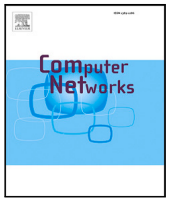
Computer Networks 230 (2023) 109794

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



A delayed eviction caching replacement strategy with unified standard for edge servers

Pengmiao Li [a], Yuchao Zhang [b,*], Huahai Zhang [b], Wendong Wang [a,*], Ke Xu [c], Zhili Zhang [d]

[a] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China
[b] School of Computer Science (National Pilot Software Engineering School), Beijing University of Posts and Telecommunications, Beijing, China
[c] Tsinghua University, Beijing, China
[d] University of Minnesota, Minneapolis, United States of America

ARTICLE INFO

Keywords:
Edge server
Caching replacement
Fetching-time

ABSTRACT

The current explosion of user traffic compels Internet service providers to cache contents at edge servers, so as to reduce the response time of user requests. To deal with such vast amounts of content in edge servers with limited storage, an efficient caching replacement strategy is necessary, consisting of admission and eviction processes. Facing such a large number of user requests and limited storage of edge servers, traditional caching replacement strategies have encountered two major problems. Firstly, they conduct eviction process immediately after cache miss, which ignores the fetching time, during which there may still be multiple access requests to the evicted contents. Secondly, most existing solutions treat admission process and eviction process separately, with different standards, some contents might be fetched and evicted back and forth, seriously affecting user experience. To solve these two problems and improve caching performance, we design two caching modules, *Delayed-Eviction* and *Unified-Standard*, and integrate them together into *Adele* framework. By conducting extensive simulations using real traces, we demonstrate that *Adele* can improve the hit rate by 31% compared with the state-of-the-art solutions.

1. Introduction

With the development of 5G and AI technology, the number of users is increasing exponentially. According to the report [1], the number of worldwide social media users is 5.31 billion, which is more than 67.1% of the world's population. These users upload and view a lot of contents, which introduces high requirements for caching and transmitting on servers. Although these contents can be cached in the origin cloud where there is enough storage space, it costs high network bandwidth to send these contents back to the requesting users, which results in high latency and poor Quality of Service (QoS). To reduce the tremendous pressure on the backbone network & data centers and also provide better QoS with lower latency, Internet content providers usually cache these contents in edge servers nearby users. However, edge servers have limited caching capacity and cannot cache all request contents as well as cloud, which affects the caching performance. Therefore, it is crucial to design an efficient caching replacement strategy for edge servers.

Existing caching replacement strategies have improved dynamic caching performance on storage servers, such as efficient & low-complexity strategies (LRU, LFU, and simple variants [2,3]) and artificial intelligence technology strategies (LeCaR [4], LRB [5], and RL-Belady [6]). These strategies assume that the complete replacement process includes four steps: user request (①), access control (②), eviction (③), and response (④). In reality, the complete caching replacement process includes ①–⑤ as shown in Fig. 1. Therefore, almost all the existing work ignores two problems: ***Fetching-Gap*** and ***Standard-Conflict*** in this process.

Problem 1 (*Fetching-Gap*)**.** Before the eviction stage, there is another transmission step (⑤). This step (⑤) requires some time [7,8], known as fetching-time, which might be greater than 100 ms [9,10], to retrieve the missing content from the cloud. Therefore, it is unwise for existing caching replacement algorithms to choose evicted contents right away when the content sought in the access control step (②) is missing. About 1 million requests [11] for content are sent during the transmission step, some of which are likely to access recently evicted content, decreasing the content hit rate and lengthening user wait times. Such situation is referred to be "*Fetching-Gap*". (see Section 3.2.1)

Problem 2 (*Standard-Conflict*)**.** A complete caching replacement strategy consists of admission and eviction policies, which have been

* Corresponding authors.
E-mail addresses: yczhang@bupt.edu.cn (Y. Zhang), wdwang@bupt.edu.cn (W. Wang).

https://doi.org/10.1016/j.comnet.2023.109794
Received 1 February 2023; Received in revised form 14 March 2023; Accepted 19 April 2023
Available online 25 April 2023

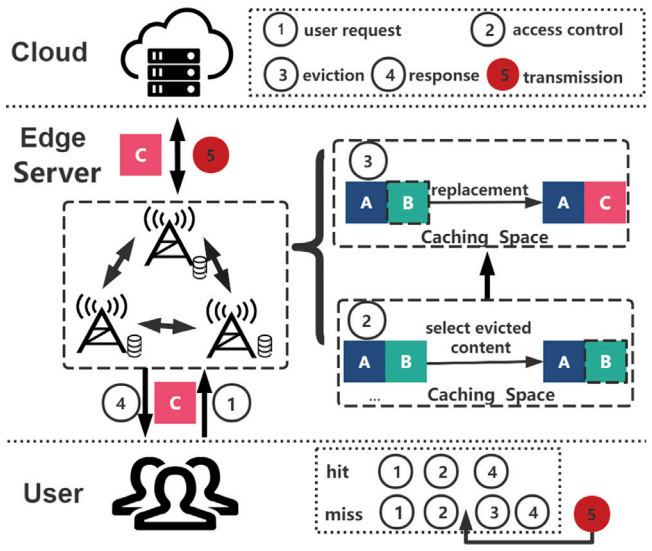


Fig. 1. Caching replacement process.

studied in many scenarios. Although a cache admission policy can be combined with an eviction policy from different scenarios to form the caching replacement strategy, these policies have distinct standards for determining the content value. When making caching decisions, using different standards to admit and remove contents will result in repeated replacement, i.e., content that has been admitted by the admission policy may be immediately deleted by the eviction policy. Such back and forth replacement will inevitably reduce the hit rate and subsequently increase users' waiting time. The reason is that evicted content might be more likely than admitted content to be accessed in the future, leading to missed requests for such content (eviction one). We call such a problem "*Standard-Conflict*". (See Section 3.2.2)

To solve these two problems, we introduce two modules for existing caching replacement strategies. For the *Fetching-Gap* problem, we propose *Delayed-Eviction* module, which will not execute eviction until the fetching is completed, ensuring that the accessed contents during the fetching period are not evicted. For the *Standard-Conflict* problem, we propose *Unified-Standard* module to measure the access probability of the requested content with a new value function as well as contents cached in the edge server. The module can ensure that the high-probability contents in access process will not be evicted in the following eviction process. Then we integrate these two modules together and propose a two-layer caching framework *Adele* using A3C algorithm. We implement *Adele* and several existing algorithms. The results show that the hit rate and the transmission latency of the algorithm with *Adele* integrated is about 31% higher and 37% lower than the state-of-the-art solution, respectively.

The main contributions of this paper are listed as follows:

- First, we disclose two problems *Fetching-Gap* and *Standard-Conflict* of existing caching strategies in edge network. In *Fetching-Gap*, strategies ignore the fetching time, during which there may still be multiple access requests to the evicted contents. In *Standard-Conflict*, under mixed strategies with different standards, some contents with higher access probability than admitted ones may be evicted. These problems significantly affect caching performance.
- Second, to address these two problems, we propose two corresponding modules *Delayed-Eviction* and *Unified-Standard* for existing caching strategies. In addition, we integrate these modules into *Adele*, which is a caching replacement strategy proposed in this paper based on A3C algorithm, to improve the caching performance.
- Third, we conduct a series of experiments with real data from *ChuangCache* Company. By numerical results, we reveal two modules' advantages in improving caching performance. And we demonstrate the proposed solution *Adele* outperforms existing caching replacement strategies in hit rate & byte hit rate and transmission latency.

The remainder of this paper is organized as follows. The current edge caching replacement strategies is summarized in Section 2. Section 3 introduces the background and motivation of this paper. Section 4 presents two modules and the framework of *Adele*. Section 5 demonstrates the setting up of *Adele* prototype and shows extensive experiment results from real data evaluations. Finally, Section 6 concludes this work.

2. Related work

Even small hit rate improvements cause significant speedup, as in the Facebook study there is a 1% improvement of hit rate with 374 μs, there is 35% speedup with 278 μs [12]. In order to improve user's QoS, it is inevitable that a large number of researchers focus on the improvement of hit rate for caching. Typical analysis of caching systems either focuses on content admission, which decides whether to cache the request content (admission policy), or content eviction to decide which content to evict when the cache is full (eviction policy). Table 1 summarizes some of the most relevant works, which have been instrumental in advancing the current research.

2.1. Admission policy

The edge servers have limited caching capacity, so only a tiny amount of content can be stored. Researchers recognize that not all contents should be stored in edge servers as the demand for various types of content varies, depending on its freshness, frequency, size, etc. In the past few years, there has been an abundance of attention given to creating an admission policy by various scholars.

Akamai [20] counted that out of the approximately 400 million files requested on a server cluster over two days, nearly three-quarters were only requested once. The Cache-on-second-hit rule algorithm (*SecondHit*) [13] was implemented due to this feature, caching the content only after it had been asked for twice. Caching admission policies based on request frequency also include *TinyLFU* [14], which determines whether the content is cached in the edge server by the freshness of the requested content. *SecondHit* and *TinyLFU* have improved the hit rate significantly, but they ignored the fact that there was a noticeable difference in the content size. Daniel et al. proposed *AdaptSize* [15] based on a novel Markov cache model, the first adaptive, size-aware cache admission policy for hot contents that achieves a high hit rate, even when object size distributions and request characteristics vary significantly over time. Considering the above mentioned factors about frequency, size, and recency, Vadim et al. proposed a novel algorithm called *RL-Cache* [17] based on model-free reinforcement learning to decide whether or not to cache a requested content in CDNs. In certain situations, apart from these factors that influence the content hit rate, the content publisher's information and the content's subject topic are also considered. Yu Guan et al. proposed a novel content-aware cache admission (*CACA* [16]) policy based on a tree-structure reinforcement learning model for video content edge caching. It determined whether caching of the requested content in the edge server by the video's characteristics, such as its category, author, and length, rather than the request pattern.

These admission policies contribute significantly in improving caching hit rate. The corresponding content eviction policies have been more extensively studied.

Table 1
Some related works.

Year	Strategy	Admission	Eviction	Major attribution	Goal
2015	SecondHit [13]	✓	×	Frequency	Hit rate, Latency
2017	TinyLFU [14]	✓	×	Frequency	Hit rate
2017	AdaptSize [15]	✓	×	Size	Hit rate
2019	CACA [16]	✓	×	Author, Item, Recency	Hit rate
2019	RL-Cache [17]	✓	×	Frequency, Size, Recency	Hit rate
2018	LeCaR [4]	×	✓	Frequency	Hit rate
2018	LHD [18]	×	✓	WorkLoads	Hit rate
2020	LRB [5]	×	✓	Frequency, Size, Recency	Byte hit rate, Latency
2016	PopCaching [19]	✓	✓	Frequency	Hit rate
2020	RL-Belady [6]	✓	✓	Frequency, Size, Recency	Hit rate

2.2. Eviction policy

Numerous scientific eviction policies are widely deployed, such as traditional methods First Input First Output (*FIFO*), Least Recently Used (*LRU*), Least Frequently Used (*LFU*), and their variants [2–4,21–23]. Recently, researchers have proposed many eviction policies in different perspectives, such as location [24], size [18], and others [24–37].

Giuseppe et al. proposed a general framework called *LeCaR* [4], which is based on two fundamental eviction policies: *LRU* and *LFU*, and is used the ML technique of regret minimization to improve the caching performance. Nathan et al. proposed a novel eviction policy for key–value caches, which was named *LHD* [18]. It predicted each content's expected hits-per-space-consumed (hit density) and filtered contents that contribute little to the cache's hit rate to fit different workloads in a lifetime for improving the hit rate. A content eviction policy *LRB* [5], combining machine learning technology to simulate *Belady* algorithm, was proposed to close to the optimal hit rate. *LRB* obtained Belady boundary and good decision ratio as an eviction quality metric, which had enabled it to take a fundamentally new approach that approximated *Belady*'s MIN algorithm to improve the caching byte hit rate and reduce transmission latency.

Although *LRB* and other strategies can improve hit rate to reduce latency, they do not explicitly consider the missing content needs some time to be fetched from origin. The edge server still receives some request information while fetching missed content. These eviction policies ignore the information in choosing the evicted content, which affects the caching performance of the edge server.

2.3. Eviction and admission

A full caching replacement strategy consists of an admission process and an eviction process. Although a complete caching replacement strategy can consist of an admission policy and an eviction policy from different scenarios, it has lower caching performance and poor QoS than the complete caching replacement strategy proposed from the same scenario. Now, few researchers have come up with a complete caching replacement strategy.

Li et al. presented a novel caching replacement method, namely *PopCaching* [19], which learned the popularity of the content and used it to determine which content it should store and which it should evict from the cache. Gang Yan et al. proposed a novel framework *RL-Bélády* [6] that can simultaneously learn both content admission and content eviction for caching in CDNs. It first put forward a lightweight architecture for predicting content next request time. Then it leveraged reinforcement learning (RL) to learn the time-varying content popularities for content admission and developed a simple threshold-based model for content eviction.

Although *PopCaching* and *RL-Bélády* can consider both content admission and content eviction with same factors, but they ignore some information generated while fetching the missing content.

3. Background and motivation

In this section, we first introduce the background of caching replacement in the edge server, shown in Section 3.1. Then, we present the motivation of this paper in Section 3.2. Finally, we analyze some performance metrics to measure the caching replacement strategy in Section 3.3.

3.1. Background

To reduce transmission latency, more edge servers closer to users have been built by equipment service providers (ESPs), including CDNs, base stations, and others. Rather than transmitting data over long distances from the cloud to the users, these edge servers can facilitate shorter-distance transmission from the edge to the users. The transmission time for data can be substantially decreased, thus reducing the amount of time users have to wait. However, these edge servers have a limited storage capacity, so we must pick out some popular contents to cache to maximize edge server performance. Additionally, as user request pattern change, the popular content will follow suit. Therefore, it is necessary to design an dynamic caching replacement strategy for edge caching system.

Fig. 2 shows the edge caching system including **USER**, **EDGE SERVER**, and **CLOUD**. User i can request any content $c_{i,t}$ to the edge server e at time t. The content requested is stored in the cloud o, whether it is or is not cached in the edge server. If edge server caches the requested content $c_{i,t}$, the request state is *hit* and the transmission latency is $l^e_{i,c_{i,t}}$. Otherwise, the state is *miss* and the transmission latency is $l^o_{i,c_{i,t}}$. So, we denote the latency of request content $c_{i,t}$ from user i at time t by equation

$$l_{c_{i,t}} = \begin{cases} l^e_{i,c_{i,t}} & , \quad \text{if state is hit} \\ l^o_{i,c_{i,t}} & , \quad \text{if state is miss.} \end{cases} \tag{1}$$

It is obvious that $l^e_{i,c_{i,t}} < l^o_{i,c_{i,t}}$. To put it briefly, the magnitude of the usual transmission delay is mainly determined by the state of the content requested by the user, and this state reflects the user's level of QoS. In order to enhance QoS, we should design an appropriate caching replacement strategy to maximize the hit rate of the edge server. However, this is not easy due to the following two problems described in Section 3.2.

3.2. Motivation

We present two new problems: *Fetching-Gap* and *Standard-Conflict*, as well as two key observations, outlined in Sections 3.2.1 and 3.2.2, following a comprehensive examination of relevant literature.

3.2.1. Fetching-Gap

When the requested content is miss, edge server will send a fetching request to a remote cloud. The fetching process takes some non-zero transmission time (*fetching-time*) [7,8]. It is worth noting that the key component of the *fetching-time* is the transmission time

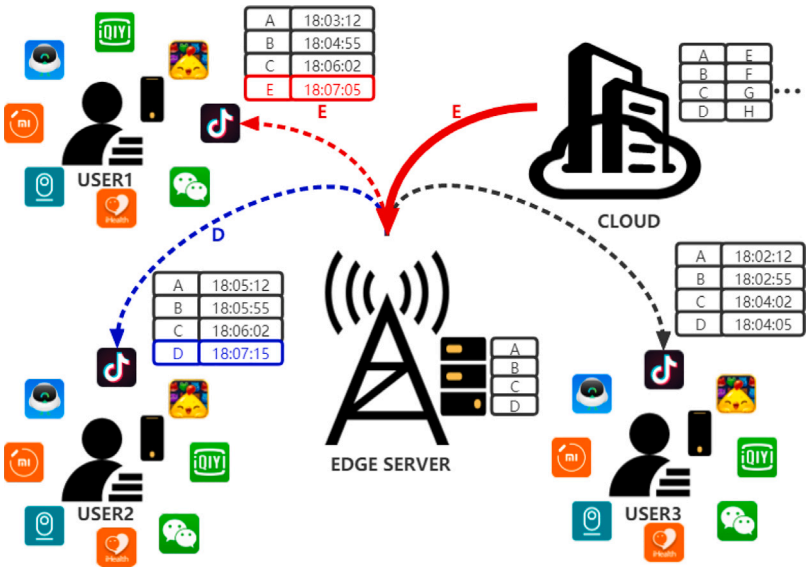


Fig. 2. Edge caching system.

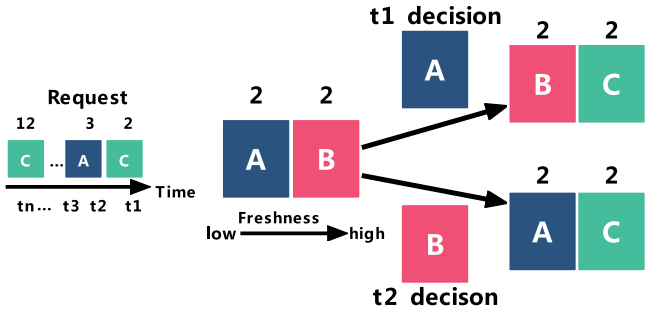


Fig. 3. Caching replacement strategy *LFU* has different evicted contents at different times to make a caching decision. *t1* is the time at which a request for content $\mathbb{C}$ arrives, and *t2* is the time at which a fetching for content $\mathbb{C}$ from cloud arrives.

for the miss content in this paper. Although high concurrency requests in some applications affect *fetching-time*, there are a large number of proven techniques & solutions [38] that are widely used in commercial/industrial distributed systems to reduce the concurrency processing time. In this paper, *fetching-time* is not determined by the number of requested contents. Current caching replacement strategies evict content to make room for the coming fetching content immediately when finding a cache miss. But the evicted content will likely be reaccessed during the *fetching-time*, resulting in another content miss. It is because that strategies lack the information generated during the *fetching-time* to make a caching decision, which may result in the evicted content being requested one during this period.

Problem 1. The edge server fetches the missing content from the cloud, meanwhile, receives many requests by users. The information of requested contents is not considered in the existing cache eviction policies and may affect the caching performance, which is called ***Fetching-Gap***.

The eviction content chosen by the same policy varies at different times. Fig. 3 shows that different contents are chosen as eviction ones by *LFU* at two times. We set the arrival time of missed content $\mathbb{C}$ as the first decision time $t1$. Evicting content $\mathbb{A}$, subsequently leaving the cache space with two contents: $\mathbb{B}$ and $\mathbb{C}$, occurred simultaneously. When the requested content $\mathbb{A}$ arrives in edge server at $t2$, it is *miss*. Otherwise, if the decision time is $t2$ as the second point, content $\mathbb{B}$ is chosen as eviction content, and then contents $\mathbb{A}$ and $\mathbb{C}$ in the cache space. When the edge server receives content $\mathbb{A}$, the latter has a higher hit rate (1>0) than the former.

Inspiration 1. *The delayed caching decision has a higher cache hit rate than the immediate decision because the former gets more information about the content of the request during the fetching-time than the latter. We call it* ***Delayed-Eviction****.*

3.2.2. Standard-Conflict

Although a complete caching replacement strategy can be constructed by blending cache admission and eviction policies from various scenarios, its efficiency could be better and further enhanced. Due to these blending caching replacement strategies, the admission and eviction policies are considered separate entities and include two different standards. Assuming that the future access potential of the requested content and the content in the cache is evaluated by different standards. It means that there may be admission content with a lower real future access probability than the evicted one or non-admission content with a higher real future access probability than any contents cached in the edge server. The situations lead to repeated accesses and evictions of the requested content, affecting the damage hit rate. Therefore, it is necessary to compare the value of access and caching contents with a consistent standard.

Problem 2. Due to loss comparison between the two type contents (request and caching contents) with the same standard, it may cache the low value in edge server, then lead to evict contents frequently. These may affect the caching performance, which is called ***Standard-Conflict***.

Fig. 4 describes the different decision results in two caching replacement strategies. The first one is *SecondHit* (cache admission policy) & *LFU* (cache eviction policy). The other one makes the cache decision based on the total frequency of content requested. At $t1$, the edge server receives the requested content $\mathbb{C}$ and discovers that the state of content $\mathbb{C}$ is missing. For the first strategy, content $\mathbb{C}$ is allowed by *SecondHit* (3 > 2). Then content $\mathbb{A}$ is chosen as eviction one via *LFU*. The caching space has two contents: $\mathbb{C}$ and $\mathbb{B}$. When the next requested content $\mathbb{A}$ arrives in edge server, the hit rate is 0. For the second strategy, content $\mathbb{C}$, as eviction one, is not allowed, by comparing total frequency from all contents, including requested and cached now. The contents cached in storage space are $\mathbb{A}$ and $\mathbb{B}$. The latter has a higher hit rate (1 > 0) than the former when the edge server receives the requested content $\mathbb{A}$.

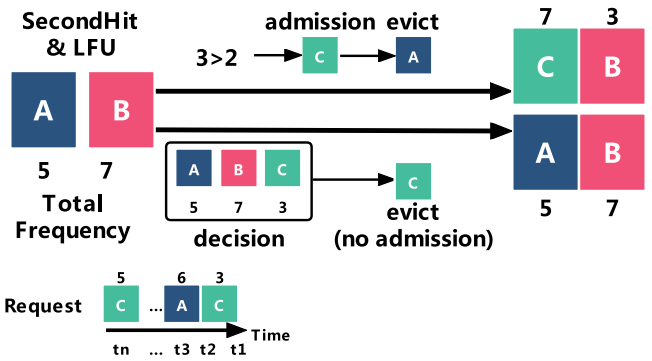


Fig. 4. Different contents are evicted by unified standard and mixed standard. .

Table 2
An example for metrics comparison: HR & TL.

content-size		ℂ-2			𝔸-1			𝔹-3		
request size		$\frac{1}{2}$	1	2	$\frac{1}{4}$	$\frac{1}{2}$	1	1	2	3
HR	*CHR*	1	1	**0**	1	1	1	1	**0**	**0**
	BHR	1	1	**7/8**	1	1	1	1	**15/16**	**19/24**
TL	*CTL*	0	0	2μ	0	0	0	0	2μ	3μ
	BTL	0	0	$1/4\mu$	0	0	0	0	$1/8\mu$	$5/8\mu$

Inspiration 2. *A caching replacement strategy with a unified standard is more beneficial than others with a mixed standard generated from separate cache admission and cache eviction policies from different scenarios. We call it* ***Unified-Standard****.*

To sum up, we adopt two inspirations to address these problems: *Fetching-Gap* and *Standard-Conflict*, by two modules: *Delayed-Eviction* and *Unified-Standard* introduced in Section 4.

3.3. Performance metric

There are many metrics to measure the performance of caching replacement strategies. It includes common Hit Rate (*HR*) metrics (Content Hit Rate (*CHR*) and Byte Hit Rate (*BHR*)) and Transmission Latency (*TL*) metrics (Content Transmission Latency (*CTL*) and Byte Transmission Latency (*BTL*)). Fig. 5 presents the caching state with different content granularity. To find a suitable metric for the current scenario, we make an example to compare these metrics and its *HR* & *TL* with different metrics are shown in Table 2.

The content sizes of 𝔸, 𝔹, and ℂ are 1, 3, and 2 respectively. The cached content sizes of three contents are 1, 19/8, and 7/4. When content ℂ requested sizes are 1/2 and 1, content and Byte are identical whether in two category metrics with *HR* and *TL*. When content 𝔸 with size 3 and content B requested size is 1, the result is the same with above, due to the cache contains sub content requested by the user. However, neither *HR* nor *TL* is the same at other request sizes, such as when content B requested size is 2. *CHR*, and *BHR* are 0, and 15/16 respectively. *CTL*, and *BTL* are 2μ, and $1/8\ \mu$, where μ is present the cost time for fetching data with size 1. It is obvious that the results become more accurate as the granularity of the metrics is refined. Therefore, *BHR* and *BTL* metrics are more accurate than others.

4. Design

In this section, we propose two modules *Delayed-Eviction* and *Unified-Standard* in Sections 4.1 and 4.2. Then, we introduce the framework of caching replacement strategy *Adele* and describe it detailedly in Section 4.3.

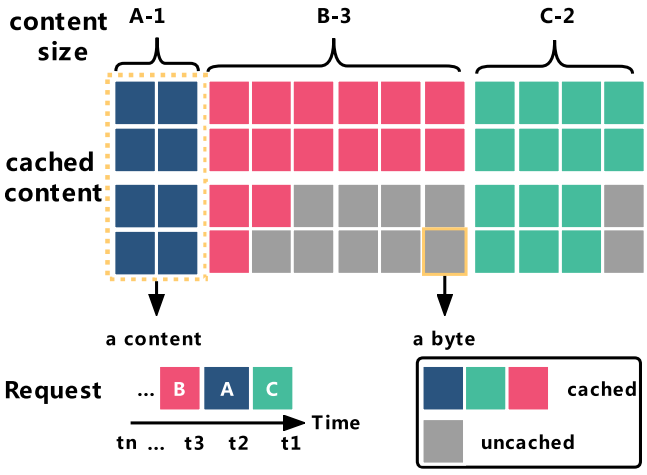


Fig. 5. An example for metrics comparison: caching state.

4.1. Delayed-Eviction

The edge server receives large amounts of request information generated during the *fetching-time*. In order to reduce the eviction of requested content generated during this period, information about these requests should be captured and taken into account by whichever caching replacement strategy is used. While large amounts of information can be captured through *Delayed-Eviction*, the question of when to select evicted content is a key concern. It is because that the selection of cache evicted content also takes some time. If the evicted content is selected too late, it is possible that missing content from the cloud has been fetched but has no cache space for storing it. As a result, it needs to be re-fetched, which increases user waiting time. In another situation, if the evicted content is selected too early, incomplete information will be collected during the fetching time, which may lead to the request for the evicted content. To address this issue, we investigate the computation time of the cache eviction policies and find that most of the computation time is short and completely negligible. Therefore, we set the selected time of cache-evicted content as the arrival time of the first frame for the fetched content returned.

4.2. Unified-Standard

We can effectively ensure that contents cached in edge server reaches a high future access value by setting an *Unified-Standard* for measuring to the content value. This paper's future access value has been established based on three key factors of content: size, frequency, and timeliness. The most intuitive attribution reflecting the value of future visits is the access frequency of historical. It is well known that the more frequently each content is accessed does not mean that it is of higher value for two reasons. The first is pretty obvious: for the same frequency of visits, content with a smaller size is of higher value due to the smaller one cached occupying a tiny space than another. Thus, we considered the single-byte access frequency to reflect the content access value. The other one is that continuous accumulation of the frequency of each content will lead to a high frequency for the retired one, which will be mistaken as the hot one at the moment.

To avoid this phenomenon, we add a sliding window based on a period time δt to count the access frequency for ensuring that the content value is fresh. In summary, the unified standard for measuring the content i value v_i^t in time t is

$$v_i^t = \frac{a \times \tau_i \times e^{\sum_{f=1}^{F_{t-\delta t,t}} \frac{F_{i,f}}{F_{t-\delta t,t}}}}{\tau_a^{t-\delta t,t}}. \tag{2}$$

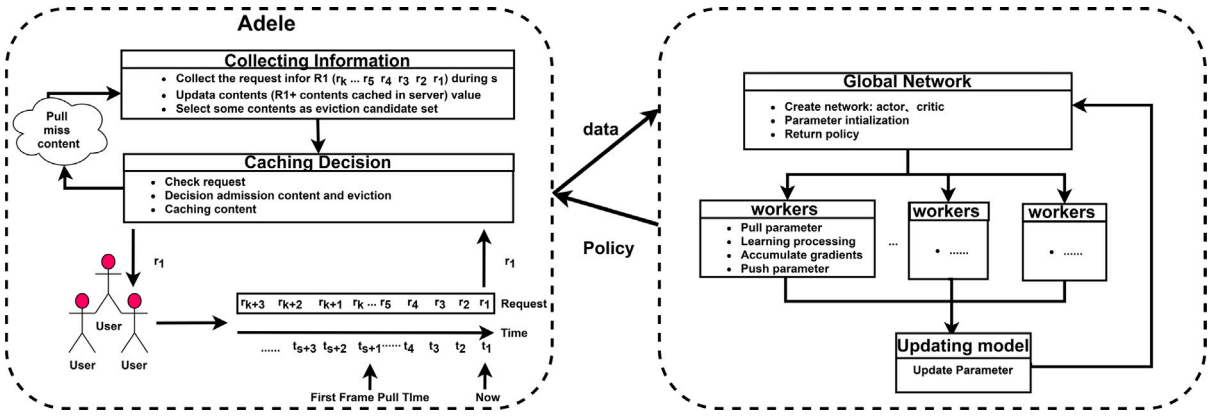


Fig. 6. The framework of *Adele*.

a is a coefficient variation. τ_i presents the size of content i and τ_a^t presents the total size of contents requested during δt to t. $F_{t-\delta t}$ indicates the total number of requests. $F_{i,f}$ presents the access times of content i in the fth request during $t-\delta t$ to t. If the fth request is not content i, $F_{i,f}$ equals 0. In particular, when the volume of requests is small, it also means that users send fewer requests during the fetching time. All content values can be calculated by Eq. (2). The value of contents mainly considered by the request number is very close when the number of requests is small. It is difficult to separate the high-value contents from those cached ones, so it is challenging to further improve the hit rate. As a result, we decide to compute the content value using a straightforward and useful function: the frequency of content requests (within the sliding window δt) divided by the size of the content as $v_i^t = \frac{f_i}{\tau_i}$. High-value content can be easily distinguished from low-value content by this function. Content's value is only determined by its size and frequency; it is unaffected by the number of requests from other contents.

4.3. Framework

The two problems faced in this research are addressed by the *Delayed-Eviction* and *Unified-Standard* modules, which can enhance the performance of caching replacement strategies currently in existence. This paper suggests an unique caching replacement framework *Adele* to more effectively leverage the benefits of both modules. The framework consists of four phases: Initialization, Collection Information, Caching Decision and Updating model, as shown in Fig. 6.

- **Initialization**. In the initialization phase, we adopt LRU as the eviction policy and admit all request contents in the edge server. Then, we initialize the global network and establish the local networks of each worker. It is worth noting that the selection probability of each action is equal at initialization.
- **Collecting Information.** When the length of request sequence recorded by server is enough for two sliding time windows, we collect the request information during the *fetching-time* and append it to historical request information. Then, we divide historical request information into two sets: high-value and low-value. Finally, we choose the low-value set as the eviction candidate set.
- **Caching decision.** Based on the collection information phase, the eviction candidate set obtained as A3C model input contents. Then, *Adele* performs A3C model to make a caching decision, which is choose the eviction content and remove it.
- **Updating model.** To keep the model fresh, the worker updates global network parameters with intervals of T seconds by executing the *push* operation.

If the requested content is hit, the edge server returns it to the user. Else, the edge server executes *Caching decision* to decide whether to cache this content and which contents to evict as Algorithm 1. With intervals of T seconds, *Adele* updates the global network parameter with Algorithm 2. *Adele* is summarized as the pseudo-code shown in Algorithm 3. The detailed design of the cache decision based on A3C algorithm is consist of three parts: eviction candidates set, state–action-reward, and A3C network.

Algorithm 1: Caching decision

Input: missing content information; cache status; caching action selection policy π from the actor in global network;
Output: eviction candidate set ecs

Get M candidate eviction sets $(C_1, C_2, \ldots, C_M)$ *randomly from content low value content, and the total size of each sets needs to be greater than* $size_{miss}$;
Get v_{miss}^t by equation (3);
for $i \in \mathcal{M}$ **do**
  Get v_i^t by equation (3);
end
$s_t = [v_{miss}^t, v_1^t, v_2^t, \ldots, v_m^t]$;
Get a_t *according to policy* $\pi(a|s;\theta)$;
if $a_t = 0$ **then**
  return null;
end
$ecs = C_{a_t}$;
return ecs;

4.3.1. Eviction candidates set

The essence of the caching replacement strategy is to select low-value content as the eviction content by comparing the value of each content cached in edge server. Each comparison of access value for each content needs to be recalculated. However, there may be content with the same value before and after the calculation, affecting the calculation speed and reducing the evicted content's selection efficiency. We additionally find that high-value content, whether or not its value is updated in real-time, has no effect on the selection of content to be evicted. It is because that the high-value content is far more valuable than other content at some time. Therefore, in order to improve the calculation speed and the selection efficiency of evicted contents, all contents cached in storage space are divided into two sets (set_1 and set_2) according to contents' value.

set_1 contains contents with lower value, accounting for $\zeta\%$ all contents. Another set set_2 contains other contents, that is, contents with higher value. The set_1 is the eviction candidate set for selecting the eviction content. In this way, only the value of the content in

the candidate set needs to be calculated each time when selecting the eviction content, thus significantly reducing the amount of computation and the range of eviction options is much smaller. Although it can reduce the amount of calculations, the value of the content is somewhat time-sensitive because the requested content is dynamic. In order to ensure that the low-value content is in the candidate set, so the values of both sets set_1 and set_2 need to be updated periodically. In addition, to avoid frequent eviction of admission content, the admission is stored in the high-value set set_2.

Although some contents with a high value are filtered out in the candidate set by the above steps, a large number of contents still exist in the candidate set, which effect the efficiency in selecting the evicted content. Therefore, to ensure efficiency in selecting the evicted content, this paper adopts the A3C algorithm as the main component of the cache eviction strategy. The A3C algorithm essentially puts Actor–Critic into multiple threads for simultaneous training and has a high training efficiency, which is suitable for selecting the evicted content in this paper.

Algorithm 2: Updating model

Input: requests sequence; initial cache status; global network parameters θ *and* ω;
Output: $d\theta$ and $d\omega$

Reset gradients $d\theta \leftarrow 0$ *and* $d\omega \leftarrow 0$;
pull global parameters θ *and* ω *to synchronize local parameters* $\theta' = \theta$ and $\omega' = \omega$;
$t = 0$;
while $t \neq t_{max}$ **do**
  if *miss* **then**
    `Caching decision` (Algorithm 1)
  end
  r_t = *Byte hit rate in future timeslot* T;
  $t \leftarrow t + 1$
end
$Q_t = V(s_t; \omega)$;
for i *in* $\{t-1, t-2, ..., 0\}$ **do**
  $Q_i = r_i + \gamma Q_{i+1}$;
  Accumulate gradients of the actor's parameter θ': $d\theta \leftarrow d\theta + \nabla_\theta log\pi(a_i|s_i;\theta')(Q_i - V(s_i;\omega'))$;
  Accumulate gradients of the critic's parameter ω': $d\omega \leftarrow d\omega + \partial(Q_i - V(s_i;\omega'))^2/\partial\omega'$;
end
return $d\theta$ and $d\omega$

4.3.2. State, action and reward

A3C is a reinforcement learning algorithm and includes state space, action space, reward, et al. In this subsection, the state space, action space and reward of our algorithm are given as follows:

- **State Space.** When the first frame of the missing content has been fetched, we randomly select multiple contents combinations in set_1 to generate m candidate eviction sets, whose total size is larger than the missing content. We set the value of missing content and the total value of contents in each candidate eviction set as state s_t, which can be expressed as $s_t = [v^t_{miss}, v^t_1, v^t_2, \dots, v^t_m]$. To integrate *Delayed-Eviction* and the *Unified-Standard* module, we expand the content's value based on Eq. (2), which is demonstrated in Eq. (3). ϵ presents the fetching time of the miss content from cloud to the edge server. When the volume of requests is small, the content value using a function: the frequency of content requests (within $\delta t + \epsilon$) divided by the size of the content.

$$v^t_i = \frac{a \times \tau_i \times e^{\sum_{f=1}^{F_{t-\delta t,t+\epsilon}} \frac{F_{i,f}}{F_{t-\delta t,t+\epsilon}}}}{\tau_a^{t-\delta,t+\epsilon}}. \tag{3}$$

  In addition, other properties of each set can be appended to the state so as to have more abundant information in eviction selection.
- **Action Space.** The length of the action space is $m+1$. Considering the *Unified-Standard*, available actions contain two types. The first one, i.e., $a_t = 0$, means that the requested content is not admitted to the cache. The other one, i.e., $a_t \in \{1, 2, \dots, m\}$, means that the missing content is admitted to the cache and the a_tth candidate eviction set is to be evicted from cache. We select the action with the highest probability as the action to be executed in practice. The probability of the action is derived from the policy $\pi(a \mid s;\theta) = P(a = a_t \mid s = s_t)$. Compared with most algorithms that make decisions one by one during eviction selection, we directly make eviction decisions for a set of eviction candidates.
- **Reward.** As this paper mentioned, a higher byte hit rate (hit rate) means better caching performance. So, the reward r_t of action a_t is defined as the byte hit rate (hit rate) in the following sliding time window after a_t.

4.3.3. A3C network

A3C algorithm contains a global network and some worker networks. The global network structure is the same as each worker network, which is Actor–Critic structure. The only difference is that the global network does not need to be trained and is only used to store the parameters of the Actor–Critic structure. The actor-network and the critic-network form the global network. The actor-network aims to optimize the caching policy $\pi(a \mid s;\theta)$, which means choosing better actions. The critic-network aims to improve the accuracy of the value function, which evaluates the actor's action selection policy.

Algorithm 3: *Adele*

initialize cache status and the global A3C network parameter θ, ω;
$\triangle T = 0$;
while *request arriving* **do**
  Record the information of accessed content;
  Update $\triangle T$;
  if *miss* **then**
    Fetch accessed content from the origin server;
    if *The first flame of the content arrives* **then**
      $ecs \leftarrow$ `Caching decision` (Algorithm 1) ;
      if $ecs \neq null$ **then**
        Evict ecs *from caching space*;
        Cache the requested content;
      end
      Update cache status;
    end
  end
  if $\triangle T >= 2T$ **then**
    create a worker thread;
    $d\theta$, $d\omega \leftarrow$ `Updating model` (Algorithm 2) ;
    $\theta \leftarrow \theta - \alpha d\theta$;
    $\omega \leftarrow \omega - \beta d\omega$;
    $\triangle T = 0$;
  end
  Return the requested content to user
end

To establish A3C network, we first define the value function $V(s_t)$ and the action value function $Q(s_t; a_t)$. Utilizing n-step sampling, the value function and the action value function can be respectively expressed as

$$V(s_t) = E_\pi[r_t + \gamma_{t+1} + \cdots + \gamma^n V(s_{t+n})]. \tag{4}$$

$$Q(s_t, a_t) = r_t + \gamma_{t+1} + \cdots + \gamma^n V(s_{t+n}). \tag{5}$$

Table 3
Base algorithms.

Algorithm	Admission	Eviction	Delayed-Eviction	Unified-Standard
LRU	×	✓	×	×
LFU	×	✓	×	×
LRB	×	✓	×	×
RL-Cache	✓	×	×	×
RL-Cache-LRU	✓	✓	×	×
RL-Belady	✓	✓	×	✓
LRU-delay	×	✓	✓	×
LFU-delay	×	✓	✓	×
LRB-delay	×	✓	✓	×
RL-Cache-LRU-delay	✓	×	✓	×
RL-Belady-delay	✓	✓	✓	×
LRU-LRU	✓	✓	×	✓
LFU-LFU	✓	✓	×	✓
LRB-LRB	✓	✓	×	✓
RL-Cache-RL-Cache	✓	✓	×	✓
RL-Belady	✓	✓	×	✓

Table 4
DateSet.

Trace num	Total request	Total content	Total time	AIRT
1	824,116	359,964	1 h	4.37 ms
2	5,201,205	817,039	1 h	0.7 ms
3	7,996,242	937,964	1 h	0.45 ms

To measure a action advantage, we define the advantage value expressed as

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t), \tag{6}$$

where $V(s_t)$ is the expectation of $Q(s_t; a_t)$ about all actions and represents the average quality of all state–action. When $A(s_t, a_t) > 0$, the action's profit is higher than the average expected profit of other actions in state s_t.

The critic-network has a fully connected layer whose input is the state and output is the state value $V_E(s; \omega)$. The parameter ω is updated by a gradient descent method as follow

$$d\omega \leftarrow d\omega + \partial(A(s_t, a_t; \theta', \omega'))^2 / \partial\omega'. \tag{7}$$

The actor adopts the policy π with parameter θ to generate the action and interact with the environment. The actor-network also has a fully connected layer whose input is the state and output is $\pi(a \mid s; \theta)$. The parameter θ of the policy is updated with a gradient ascent method as follow

$$d\theta \leftarrow d\theta + \nabla_{\theta'} \pi(a_t | s_t; \theta') A(s_t, a_t; \theta', \omega'). \tag{8}$$

We can improve the estimation accuracy of the value function and the probability of choosing a high-profit action based Eqs. (7) and (8).

A3C performs training by creating multiple online learning threads named workers, which have two main operations. One is *fetch*, which directly assign the parameters of the global network to the local network in the worker. The other is a *push*, which updates the parameters $\theta\&\omega$ of the global network by Eq. (9). α and β are the learning rate of actor and critic respectively.

$$\theta = \theta - \alpha d\theta, \omega = \omega - \beta d\omega \tag{9}$$

5. Evaluation

In this section, we firstly introduce the experiment setting in Section 5.1. Then, we illustrate two modules' advantages for existing algorithms in Section 5.2. Finally, in Section 5.3, we evaluate our approach *Adele* to compare some state-of-arts algorithms with real traces (Section 5.1).

5.1. Experiment setting

In this subsection, we introduce five algorithms as baselines and the information detailed of the dataset in experiments.

5.1.1. Simulation environment

The simulation is implemented by python on the Ubuntu 18.04.5-LTS operating system. The simulation devices used in this paper are three servers with the same configuration. These devices is equipped with 64G of RAM, Inter(R) Xeon(R) Gold 5122 CPU @ 3.60 GHz CPU, GeForce RTX 2080 GPU, and 1TB hard disk.

5.1.2. Base algorithms

We compare the caching performance of our algorithm *Adele* with some representative algorithms: *LRU*, *LFU*, *LRB*, *RL-Cache*, *RL-Belady* and other variants shown in Table 3. For example, *LRU* is an eviction algorithm. *LRU-delay* is an eviction algorithm with *Delayed-Eviction* module. *LRU-LRU* is not only an eviction algorithm but also an admission algorithm with *Unified-Standard* module. Other algorithms use the above corresponding representation rules. The only special algorithm is *RL-Cache-LRU* by mixing cache admission and eviction algorithm without *Delayed-Eviction* and *Unified-Standard*.

5.1.3. Dataset

The dataset conclude three traces from real company *ChuangCache* in China and shown in Table 4. The three traces represent the request number of *Low*, *Medium*, and *High* frequencies with an average inter-request time (AIRT) of 4.37 ms, 0.7 ms and, 0.45 ms, respectively.

5.2. Modules performance

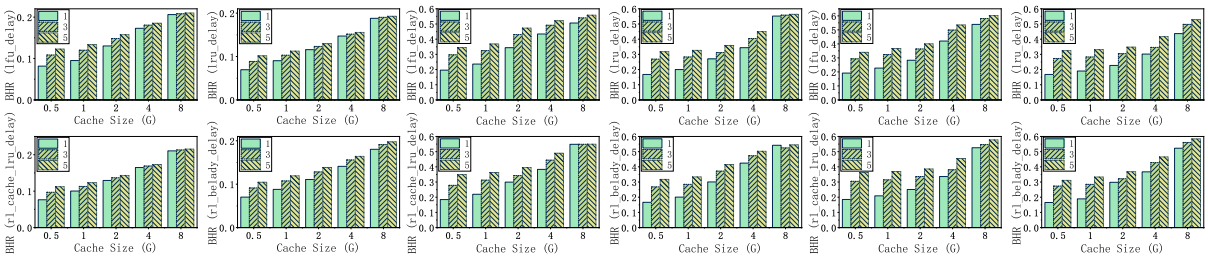
To verify that two modules proposed in this paper can improve the performance for existing algorithms, we make a series of experiments. A detailed analysis of experiments are presented separately in this section for *Delayed-Eviction* and *Unified-Standard*.

5.2.1. Delayed-Eviction

According to statistics [7], the AIRT is 1 μs and the typical fetching times include 1 ms, 10 ms and 200 ms for three use cases: Intra-data center proxy, nearby data center, and remote data center in CDN trace. So, the frequency range of requests generated during fetching time is [1,000, 200,000] in each ms. In our dataset, the min AIRT is 0.45 ms and the frequency range of requests generated during fetching time is about [0.2, 450]. These two frequency ranges are far apart.

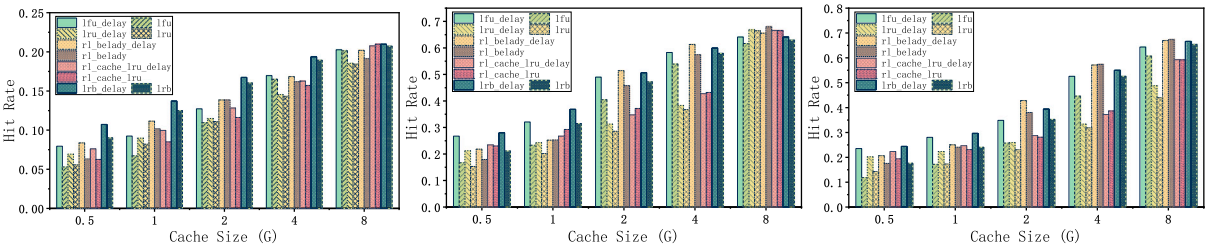
To simulate the high-frequency scenario, we scale down the AIRT by scaling up the fetching time (5s), and the number range of requests is about [1,100, 12,000]. Even to compare the results under different AIRT, we set extra fetching times (1 s and 3 s) and measure the byte hit rate of several original algorithms with *Delayed-Eviction* under different cache sizes and data sources, as shown in Fig. 7. We choose *LFU*, *LRU*, *RL-Cache-LRU*, and *RL-Belady* as the original algorithms. The cache sizes are 0.5G, 1G, 2G, 4G, and 8G. The datasets are from traces numbered 1, 2, and 3 and the result shown in Fig. 7(a), Fig. 7(b), and Fig. 7(c) respectively. We can easily find that the cache byte hit rate gradually increases with increasing fetching time under the same algorithm. The cache byte hit rate also increases with the increase of cache size.

To demonstrate that delayed eviction can provide a performance improvement for all caching replacement algorithms, we compare the byte hit rate and content hit rate under the same conditions with and without this algorithm, as shown in Figs. 8 and 9. The comparison plots show that the algorithm with *Delayed-Eviction* has improved both in terms of byte hit rate and hit rate than the algorithm without *Delayed-Eviction*, both for different data sets and cache sizes. Thus the module has some performance improvement function for the existing caching replacement algorithms.



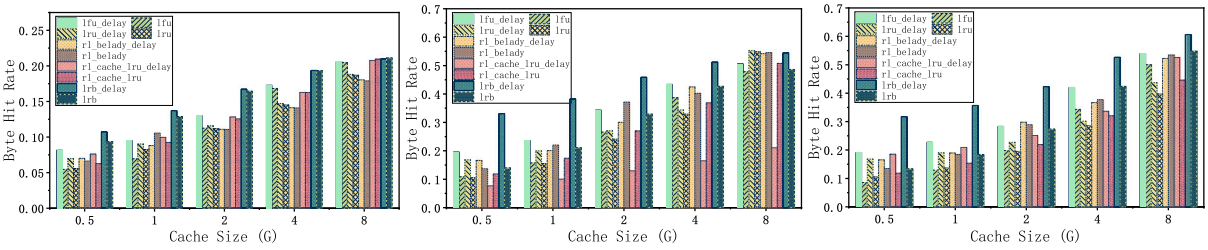
(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 7. [Result Analysis: *Delayed-Eviction*] The byte hit rate of four existing algorithms with *Delayed-Eviction* at three fetching time in different traces.



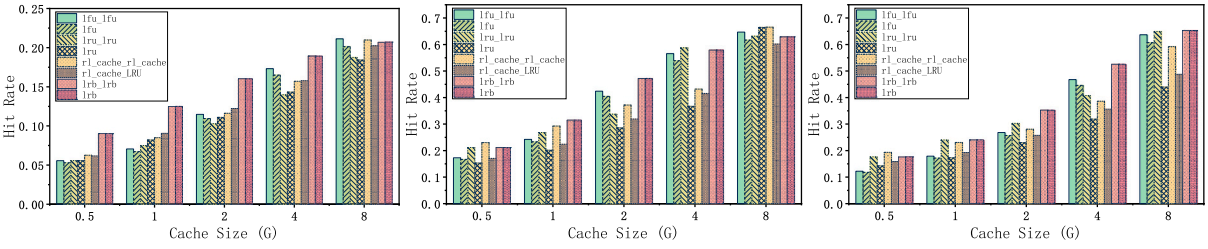
(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 8. [Result Analysis: *Delayed-Eviction*] The hit rate comparison between using *Delayed-Eviction* and without *Delayed-Eviction* for four algorithms in different cache sizes with three traces.



(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 9. [Result Analysis: *Delayed-Eviction*] The byte hit rate comparison between using *Delayed-Eviction* and without *Delayed-Eviction* for four algorithms in different cache sizes with three traces.



(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

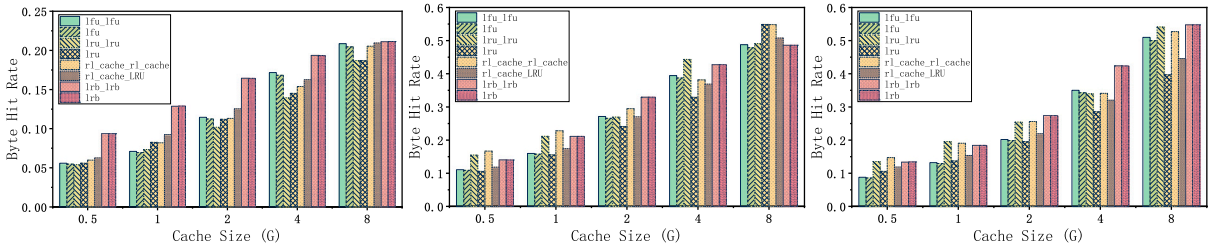
Fig. 10. [Result Analysis: *Unified-Standard*] The hit rate comparison between using *Unified-Standard* and without *Unified-Standard* for four algorithms in different cache sizes with three traces.

5.2.2. *Unified-Standard*

In order to verify that the *Unified-Standard* module can provide performance improvements for all caching replacement algorithms, we set *Unified-Standard* for each cache algorithm and compare them with the same algorithm without *Delayed-Eviction*. We make experiments to measure the cache byte hit rate and content hit rate with different cache sizes and traces. The experiment results of hit rate and byte hit rate are respectively shown in Figs. 10 and 11. We can find that the improvement of byte hit rate and hit rate is more obvious in trace 2 (Figs. 8(b) and 9(b)) and 3 (Figs. 8(c) and 9(c)) compared to trace 1 (Figs. 8(a) and 9(a)). In particular, the performance of some algorithms is not improved in the small cache size in trace 1.

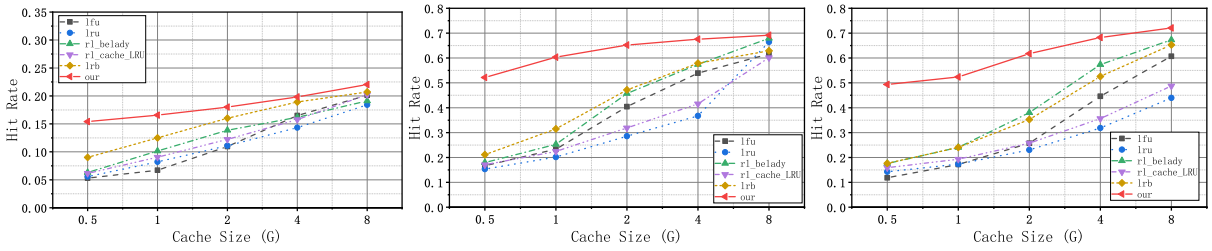
5.3. Overall Performance

We conduct a series of experiments to show the hit rate and byte hit rate performance. Figs. 12(a), 12(b), and 12(c) show that hit rates of each trace with cache size 0.5G, 1G, 2G, 4G, and 8G. We can find that the five comparison algorithms are unstable in their advantages both in different data sources and in different cache sizes. But our algorithm's (red line) hit rate outperforms other algorithms in either



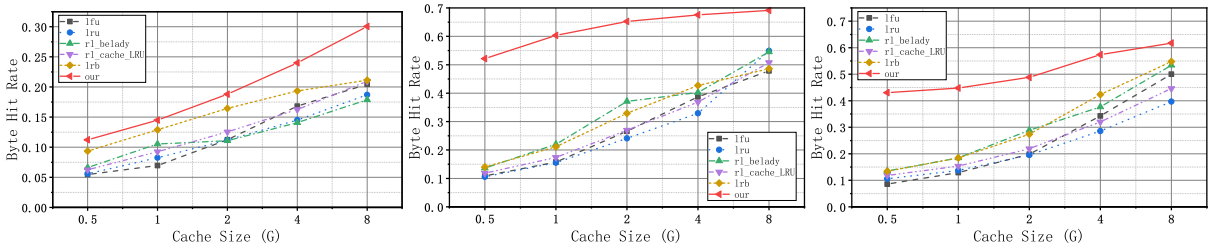
(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 11. [Result Analysis: *Unified-Standard*] The byte hit rate comparison between using *Unified-Standard* and without *Unified-Standard* for four algorithms in different cache sizes with three traces.



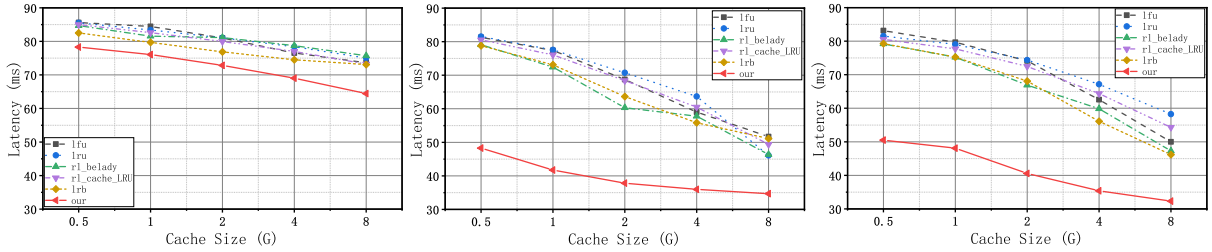
(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 12. [System Performance] The hit rate of *Adele* and five existing solutions in different cache sizes with three traces.



(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 13. [System Performance] The byte hit rate of *Adele* and five existing solutions in different cache sizes with three traces.



(a) The *Low* frequency trace. (b) The *Medium* frequency trace. (c) The *High* frequency trace.

Fig. 14. [System Performance] The transmission latency of *Adele* and five existing solutions in different cache sizes with three traces.

case. So, *Adele*'s **hit rate** is **higher** than other algorithms in the edge server. At the same bandwidth, our algorithm's CTL is less than other algorithms's also.

Figs. 13(a), 13(b), and 13(c) show that byte hit rates of each trace with cache size 0.5G, 1G, 2G, 4G, and 8G. Our algorithm's byte hit rate outperforms other algorithms in either case. The result of the byte hit rate is similar to the hit rate. Byte hit rate can highlight the accuracy of the user waiting time even more than the hit rate. It is because that the waiting time is affected by the missing content size. In conclusion, the higher the byte hit rate is, the **lower** the **waiting time** is. Figs. 14(a), 14(b), and 14(c) show transmission latency in three traces with cache size 0.5G, 1G, 2G, 4G, and 8G. *Adele*'s latency is lower than other algorithms' in each one. Therefore, the BTL and waiting time of our algorithm is shorter than others at the same bandwidth. Finally, we measured the average process time (0.26 ms, 0.27 ms, 0.31 ms, 0.36 ms, 0.4 ms, and 0.87 ms) per request item of six algorithms (LRU, LFU, LRB, RL-Cache, RL-Belady, and *Adele*).

It is worth noting that as the cache size gradually increases, the hit rate and byte hit rate slowly converges to the same value. When the cache size becomes larger than a certain size, the advanced algorithms will degenerate to LRU [39]. This is because with a large cache capacity, contents that have been accessed more than twice will all be kept in cache by LRU. Other advanced algorithms have a longer eviction distance for content that appears more than twice compared

to only once. Thus, content that has only appeared once is eliminated preferentially, which is quite similar to LRU [40]. So, in this paper, when the cache size is near 8G, the comparison algorithm, including LRU, is close to *Adele*. Since both hit rate and byte hit rate represent cache performance, they also show such similarity in performance with 8G cache capacity.

To verify the impact of these two modules on the cache performance, we compared the hit rate & byte hit rate with and without Delayed-Eviction & Unified-Standard, as shown in Figs. 7–9 and 10–11, respectively. And, we conduct a series of experiments in *Adele* combined two modules to show the hit rate and byte hit rate performance, as shown in Figs. 12–14. The comparison plots show that existing algorithms with Delayed-Eviction have improved both in terms of byte hit rate and hit rate than existing algorithms without Delayed-Eviction, both for different data sets and cache sizes. This indicates that Delayed-Eviction can boost the effectiveness of current cache replacement techniques. From Figs. 10–11, we can find that existing algorithms with Unified-Standard does not have a lower overall hit rate or byte hit rate than ones without Unified-Standard. This means that Unified-Standard can improve the performance of existing cache replacement algorithms. Figs. 12–14 demonstrates that the proposed algorithm outperforms the competition in terms of hit rate and byte hit rate. It has a shorter latency than the other algorithms. Another way to identify that our algorithm *Adele* has a larger hit rate improvement than the value of the existing algorithms utilizing any module is to compare the results (such as Figs. 8, 10, and 12). The above analysis also shows that the performance improvement of *Adele* is not only contributed by one module, but both modules play a role.

6. Conclusion

In this paper, we find two problems *Fetching-Gap* and *Standard-Conflict* about caching replacement strategies at edge servers. To address these problems, we propose two modules *Delayed-Eviction* and *Unified-Standard* for existing caching replacement strategies to improve the caching performance. Based on the two modules, a new caching replacement strategy *Adele* is designed by the A3C algorithm in this paper. We make a series of experiments to verify the performance about two modules and *Adele*. The result shows that *Delayed-Eviction* and *Unified-Standard* can improve the hit rate and byte hit rate for existing caching replacement strategies. Simulations experiments demonstrated the superiority of the proposed strategy *Adele* to the state-of-the-art in hit rate and byte hit rate.

CRediT authorship contribution statement

Pengmiao Li: Research algorithm development, System architecture design, Data analysis and/or interpretation, Validation, Writing – original draft, Writing – review & editing. **Yuchao Zhang:** Paper conception, Design of research, Writing & review, Funding acquisition. **Huahai Zhang:** Research algorithm development, Data analysis and/or interpretation, Validation, Writing – original draft. **Wendong Wang:** Conceptualization, Supervision, Funding acquisition. **Ke Xu:** Review the quality of papers, Writing & review. **Zhili Zhang:** Review the quality of papers, Writing & review.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

The work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62072047 and 62172054, the National Key R&D Program of China under Grant 2019YFB1802603, the Key Project of Beijing Natural Science Foundation, China under M21030, and the BUPT-ChuangCache Joint Laboratory Project under Grant A2022164. The work of Pengmiao Li was supported in part by the BUPT Excellent Ph.D. Students Foundation, China under CX2019134.

References

[1] We Are Social & Hootsuite, 2022. https://wearesocial.cn/.
[2] M. Lee, F. Leu, Y. Chen, Pareto-based cache replacement for YouTube, World Wide Web 18 (6) (2015) 1523–1540, http://dx.doi.org/10.1007/s11280-014-0318-9.
[3] D. Mátáni, K. Shah, A. Mitra, An O(1) algorithm for implementing the LFU cache eviction scheme, CoRR abs/2110.11602, 2021, arXiv:2110.11602.
[4] G. Vietri, L.V. Rodriguez, W.A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, G. Narasimhan, Driving cache replacement with ML-based lecar, in: A. Goel, N. Talagala (Eds.), 10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018, USENIX Association, 2018.
[5] Z. Song, D.S. Berger, K. Li, W. Lloyd, Learning relaxed belady for content distribution network caching, in: R. Bhagwan, G. Porter (Eds.), 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020, USENIX Association, 2020, pp. 529–544.
[6] G. Yan, J. Li, RL-Bélády: A unified learning framework for content caching, in: C.W. Chen, R. Cucchiara, X. Hua, G. Qi, E. Ricci, Z. Zhang, R. Zimmermann (Eds.), MM '20: The 28th ACM International Conference on Multimedia, Virtual Event / Seattle, WA, USA, October 12-16, 2020, ACM, 2020, pp. 1009–1017, http://dx.doi.org/10.1145/3394171.3413524.
[7] N. Atre, J. Sherry, W. Wang, D.S. Berger, Caching with delayed hits, in: H. Schulzrinne, V. Misra (Eds.), SIGCOMM '20: Proceedings of the 2020 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, Virtual Event, USA, August 10-14, 2020, ACM, 2020, pp. 495–513, http://dx.doi.org/10.1145/3387514.3405883.
[8] C. Zhang, H. Tan, G. Li, Z. Han, S.H. Jiang, X. Li, Online file caching in latency-sensitive systems with delayed hits and bypassing, in: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022, IEEE, 2022, pp. 1059–1068, http://dx.doi.org/10.1109/INFOCOM48880.2022.9796969.
[9] R. Krishnan, H.V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T.E. Anderson, J. Gao, Moving beyond end-to-end path information to optimize CDN performance, in: A. Feldmann, L. Mathy (Eds.), Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009, Chicago, Illinois, USA, November 4-6, 2009, ACM, 2009, pp. 190–201, http://dx.doi.org/10.1145/1644893.1644917.
[10] X. Fan, E. Katz-Bassett, J.S. Heidemann, Assessing affinity between users and CDN sites, in: M. Steiner, P. Barlet-Ros, O. Bonaventure (Eds.), Traffic Monitoring and Analysis - 7th International Workshop, TMA 2015, Barcelona, Spain, April 21-24, 2015. Proceedings, in: Lecture Notes in Computer Science, vol. 9053, Springer, 2015, pp. 95–110, http://dx.doi.org/10.1007/978-3-319-17172-2_7.
[11] C. Zhang, H. Tan, G. Li, Z. Han, S.H. Jiang, X. Li, Online file caching in latency-sensitive systems with delayed hits and bypassing, in: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022, IEEE, 2022, pp. 1059–1068, http://dx.doi.org/10.1109/INFOCOM48880.2022.9796969.
[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny, Workload analysis of a large-scale key-value store, in: P.G. Harrison, M.F. Arlitt, G. Casale (Eds.), ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012, ACM, 2012, pp. 53–64, http://dx.doi.org/10.1145/2254756.2254766.
[13] B.M. Maggs, R.K. Sitaraman, Algorithmic nuggets in content delivery, Comput. Commun. Rev. 45 (3) (2015) 52–66, http://dx.doi.org/10.1145/2805789.2805800.
[14] G. Einziger, R. Friedman, B. Manes, Tinylfu: A highly efficient cache admission policy, ACM Trans. Storage 13 (4) (2017) 35:1–31, http://dx.doi.org/10.1145/3149371.
[15] D.S. Berger, R.K. Sitaraman, M. Harchol-Balter, AdaptSize: Orchestrating the hot object memory cache in a content delivery network, in: A. Akella, J. Howell (Eds.), 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017, USENIX Association, 2017, pp. 483–498.

[16] Y. Guan, X. Zhang, Z. Guo, CACA: Learning-based content-aware cache admission for video content in edge caching, in: L. Amsaleg, B. Huet, M.A. Larson, G. Gravier, H. Hung, C. Ngo, W.T. Ooi (Eds.), Proceedings of the 27th ACM International Conference on Multimedia, MM 2019, Nice, France, October 21-25, 2019, ACM, 2019, pp. 456–464, http://dx.doi.org/10.1145/3343031.3350890.

[17] V. Kirilin, A. Sundarrajan, S. Gorinsky, R.K. Sitaraman, RL-cache: Learning-based cache admission for content delivery, IEEE J. Sel. Areas Commun. 38 (10) (2020) 2372–2385, http://dx.doi.org/10.1109/JSAC.2020.3000415.

[18] N. Beckmann, H. Chen, A. Cidon, LHD: Improving cache hit rate by maximizing hit density, in: 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 18, USENIX Association, Renton, WA, 2018, pp. 389–403.

[19] S. Li, J. Xu, M. van der Schaar, W. Li, Popularity-driven content caching, in: 35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016, IEEE, 2016, pp. 1–9, http://dx.doi.org/10.1109/INFOCOM.2016.7524381.

[20] Akamai, 2022. https://www.akamai.com/.

[21] C. Chan, S. Hu, P. Wang, Y. Chen, A FIFO-based buffer management approach for the ATM GFR services, IEEE Commun. Lett. 4 (6) (2000) 205–207, http://dx.doi.org/10.1109/4234.848414.

[22] N. Blefari-Melazzi, G. Bianchi, A. Caponi, A. Detti, A general, tractable and accurate model for a cascade of LRU caches, IEEE Commun. Lett. 18 (5) (2014) 877–880, http://dx.doi.org/10.1109/LCOMM.2014.031414.132727.

[23] D. Lee, J. Choi, J. Kim, S.H. Noh, S.L. Min, Y. Cho, C. Kim, 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, Atlanta, Georgia, USA, May 1-4, 1999, ACM, 1999, pp. 134–143.

[24] Z. Sun, M.R. Nakhai, Distributed learning-based cache replacement in collaborative edge networks, IEEE Commun. Lett. 25 (8) (2021) 2669–2672, http://dx.doi.org/10.1109/LCOMM.2021.3081823.

[25] R.A. Dziyauddin, D. Niyato, N.C. Luong, A.A.A.M. Atan, M.A.M. Izhar, M.H. Azmi, S.M. Daud, Computation offloading and content caching and delivery in vehicular edge network: A survey, Comput. Netw. 197 (2021) 108228, http://dx.doi.org/10.1016/j.comnet.2021.108228.

[26] Y. Zhang, P. Li, Z. Zhang, B. Bai, G. Zhang, W. Wang, B. Lian, Challenges and chances for the emerging short video network, in: IEEE International Conference on Computer Communications, Infocom, IEEE, 2019.

[27] T. Zong, C. Li, Y. Lei, G. Li, H. Cao, Y. Liu, Cocktail edge caching: Ride dynamic trends of content popularity with ensemble learning, in: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, 2021, pp. 1–10, http://dx.doi.org/10.1109/INFOCOM42981.2021.9488910.

[28] T. Li, T. Braud, Y. Li, P. Hui, Lifecycle-aware online video caching, IEEE Trans. Mob. Comput. 20 (8) (2021) 2624–2636, http://dx.doi.org/10.1109/TMC.2020.2984364.

[29] X. He, K. Wang, H. Lu, W. Xu, S. Guo, Edge QoE: Intelligent big data caching via deep reinforcement learning, IEEE Netw. 34 (4) (2020) 8–13, http://dx.doi.org/10.1109/MNET.011.1900393.

[30] Y. Zhang, P. Li, Z. Zhang, B. Bai, G. Zhang, W. Wang, B. Lian, K. Xu, AutoSight: Distributed edge caching in short video network, IEEE Netw. 34 (3) (2020) 194–199, http://dx.doi.org/10.1109/MNET.001.1900345.

[31] S. Liu, C. Zheng, Y. Huang, T.Q.S. Quek, Distributed reinforcement learning for privacy-preserving dynamic edge caching, IEEE J. Sel. Areas Commun. 40 (3) (2022) 749–760, http://dx.doi.org/10.1109/JSAC.2022.3142348.

[32] P. Li, Y. Zhang, H. Zhang, W. Wang, K. Xu, Z. Zhang, CRATES: A cache replacement algorithm for low access frequency period in edge server, in: 17th International Conference on Mobility, Sensing and Networking, MSN 2021, Exeter, United Kingdom, December 13-15, 2021, IEEE, 2021, pp. 128–135, http://dx.doi.org/10.1109/MSN53354.2021.00033.

[33] T. Xie, T. He, P.D. McDaniel, N. Nambiar, Attack resilience of cache replacement policies, in: 40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021, IEEE, 2021, pp. 1–10, http://dx.doi.org/10.1109/INFOCOM42981.2021.9488697.

[34] J. Gao, S. Zhang, L. Zhao, X. Shen, The design of dynamic probabilistic caching with time-varying content popularity, IEEE Trans. Mob. Comput. 20 (4) (2021) 1672–1684, http://dx.doi.org/10.1109/TMC.2020.2967038.

[35] L. Qiu, G. Cao, Popularity-aware caching increases the capacity of wireless networks, IEEE Trans. Mob. Comput. 19 (1) (2020) 173–187, http://dx.doi.org/10.1109/TMC.2019.2892419.

[36] J. Yao, T. Han, N. Ansari, On mobile edge caching, IEEE Commun. Surv. Tutor. 21 (3) (2019) 2525–2553, http://dx.doi.org/10.1109/COMST.2019.2908280.

[37] M. Amadeo, G. Ruggeri, C. Campolo, A. Molinaro, Diversity-improved caching of popular transient contents in vehicular named data networking, Comput. Netw. 184 (2021) 107625, http://dx.doi.org/10.1016/j.comnet.2020.107625.

[38] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H.C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, V. Venkataramani, Scaling memcache at facebook, in: N. Feamster, J.C. Mogul (Eds.), Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013, USENIX Association, 2013, pp. 385–398.

[39] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, B. Cheng, OSCA: An online-model based cache allocation scheme in cloud block storage systems, in: USENIX Annual Technical Conference, 2020.

[40] K. Zhou, S. Sun, H. Wang, P. Huang, X. He, R. Lan, W. Li, W. Liu, T. Yang, Demystifying cache policies for photo stores at scale: A tencent case study, in: Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018, ACM, 2018, pp. 284–294, http://dx.doi.org/10.1145/3205289.3205299.

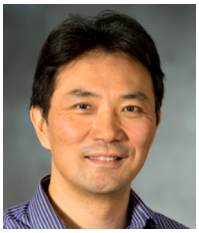
Pengmiao Li is currently a Ph.D. candidate in State Key Laboratory of Networking and Switching Technology from Beijing University of Posts and Telecommunications, Beijing, China. Her current research interests include edge caching, CDN and edge computing.

Yuchao Zhang received her Ph.D. degree from Computer Science Department at Tsinghua University in 2017. Before that, she received the B.S. degree in computer science and technology from Jilin University in 2012. Her research interests include large scale datacenter networks, content delivery networks, data-driven networks and edge computing. She is currently with the Beijing University of Posts and Telecommunications as an associate professor.

Huahai Zhang is currently a mphil student from Beijing University of Posts and Telecommunications, Beijing, China. His current research interests include edge caching, data systems, and data-driven networks.

Wendong Wang (M'05) received his B.E. and M.E. degrees both from the Beijing University of Posts and Telecommunications, China, in 1985 and 1991, respectively, where he is currently a Full Professor in State Key Laboratory of Networking and Switching Technology. He has published over 200 of papers in various journals and conference proceedings. His current research interests are the next generation net-work architecture, network resources management and QoS, and mobile Internet. He is a member of IEEE.

Ke Xu [M'02, SM'09] received his Ph.D. from the Department of Computer Science and Technology at Tsinghua University, where he serves as a full professor. He serves as an associate editor for IEEE Internet of Things Journal and has guest edited several special issues in IEEE and Springer Journals. His research interests include next generation Internet, P2P systems, Internet of Things, network virtualization, and network economics. He is a member of ACM.



Zhili Zhang graduated with B.S. in Computer Science with highest distinction from Nanjing University, Nanjing, China. Then received his M.S. and Ph.D. degrees in computer science from the University of Massachusetts, Amherst in 1992 and 1997. He joined the Department of Computer Science and Engineering at University of Minnesota in January 1997, where he is now a Full Professor. Zhi-Li is a Fellow of IEEE.