respectively. We create log-scale bins of object size and frequency, and add up the total requests and total requested bytes for each bin. This allows us to analyze the characteristics of different traces and reason about what is needed to achieve good OHR and BHR for different traces.

From Fig. 17, we can see that on the Wiki 18 and 19 traces: i) with respect to the total request number, most requests are from objects with sizes in the range (200B, 500KB), and ii) with respect to the total requested bytes, most of the requested bytes are from objects with sizes greater than 200B. The Wikimedia 19 trace is totally different from the other 2 CDN traces. Most requests and requested bytes in Wikimedia 19 trace are from objects with sizes in the range (6KB, 160KB). On Twitter traces, particular size classes have more requests and requested bytes.

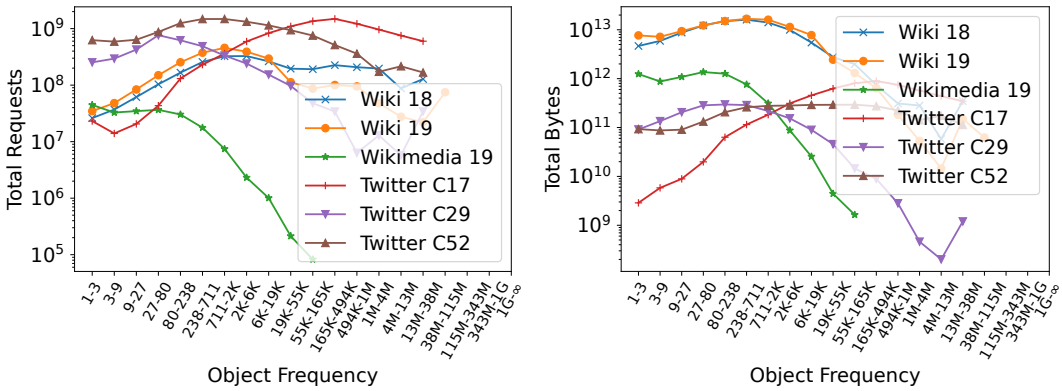
From Fig. 18, we can see that: i) with respect to the total request number, most requests in Wiki 18 are from objects with frequency greater than 200; and most requests in Wiki 19 are from objects with frequency in range (200, 500k). In contrast, ii) objects with frequency less than 200 in Wiki 18 and Wiki 19 contribute to a significant portion of the requested bytes. This indicates that OHR can be optimized by predicting popular objects (frequency > 200), while BHR optimization requires good prediction on unpopular objects as well. The Wikimedia 19 trace is totally different from the other 2



(a) Total requests for each object size range

(b) Total requested bytes for each object size range

Figure 17: The distribution of object requests and requested bytes among different object size ranges.



(a) Total requests for each object frequency range

(b) Total requested bytes for each object frequency range

Figure 18: The distribution of object requests and requested bytes among different object frequency ranges.

CDN traces. Most requests and requested bytes in Wikimedia 19 are from objects with frequency less than 1k. Therefore, the modeling of unpopular objects is important to achieve good OHR and BHR on Wikimedia 19 trace. As for twitter in-memory traces, they have various patterns. Almost all object frequency ranges are important.

G RAVEN VS. ADMISSION ALGORITHMS

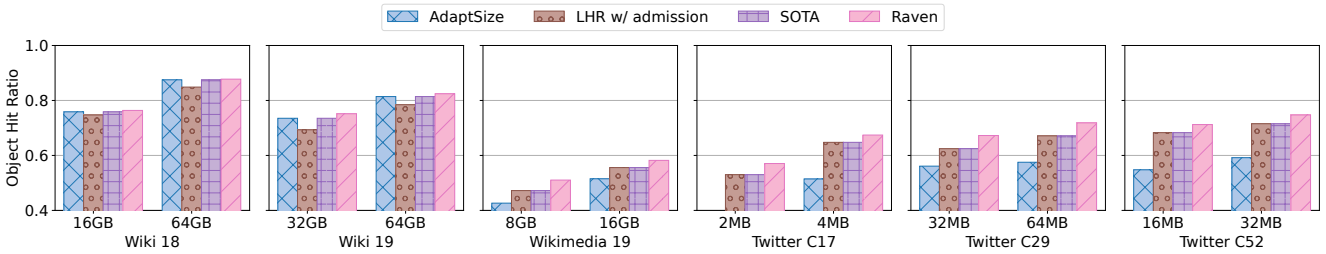
Fig. 19 shows that although Raven doesn't adopt admission algorithms, it still consistently outperforms other methods with admission controls, such as AdaptSize, original LHR, and other baseline methods mentioned in section 5.1.2 (marked as SOTA).

Specifically, from Fig. 19a, we can see that Raven without admission control improves OHR from 0.3% to 4.8% with an average of 2.7% on different datasets and settings. Meanwhile, Fig. 19b shows that Raven improves BHR from 5.9% to 12.1% with an average of 8.2%. The improvement is calculated by absolute value instead of relative value. Those results again validate the effectiveness and robustness of Raven. It is reasonable to expect that Raven can achieve better performance with the help of the admission algorithm and we leave those designs for future work.

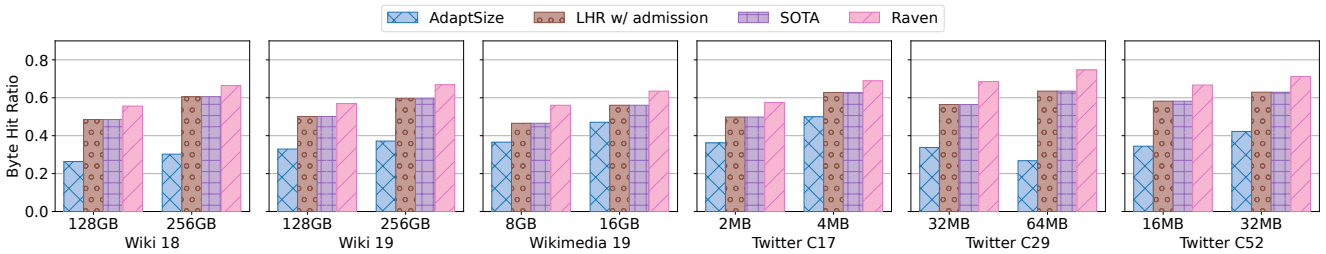
H RESULTS ON MORE CACHE SIZE SETTINGS AND FULL BASELINES

Fig. 20 shows the object hit ratio results on the Twitter cluster 29 trace in 5 cache size settings ranging from 32 MB to 512 MB, and the byte hit ratio results on the Wikimedia 2019 trace in 5 cache size settings ranging from 4 GB to 64 GB. We can see that Raven achieves the best OHR and BHR across all cache settings. The improvement of OHR and BHR is larger on smaller cache sizes. As the cache size increases, the margin decreases and different policies have relatively same performance.

Fig. 21 shows the OHR of the 14 baselines on the Twitter cluster29 trace with a cache size of 32 MB and the BHR of the 14 baselines on the Wikimedia 2019 trace with a cache size of 8 GB. The 14 baselines are 1) heuristics algorithms: LRU, ThS4LRU [32], Random, LFUDA [4], LRUK [56], Hyperbolic [13], GDSF [4], FIFO, ThLRU; 2) learning-based algorithms: LRB [68], UCB [23], LHD [7], LHR [77],



(a) Object Hit Ratio



(b) Byte Hit Ratio

Figure 19: Raven compared to admission algorithms and outperforms them. LHR w/ admission is the original version.

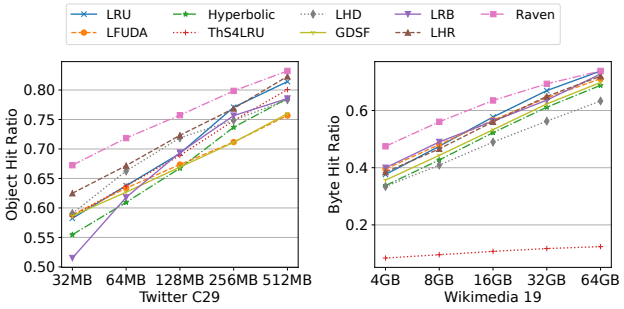


Figure 20: Results of object hit ratio and byte hit ratio on more cache size settings for a subset of workloads.

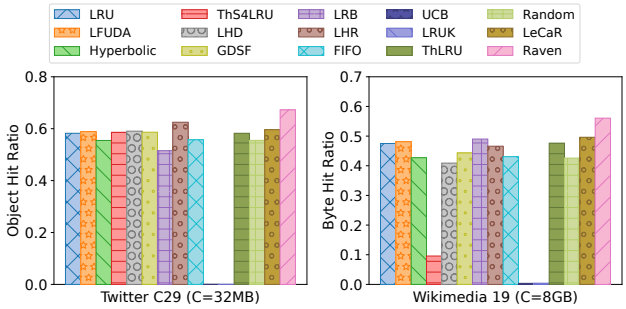


Figure 21: Comparison to 14 baseline algorithms.

LeCaR [72]. Raven performs best among the 14 baseline caching policies.