PackCache: An Online Cost-driven Data Caching Algorithm in the Cloud

Jiashu Wu, Hao Dai, Yang Wang, Yong Zhang, Dong Huang, and Chengzhong Xu, *Fellow, IEEE*

Abstract—In this paper, we study a data caching problem in the cloud environment, where multiple frequently co-utilised data items could be packed as a single item being transferred to serve a sequence of data requests dynamically with reduced cost. To this end, we propose an online algorithm with respect to a homogeneous cost model, called *PackCache*, that can leverage the FP-Tree technique to mine those frequently co-utilised data items for packing whereby the incoming requests could be cost-effectively served online by exploiting the concept of anticipatory caching. We show the algorithm is $2/\alpha$ competitive, reaching the lower bound of the competitive ratio for any deterministic online algorithm on the studied caching problem, and also time and space efficient to serve the requests. Finally, we evaluate the performance of the algorithm via experimental studies to show its actual cost-effectiveness and scalability.

Index Terms—Data Caching, Data Packing, Cloud Computing, Competitive Ratio, Complexity Analysis

1 Introduction

As the complexity of applications keeps increasing, various tasks in real life may require more than one data items to collaboratively complete the task. For instance, during text preprocessing [1], a tokenisation model and a token dictionary may be required together for Chinese sentence analysis. Hence, given that data items are sometimes correlated in accesses, they thus can be packed to serve the requests via caching at reduced cost. As a way to reduce communication overhead and minimise response delay for data accesses in the context of cloud computing, data caching is playing an essential role in managing data in such applications. Despite of the co-utilisation pattern of data items, past works seldom transfer data items via caching in a packed manner [2, 3], especially in online scenarios, where data layout and access optimisation is infeasible due to the online nature. Hence, it hinders the cost-efficiency of applications in the cloud environment when serving data requests.

Motivated by the observation that data-item co-utilisation is frequent in the cloud environment, and the cost benefit can be brought by the packing mechanism, an offline caching algorithm, which combines a greedy strategy and a dynamic programming technique to serve requests in a packable manner, was proposed to minimise the service cost [4]. However, the offline scenario is not always realistic. On the other hand, the Jaccard Similarity-based (JS-based) method used in [4] might also perform unstably in online settings to mine the frequent co-utilised data items [5], thus crippling the deployments of the algorithm in practice.

In this paper we address these issues by presenting an online packable data caching algorithm, called *PackCache*, which utilises the FP-Tree to discover frequently co-utilised data items as it performs more stable and generates less variations [5] than the JS-based method in online settings. Additionally, the algorithm leverages the concept of anticipatory caching [6] to serve the requests and maintain the data caches in an online fashion. To our best knowledge, the work in this paper is among the first to tackle the caching problem in an online packable manner.

The *PackCache* algorithm is designed by following the setting adopted in our previous studies [2, 7, 3], where a homogeneous cost model is employed [8] — for each cache server, the storage cost is fixed and for each pair of servers, the communication costs are also identical. Based on this model, we further show that the *PackCache* algorithm is $2/\alpha$ competitive, where α is the discount factor, i.e., the cost saving ratio achieved by the packing mechanism. Moreover, we also prove that $2/\alpha$ is the lower bound of the competitive ratio of the packable caching problem for any deterministic online algorithm, which verifies that the proposed algorithm is tight in terms of competitive performance. Finally, the *PackCache* algorithm is both time and space efficient in serving a request in $O(1)$ time and $O(n^2)$ space, where n is the length of the request sequence.

To evaluate its actual performance in reality, we also implement the *PackCache* algorithm and conduct experiments to show its cost-effectiveness and scalability. In summary, we make the following contributions in this paper:

- We study a cost-driven data caching problem with packable serving mechanism under the homogeneous cost model and propose a cost-efficient *PackCache* algorithm that utilises the FP-Tree to discover frequently co-utilised data items and adopt the anticipatory caching mechanism to serve requests.
- We show that the *PackCache* algorithm achieves a competitive ratio of $2/\alpha$, which is also shown to

* *This work is supported by Key-Area Research and Development Program of Guangdong Province (2021B0101400005), Yang Wang is the corresponding author (yang.wang1@siat.ac.cn).*

- *Jiashu Wu, Hao Dai and Dong Huang are with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, 518055, and the University of Chinese Academy of Sciences, Beijing, 100049.*
- *Yang Wang and Yong Zhang is with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, 518055.*
- *Chengzhong Xu is with the University of Macau, Macau, 999078.*
- *DOI: 10.1109/TC.2022.3191969*

Manuscript received Jan 00, 2022; revised Jan 00, 2022.

be the lower bound of the online packable caching problem for any deterministic online algorithm.

- We implement the *PackCache* algorithm and evaluate its performance in practice. The results show that compare with its individually-served counterpart, the *PackCache* algorithm is cost effective and scalable, i.e., under α value of 0.8 and 0.6, a cost reduction of 5% and 10.7% can be achieved.

The rest of the paper is organised as follows: Section 2 presents some related works and compares them with ours to demonstrate our research opportunities. After that, we present our model and the detailed problem formulation in Section 3. The *PackCache* algorithm, its competitive ratio analysis and complexity analysis are presented in Section 4. Experimental setups and results are illustrated in Section 5. The last section concludes the paper.

2 Related Work

The caching problems can be divided into online case and offline case distinguished by different ways to receive requests. In the offline case, the full knowledge about requests is available in advance, while the online case knows nothing about the future requests sequence. In the offline setting and a QoS perspective, Zhang et al., [9] studied the delay-optimal cooperative caching in the edge environment and proposed a greedy caching placement algorithm with an approximation ratio of $(1-1/e)$ in linear time. By following a similar idea, Zhang et al., [10] introduced a collaborative hierarchical caching mechanism with an attempt to maximise the overall cache hitting rate. George et al., [11] unified both goals by utilising a cooperative caching algorithm with proactive cache updating policy to reduce the delay in online video access and jointly improve the cache hit ratio.

However, our work differs from previous works in terms of system model and problem goal. In particular, instead of being capacity-oriented and aiming to maximise the cache hit ratio, the goal of our algorithm is to minimise the cost of request serving given adequate resources in the cloud.

Later, in the offline setting and a cost perspective, Li et al., [12] established a Markov model and a multiple linear regression model to assist caching usage prediction and guide the caching placement. From the theoretical perspective, Khanafer et al., [13] formulated the computation and caching cost trade-off as a constrained ski-rental problem assisted by the first or second moment of the arrival distribution, which outperformed existing approaches in worst-case competitive ratio. Puttaswamy et al, [14] and Erradi et al., [15] also tackled the storage cost saving by utilising a hybrid adaptive storage solution under multiple storage services, and via a tier-wise object placement algorithm, respectively. Both methods benefited the cost saving during cloud data storage. Wang et al., [6] proposed a 2-competitive online algorithm that applied an individually-served manner to handle the incoming requests and utilised anticipatory caching to maintain the caches of data items under the content delivery edge network. However, none of the above works attempted the packable mechanism during cloud caching.

Recently, Huang et al., [4] introduced the DP_Greedy algorithm that combined an existing dynamic programming (DP)-based algorithm and a greedy strategy to effectively cache data items in the cloud environment with packing being enabled. However, their work was only feasible in offline settings, leaving the online setting untouched.

All aforementioned methods are either in absence of packing mechanism when serving data requests at all or only feasible in offline settings that possess complete knowledge about requests, such as the work in [4]. To the best of our knowledge, there is still in shortage of cost-effective packable data caching algorithms working in online settings. Our work instead fills this gap in a realistic and cost-effective way by incorporating the co-relationships between data items into an online algorithm via a packing mechanism to serve data requests.

3 Problem Formulation

In this section, we describe the problem formulation of the cost-driven packable data caching problem in details. We first define some useful concepts that will be used in this paper, and then give a standard form of the solution to the problem following the idea proposed in study [2, 4].

3.1 Problem Model

Suppose in a cloud environment, there are k distinct data items with diverse co-utilisation patterns. The set of data items are denoted by $D = \{d_1, d_2, \cdots, d_k\}$, which will be cached in a fully connected network with m cache servers, denoted by $S = \{s_1, s_2, \cdots, s_m\}$. A sequence of data requests, $R = \{r_1, r_2, \cdots, r_n\}$, are made to request these data items, where the tuple $r_i =< s_j, t_i, D_i >$ represents that request r_i is made at server s_j ($s_j \in S$) at time t_i for a data item subset D_i. For each request r_i, it can either request a single data item, i.e., $D_i = \{d_{i1}\}, d_{i1} \subseteq D$, or can request two data items, i.e., $D_i = \{d_{i1}, d_{i2}\}, d_{i1}, d_{i2} \in D$.

When serving a data request, the shared data items need to be either held locally in the cache of server that receives the data request, or be replicated and transferred from another server to the server with the received request to satisfy the request. After being used, the data will be destroyed at certain time to achieve minimal caching cost so that the requests can be served in a cost-efficient manner.

Different from previous works, in this paper we consider the online setting, i.e., there is no knowledge of where and when each data request is made and which data item subset is requested. For simplicity, we assume that there exists at most one request per time instance as many other previous studies assumed [2, 7, 3, 4], so we can use t_i to represent the arrival time of the request r_i.

As the functionalities of cloud applications become more sophisticated, it is likely that two correlated data items could be frequently utilised together (i.e., co-utilised) by requests. In this circumstance, packing these data items as a package to serve data requests jointly is both convenient and cost-effective. We define a discount factor α, $(0 < \alpha \leq 1)$, which measures the ratio between cost of serving a request of two frequent data items in a packed manner, and the cost of the individually-served non-packing case. Note that the packing will not be leveraged during caching for the online case to avoid time-consuming unpacking during single data requests. On the other hand, for infrequently co-utilised data items, the packing mechanism will not be applied. Since if data items are frequently co-utilised, the application that requests these data items usually possesses an optimised way to handle and unpack the package, hence the burden caused by unpacking is negligible. While for infrequent data item pairs, the lack of optimised unpacking mechanism and subsequent single data accesses yield extra cost burdens. Hence, it is not worth packing infrequent data items. In this paper, we only consider packages with two data items, but without loss of generality, the algorithm is convenient to be extended to multiple data item packing cases.

We also adopt the space-time diagram [4] to clarify the problem. A feasible schedule (shown in Fig. 1), is a way to use caching or transferring to get all data requests satisfied along the timeline, and a standard form of a schedule is that all transfers occur at the request time instance. The work [7] confirmed that there exists at least one optimal schedule which belongs to the standard form. Different from previous research [2], we consider the multiple data item caching problem and we take data item's co-utilisation into account and enable the packing mechanism, instead of serving them individually. Besides, our work stands out from [4] as our proposed algorithm works in the online setting, which makes no assumption regarding data request sequence, and provide the corresponding competitive ratio with its lower bound.

In terms of request satisfaction, since there exists a sequence of data requests, we define the request being satisfied as the data items in that specific request are satisfied.

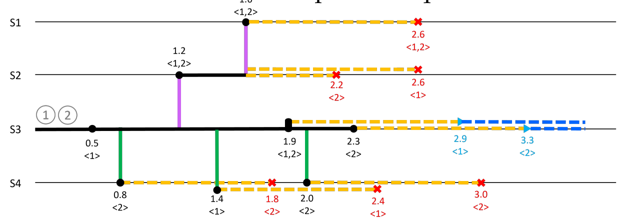

Fig. 1. A feasible schedule (bold lines) of a packable caching model ($\mu = \lambda$). The data items are initially on server S3. Each vertex and numbers below it represent at time t a request is asking for data item in $<>$. Horizontal lines represent data caching, while green and purple vertical lines indicate individual and packed data item transferring, respectively. The dashed lines indicates extra caching made by the system. The red cross represents cache elimination, while the blue rectangle represents although the expiration is detected, the data is still kept to prevent data loss. The cost is $C = (1 + 2 + 4.7 + 3)\mu + (3 + 2 * 2 * \alpha)\lambda$.

3.2 Cost Model

We adopt the same model presented in [4] to conduct our study. A universal caching costs per time unit for each server is applied, denoted by μ, and the transfer costs between any pair of servers are also identical, denoted by λ. Suppose $C_{ij}^p, (1 \leqslant p \leqslant k)$ represents the cost to serve a data item d_p contained in request $r_j =< s_j, t_j, D_j >$, i.e., $d_p \in D_j$, and request $r_i =< s_i, t_i, D_i >$, $d_p \in D_i$, is the most recent request for data item d_p before r_j. If r_j is the first request for data d_p, $t_i = 0$. We give a formal definition of C_{ij}^p as follows:

$$C_{ij}^p = \begin{cases} (t_j - t_i)\mu + \epsilon\lambda, & t_j > t_i \\ +\infty, & \text{Otherwise} \end{cases} \tag{1}$$

where ϵ is a variable with value in $\{0, 1\}$ to signify how to treat the transfer cost during the computation of C_{ij}^p. Specifically, if the request r_j is served by a local data cache resulted from r_i, i.e., $s_i = s_j$, then $\epsilon = 0$. Otherwise, $\epsilon = 1$, implying the requested data item d_p is first cached from t_i to t_j on cache server s_i, and then transferred from s_i to s_j to serve request r_j. If d_p is one of the data item in a request for a frequent data pair and server s_i contains the requested data pair, the corresponding transfer cost λ will have the discount factor α being applied, i.e., $\epsilon\alpha\lambda$. Note that the cost model is generalisable to reflect the overheads faced by the algorithm. Considering that in practice the cost of transferring a unit of data is fixed, then the cost reduction proportionally reflects the reduction in terms of data communication overhead. Similarly, considering that the network condition provided by modern cloud service providers are relatively stable, then the communication overhead reduction also proportionally translates to the decrease of time overhead.

3.3 Problem Goal

To satisfy a data request, data items may need to be either cached locally to serve the subsequent requests made on this server or transferred from another cache server that has the requested data item so that the request could be satisfied. A transfer operation implies the data item is replicated, then the copy is transferred and cached in the destination server and then destroyed in the future for cost saving. Since the replication cost and deletion cost are always constants, they can be merged into the transfer cost or the caching cost. Without loss of accuracy, we assume these cost are free as in many previous studies [2, 7, 3, 4].

The problem goal is to find an optimal schedule among many feasible schedules so that the total cost to serve all these data requests is minimised. We denote $\Pi(i)$ as all feasible schedules to satisfy the requests up to r_i, $\phi(R)$ as a feasible schedule and $\phi^*(R)$ as the optimal for this n-length request sequence, i.e., $|R| = n$. Each schedule $\phi(R)$ has a cost $cost(\phi(R))$, which is defined as follows:

$$cost(\phi(R)) = \sum_{r_i \in R} cost(r_i) = \sum_{r_i \in R} \sum_{d_p \in D_i} cost(d_p) \tag{2}$$

whereby we have the same formal definition of the problem goal as in [4]:

$$\phi^*(R) = \operatorname*{arg\,min}_{\phi(R) \in \Pi(R)} cost(\phi(R)) \tag{3}$$

In this paper, we study the online form of this problem and give an $2/\alpha$-competitive online algorithm.

4 An Online $2/\alpha$-Competitive Algorithm

Given that the full information regarding data requests are usually not available in advance in practice, to make the algorithm more generalisable, we propose the *PackCache* algorithm that can work under an online manner. In this section, we first describe the workflow and rationale of the proposed *PackCache* algorithm, which is a $2/\alpha$-competitive algorithm. Then, we prove the competitive ratio and its lower bound. We finally analyse the time and space complexity of the *PackCache* algorithm.

Algorithm 1 The *PackCache* algorithm

```
Input: online request in the form of $r_i =< s_j, t_i, D_i >$
Output: the average cost to serve this request sequence, denoted by ave_cost
1: /* all data items are initially located at cache server $s_1$ */
2: Initialise: $c[d_k] \leftarrow 1$; $E[d_k]^j \leftarrow 0, 1 \leq j \leq m$
```

```
 3: transfer cost $C_T \leftarrow 0$, caching cost $C_C \leftarrow 0$
 4: if (request $r_i$ arrives $s^j$ at time $t_i$) then
 5:     use $frequent\_item\_miner(r_i)$ to mine frequently co-utilised data items based on minimum support $\gamma$ using request history up to time $t_i$
 6:     $C_T \leftarrow C_T + serve\_request(r_i)$
 7: end if
 8: if (a copy $d_p$ expires on $s_j$ at $t_i$) then
 9:     $copy\_expire(d_p, s_j, t_i)$
10: end if
11: return average request cost
```

In the *PackCache* algorithm, we maintain a global counter $c[d_k]$, which keeps track of the number of alive copies of data item d_k. Without loss of generality, we assume that all data items are initially cached at server s_1 with only one copy, i.e., $c[d_k] = 1$. Besides, we also maintain a local expiration time tracker $E[d_k]^j$ which stands for the expiration time of data item d_k on server s_j and is initialised to be 0. A previous request recorder $r^j_{<k>}$ is also maintained to record the latest request of d_k on server s_j. The initialisation process has been shown in `line 2 - 3` in Algorithm 1.

The *PackCache* algorithm constitutes 3 components, namely the *FP-Tree-based frequent data itemset miner* as shown in Algorithm 2, the *Request serving component* as shown in Algorithm 3 and the *Data item copy expiration handler* as shown in Algorithm 4.

Algorithm 2 Function $frequent\_item\_miner(r_i)$

```
Input: an incoming request in the form $r_i =< s_j, t_i, D_i >$
Output: the $FreqI$, i.e., the frequent data itemsets
1: if (request $r_i$ is a double data item request) then
2:     add request $r_i$ into request history
3:     update FP-Tree
4: end if
5: retrieve all data item pairs that have their support greater than or equal to the minimum support $\gamma$ to form $FreqI$
6: return $FreqI$
```

As shown in Algorithm 2 and `line 5` in Algorithm 1, upon receiving a new data request, the *PackCache* algorithm utilises FP-Tree to discover frequently co-utilised data itemsets with a given minimum support γ based on the request history available till the current time t_i. As indicated in `line 1 - 3` in Algorithm 2, only double data item requests will be added into the request history to guide the mining process. The advantage of utilising the FP-Tree-based frequent itemset miner over the Jaccard Similarity-based one is that the FP-Tree-based frequent itemset miner performs more stable when data requests come in an online manner, and hence it will generate less variations [5, 16].

Algorithm 3 Function $serve\_request(r_i)$

```
Input: an incoming request in form $r_i =< s_j, t_i, D_i >$
Output: the transfer cost of serving this request
 1: transfer cost $C_T \leftarrow 0$
 2: $\Delta t \leftarrow \frac{\lambda}{\mu}$
 3: if (request $r_i$ arrives $s_j$ at time $t_i$) then
 4:     if (request $r_i$ contains a single data item $d_i$) then
 5:         if ($E[d_i]^j = 0$) then
 6:             $r_i$ served by a transfer from $s_k$ with $d_i$, $k \neq j$;
 7:             $C_T \leftarrow C_T + \lambda$
 8:             $E[d_i]^j \leftarrow t_i + \Delta t$
 9:             $c[d_i] \leftarrow c[d_i] + 1$
10:         else if ($E[d_i]^j \neq 0$) then
11:             serve $r_i$ by the local copy on $s_j$
12:             $E[d_i]^j \leftarrow t_i + \Delta t$
13:         end if
14:         $r^j_{<i>} \leftarrow t_i$
15:     else if (request $r_i$ contains two data items $d_{i1}$ and $d_{i2}$) then
16:         if ($E[d^j_{i1}] = 0$ and $E[d^j_{i2}] = 0$) then
17:             if ($d_i = (d_{i1}, d_{i2}) \in FreqI$) then
18:                 $d_{i1}$ & $d_{i2}$ are transferred in two ways for cost minimisation: 1) individual item from any server that caches a copy; 2) packed items from any server with both.
19:                 if transferred individually then
20:                     $C_T \leftarrow C_T + 2\lambda$
21:                 else
22:                     $C_T \leftarrow C_T + 2\alpha\lambda$
23:                 end if
24:             else
25:                 $d_{i1}$ & $d_{i2}$ trans. them from $s^k$, $k \neq j$;
26:                 $C_T \leftarrow C_T + 2\lambda$
27:             end if
28:             $E[d^j_{i1}], E[d^j_{i2}] \leftarrow t_i + \Delta t$
29:             $c[d_{i1}] \leftarrow c[d_{i1}] + 1$, $c[d_{i2}] \leftarrow c[d_{i2}] + 1$
30:         else if ($E[d^j_{i1}] \neq 0$ and $E[d^j_{i2}] \neq 0$) then
31:             serve request $r_i$ by local copy on $s_j$
32:             $E[d^j_{i1}], E[d^j_{i2}] \leftarrow t_i + \Delta t$
33:         else
34:             $d_{i1}$ or $d_{i2}$ is missed locally, trans. from $s_k$, $k \neq j$;
35:             $C_T \leftarrow C_T + \lambda$
36:             $E[d^j_{i1}], E[d^j_{i2}] \leftarrow t_i + \Delta t$
37:             $c[d_{i*}] \leftarrow c[d_{i*}] + 1$, $d_{i*}$ is the absent data item
38:         end if
39:         $r^j_{<i1>}, r^j_{<i2>} \leftarrow t_i$
40:     end if
41: end if
42: return $C_T$
```

The request serving mechanism of the *PackCache* algorithm has been shown in Algorithm 3. Upon receiving a single data item request (`line 4 - 14`), the request serving component will transfer the requested data item from any server s_k who possesses an alive copy if the requested data item is not locally cached on server s_j, i.e., $E[d_i]^j = 0$. Tracker $E[d_i]^j$, $c[d_i]$ and the transfer cost C_T will be updated accordingly as shown in `line 7 - 9`. On the other hand, if the requested data item is available locally, it will be served directly without transferring as shown in `line 10 - 12`. Finally, tracker $r^j_{<i>}$ is updated to reflect the recent request for the data item.

In the *PackCache* setting, requests with two data items are also allowed. The request serving works similarly with the single data item request case, except in the transfer scenarios. As indicated in `line 17 - 23`, when both requested data items are absent, a discounted transfer cost can be applied if these two data items form an itemset that is frequent, otherwise they will be served in an individual manner and the usual transfer cost will be applied as in `line 24 - 26`. Depending on the length of the idle caching time, the algorithm will use the most cost-efficient way to perform the transfer, either transfer these two data items individually, or transfer them in a packed manner. On the other hand, if both requested items are cached locally, no extra transfer is required and the request will be served directly as in `line 30 - 32`. Similar to the single data item requests, as shown in `line 33 - 37`, when any one of the requested data item is not stored locally, the transfer of the lacked data item is done individually, making the discount not applicable. Note that in this case, as well as the single data item request case, the *PackCache* algorithm will only transfer the requested data item that is locally absent, instead of transferring a data item package which contains the required data item. Since in the online setting, complete knowledge about the entire request sequence is lacked, therefore it may not be worthy to use a data item package to satisfy a single data item request, which distinguishes the *PackCache* algorithm with its offline counterpart. Finally, tracker $r_{<i>}$ is updated.

Algorithm 4 Function $copy\_expire(d_p, s_j, t_i))$

Input: the copy expiration event information
Output: no output will be yielded by this function

```
1: $\Delta t \leftarrow \frac{\lambda}{\mu}$
2: if ($c[d_p] = 1$) then
3:     $E[d_p]^j \leftarrow t_i + \Delta t$
4: else
5:     drop the $d_p$ copy at $s_j$
6:     $E[d_p]^j \leftarrow 0$
7:     $c[d_p] \leftarrow c[d_p] - 1$
8: end if
```

For the copy expiration handler, we adopt the mechanism inspired by the *anticipatory caching* concept. The period $\Delta t = \lambda/\mu$ is calculated as shown in `line 1` in Algorithm 4 and is also used in Algorithm 3. If the period between the current time and the time of the latest previous request of this data item is less than or equal to Δt, this data item is worth being cached locally as the caching cost is less than or equal to the transfer cost incurred by the eviction of this data item after its latest previous request. When the period Δt is reached, if this data item has more than one alive copies, then the local copy will be dropped for caching cost efficiency as in `line 4 - 7` in Algorithm 4. Otherwise, if the local copy is the only alive copy, then its expiration time will be extended by another Δt to prevent data loss as in `line 2 - 3` in Algorithm 4.

4.1 Competitive Analysis

The design of the *PackCache* algorithm produces the following observations:

Observation 1. *If $c[d_k] > 1$, i.e., there exists more than one alive copies of data item d_k, then no copy can survive for more than Δt on any server $s_j, j \in [1, m]$.*

Observation 2. *Since the algorithm has no knowledge about future requests, in the worst case, it needs to cache d_k after satisfying its request for Δt period. However, if no subsequent request of d_k comes in Δt period, the optimal algorithm will not cache it at all.*

Observation 3. *Data loss will not happen, hence at any given time, there is always a copy that can serve the incoming request, either by local caching or by transferring.*

Based on these observations, we give the following theorems of the competitive ratio of the *PackCache* algorithm:

Theorem 1. *The* PackCache *algorithm is $\frac{2}{\alpha}$-competitive.*

Proof. In the proof, we denote $C^i_{On\_PC}$ as the cost of serving data request r_i by the proposed *PackCache* algorithm, and denote C^i_{OPT} as the cost of serving the same data request by the optimal offline way in the packed setting. We will first discuss the proof of single data item requests, then extend the proof to double data item requests.

Considering the following cases for a single data item request:

Case 1: If $r_i =< s_j, t_i, \{d_k\} >$ is the first request that arrives at server s_j, $j \in [1, m]$, then we have $C^i_{On\_PC} = \mu\Delta t + \lambda = 2\lambda$ and $C^i_{OPT} = \lambda$. Hence we have $\frac{C^i_{On\_PC}}{C^i_{OPT}} = 2 \leq \frac{2}{\alpha}$, given that $0 < \alpha \leq 1$.

Case 2: When the request $r_i =< s_j, t_i, \{d_k\} >$ arrives, two scenarios should be considered as follows:

Case 2.1: If $t_i \in [r^j_{<k>}, r^j_{<k>} + \Delta t]$ and $E[d_k]^j \neq 0$, which means there is a local copy ready to serve the request and the caching period is within Δt, therefore we have $C^i_{On\_PC} = \mu\Delta t \leq \lambda$, $C^i_{OPT} = \mu\Delta t \leq \lambda$, hence $\frac{C^i_{On\_PC}}{C^i_{OPT}} = 1 < \frac{2}{\alpha}$.

Case 2.2: If $t_i > r^j_{<k>} + \Delta t$ and $E[d_k]^j = 0$, i.e., there exists no local cache of the data item to server the request. Then the transfer will occur, i.e., $C^i_{On\_PC} = \mu\Delta t + \lambda = 2\lambda$, $C^i_{OPT} = \lambda$, hence we have $\frac{C^i_{On\_PC}}{C^i_{OPT}} = 2 \leq \frac{2}{\alpha}$.

Now lets extend the above proof of single data item request to double data item request. For the double data item requests, if these two data items belong to a frequent itemset, a discount factor $0 < \alpha \leq 1$ can be applied during data item transferring, otherwise, the discount factor is not applicable. Firstly, for those double data item requests in which the two data items cannot form a frequent itemset, they will be served in a separated manner and the discount factor is not applicable. Therefore, the cost in this case is twice in the analysis of a single data item request. The constant 2 will be cancelled out for both the *PackCache* and the optimal algorithm and hence the competitive ratio result remains unchanged. On the other hand, for those double data item requests in which the two data items can form a frequent itemset, the result of Case 2.1 will remain unchanged. In that case, the request is served using local cached copy and hence the discount factor is not involved and the result is not affected.

For Case 1, it needs to be reconsidered under the double data item request:

Case 1′: We have $C^i_{On\_PC} = 2\mu\Delta t + 2\lambda = 4\lambda$. On the other hand, $C^i_{OPT} = 2\alpha\lambda$. Hence, $\frac{C^i_{On\_PC}}{C^i_{OPT}} = \frac{2}{\alpha}$.

And for Case 2.2, it also needs to be reconsidered in two scenarios as follows:

Case 2.2′-1: If neither of these two requested data item has available local cache copy, i.e., $E[d_{k1}]^j = E[d_{k2}]^j = 0$. Then, we have $C^i_{On\_PC} = 2\mu\Delta t + 2\lambda = 4\lambda$, while $C^i_{OPT} = 2\alpha\lambda$. Hence, we have $\frac{C^i_{On\_PC}}{C^i_{OPT}} = \frac{2}{\alpha}$.

Case 2.2′-2: If only one of the requested data item is absent locally, then we only need to transfer that data item individually. Hence, we have $C^i_{On\_PC} = \mu\Delta t + \lambda = 2\lambda$, while $C^i_{OPT} = \lambda$. Therefore, we have $\frac{C^i_{On\_PC}}{C^i_{OPT}} = 2 \leq \frac{2}{\alpha}$, given that $0 < \alpha \leq 1$.

Since the $C^i_{On\_PC}$ is the cost of serving request r_i, same for C^i_{OPT} which is the cost of the corresponding optimal algorithm, therefore, for the entire request sequence $R, |R| = n$,

$$\frac{C^R_{On\_PC}}{C^R_{OPT}} = \frac{\sum_{1\leq i\leq n} C^i_{On\_PC}}{\sum_{1\leq i\leq n} C^i_{OPT}} \tag{4}$$

Hence, we can conclude the theorem with the following result:

$$\lim_{n\to+\infty} \frac{C^R_{On\_PC}}{C^R_{OPT}} = \frac{2}{\alpha} \tag{5}$$

□

The following theorem shows that there does not exist any deterministic online algorithm that can yield a better performance than $\frac{2}{\alpha}\times$ optimal result, implying our algorithm is tight.

Theorem 2. *The competitive ratio of the online packable caching problem is lower bounded by $\frac{2}{\alpha}$.*

Proof. Without loss of generality, we establish a special instance with two data items in which both λ and μ are set to 1, and thus Δt is 1. Initially, at least one server has both data items, and at least one server has each single data item. We define l_i to be length of local caching after satisfying request r_i. As mentioned in the previous proof, $2/\alpha$ will only appear in Case 1 and 2.2′-1, where both requested data items are absent locally. Hence, we only focus on the data pair request in which both items are absent.

We prove the theorem by reduction. Initially, a request $r_1 =< s_i, 0, \{d_1, d_2\} >$ arrives, both data items are absent and hence the request will be satisfied by a transfer. Then, the local caching length l_1 will have the following two cases:

Case 1.1: If $l_1 = 1$, then there is no subsequent requests comes after r_1, which yields $\frac{\mathcal{A}}{OPT} = \frac{2+2l_1}{2\alpha} = \frac{2}{\alpha}$. $l_1 > 1$ will never hold since the copy will expire.

Case 1.2: If $l_1 < 1$, it means there is a request comes in ϵ time, $\epsilon < 1$. Similarly, we have l_2 to be the caching length after satisfying r_2, and hence it leads to two cases similar with the above:

Case 2.1: If $l_2 = 1$, then there is no subsequent requests comes after r_2, which yields $\frac{\mathcal{A}}{OPT} = \frac{2+2(l_1+l_2)}{2\alpha+2min(\alpha,l_1)} < \frac{2}{\alpha}$.

Case 2.2: If $l_2 < 1$, it repeats Case 1.2. We keep receiving request r_k. Assume upon request r_{k-1}, $\frac{\mathcal{A}^{k-1}}{OPT^{k-1}} \leq \frac{2}{\alpha}$ holds, we have the following:

Case k: We have $\frac{\mathcal{A}^k}{OPT^k} = \frac{2+2l_k}{\alpha+2min(\alpha,l_k)} < \frac{2}{\alpha}$ and it holds when $l_k \leq 1$ so we omit Case k.2. Hence, we conclude that $2/\alpha$ is a lower bound of the competitive ratio, implying that no deterministic algorithm can do better than this. □

4.2 Complexity Analysis

The implementation of the algorithm follows the workflow described in Algorithm 1 - 4 in Section 4. In terms of the space complexity, maintaining $E[d_k]^j$ costs the highest space consumption among all tracker variables and is $O(mn)$, while constructing the FP-Tree has a space complexity $O(n^2)$. Given that the number of servers m is far less than the number of requests n, therefore, the overall space complexity of the *PackCache* algorithm is $O(n^2)$.

As for the time complexity, when serving each incoming data request, manipulating $E[d_k]^j$, $c[d_k]$ and $r^j_{<i>}$ can be done using $O(1)$ time with efficient implementation. Hence, serving request in general remains in constant time complexity, which is highly efficient. Despite that constructing the FP-Tree, handling expired copies and calculating the caching cost may cause some overhead, fortunately, all these operations can be processed by background daemons running in parallel with the request serving process. Hence, they will not impair the time complexity of request serving.

5 Performance Studies

To verify the performance of our algorithm in practice, extensive experiments are conducted. We design a solver in Python, which effectively implements our algorithm. Follow [4], the experiment data comes from the taxi trace data from City of Shenzhen in China. The territory of city is partitioned into 50 parts, each maintains a cache server to serve the user requests of taxis, which are regarded as shared data items. For instance, at time t_i, server in region s_j receives a request of two taxis will be used as a request in our experiment. The dataset contains various request pairs with relatively high Jaccard Similarity. According to the research results [4], the trace of the taxi can be roughly seen as the trace on how data are requested from different servers.

The algorithm is characterised by several parameters, which include number of data items k, number of caching nodes m, number of requests n, the discount factor α, the minimum support threshold γ for the FP-Tree frequent data itemset mining, the caching cost μ and the transfer cost λ. To concentrate our study on the factors we concerned about, we deliberately ignore some other factors that may influence the algorithm, such as CPU power, network condition and bandwidth of the network, etc. On the other hand, we take the average cost as the major performance metric since many other performances can be reflected from it such as the network bandwidth occupancy rate.

During experiments, 10 taxis are randomly selected, each acts as a distinct data item $(d_1, d_2, \cdots, d_{10})$ as this value can be well handled and without loss of generality to reflect general case. We partition the city into 50 parts, each having a caching server, and set the discount factor $\alpha = 0.8$. The default transfer cost λ and caching cost μ are all set to be 3 to balance between transferring and caching. We set the minimum support $\gamma = 0.01$ based on our experience on research of human mobility behaviors in metropolitan city [4]. Finally, several factors are varied to testify the effectiveness and robustness of the *PackCache* algorithm.

When evaluating the *PackCache* algorithm, we compare it with the algorithm that individually serves packed requests without the data packing mechanism to demonstrate the effectiveness of the packable *PackCache* algorithm. Besides, we also compare with the offline counterpart to show that although the performance is lower bounded by $2/\alpha$, the *PackCache* algorithm usually performs better than the theoretical lower bound.

5.1 Impact of Ratio $\rho = \lambda/\mu$

When evaluating the effectiveness and robustness of the *PackCache* algorithm with different ρ ratios, we set $\lambda+\mu = 6$ intentionally. The experimental results of the transfer cost is illustrated in Fig. 2(a).

According to Fig. 2(a), the average transfer cost increases as the ratio ρ increases. It is natural to observe because as the ratio ρ increases, the transfer cost λ is gradually emphasised while the caching cost μ is gradually declined. Under all ρ settings ranging between 0.2 and 5.0, the *PackCache* algorithm outperforms its individually-served counterpart, which demonstrates the superiority and robustness of the *PackCache* algorithm. For instance, when ρ is set to 1, the cost reduction achieved by the *PackCache* is around 4.6%. Besides, we can observe that the cost does not increase in

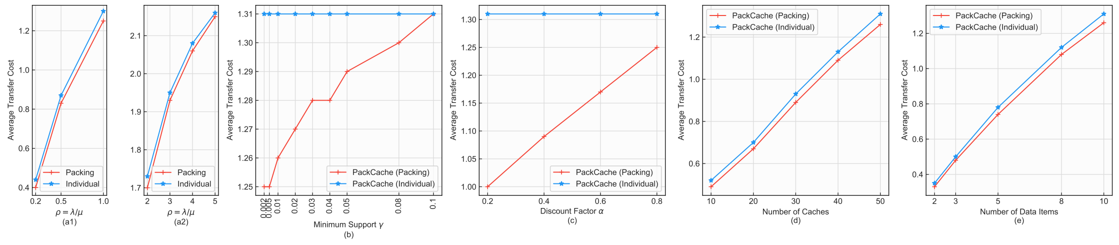

Fig. 2. The average transfer cost of *PackCache* algorithm and its individually-served counterpart under (a1) & (a2) different ρ ratios, (b) different minimum support γ, (c) different discount factor α, (d) different number of cache servers and (e) different number of data items.

a linear trend as the ρ ratio raises. The reason is that when the ratio ρ raises, the transfer cost becomes higher than the caching cost, which gradually encourages local caching of data items instead of transferring them to serve the requests. Data items will be cached locally for longer periods, the local caching becomes less frequent to expire, leading to less transfers being performed.

5.2 Impact of Minimum Support γ

The effectiveness of the *PackCache* algorithm under different minimum support γ during FP-Tree construction has been illustrated in Fig. 2(b). The minimum support value controls the threshold of whether two data items will be considered frequent or not. The higher the γ is, the more frequently should both data items be co-utilised in order to be considered as frequent. A higher minimum support γ will discourage the discounted packing from being applied and vice versa. Hence, by observing Fig. 2(b), we notice that the average transfer cost of *PackCache* algorithm increases as the minimum support raises. The average transfer cost even approaching its individually-served counterpart when the minimum support becomes relatively large, i.e., 0.1 in this case. This is natural to observe since when the minimum support γ is relatively small, more co-utilised data items will be considered as being frequent and hence the benefit brought by the data packing can be fully exploited. While the minimum support γ becomes higher, less data co-utilisation will be considered to be frequent and hence the data packing benefit gradually diminishes. Therefore, the performance of the *PackCache* algorithm will approach its individual counterpart as the minimum support increases.

Specifically, when the minimum support γ is set to the default value 0.01, the cost reduction achieved by the *PackCache* algorithm is around 3.8%. Furthermore, under all minimum support settings except the extremely high setting, a significant cost reduction has been observed, which demonstrates the effectiveness of the *PackCache* algorithm and its robustness in terms of varied γ settings.

5.3 Impact of Discount Factor α

The performance of *PackCache* algorithm under different discount factor α is shown in Fig. 2(c). The discount factor α controls the benefit that the packing mechanism can bring. The higher the discount factor, the less benefit when utilising the packing mechanism. Hence, as observed from Fig. 2(c), the average transfer cost increases linearly as the discount factor α raises. Despite the raise of the average transfer cost, the *PackCache* algorithm still outperforms its individual counterpart by a large margin under all α settings. For instance, when α is 0.6, the cost reduction achieved by the *PackCache* algorithm is 10.7%. Even for the highest setting 0.8, which is the default value we utilised following 5, the cost reduction is still around 4.6%. Hence, the results demonstrates the excellent performance and robustness of the *PackCache* algorithm.

5.4 Scalability of the Algorithm

To testify the scalability of the *PackCache* algorithm, three variables are adjusted, i.e., number of requests, number of cache servers, and number of data items. The evaluation results are presented in Fig. 3, 2(d) - 2(e), respectively.

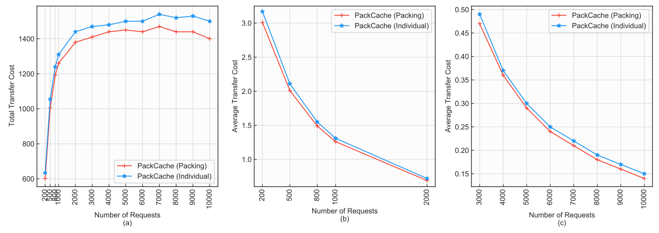

Fig. 3. The total and average transfer cost of the *PackCache* algorithm and its individual counterpart when serving different number of requests.

Scalability (Number of requests): From Fig. 3(a), it is natural that the total transfer cost of request serving rises when the number of requests increases from 200 to 10000. The total transfer cost reaches a plateau when the number of requests becomes even larger. The reason is that when the number of requests becomes larger, the requests will come in a denser manner, making the cached copies hardly expire. Hence, it significantly reduces the need of copy transfer, and hence it also results in the transfer cost stops growing.

In terms of average request transfer cost, a clear decreasing trend is observed from Fig. 3(b) and (c). For instance, when the number of requests is 500 and 2000, the *PackCache* algorithm achieves cost reduction of 4.7% and 4.2%, respectively, which verifies the scalability of the *PackCache* algorithm when serving varied number of requests.

Scalability (Number of cache servers): The performance of the *PackCache* algorithm when working under different number of cache servers has been presented in Fig. 2(d). The average transfer cost increases linearly when the number of cache servers raises. The linear trend indicates that the *PackCache* algorithm scales stably when the number of cache servers varies, demonstrating excellent scalability in terms of different number of cache servers. Under different number of cache servers, the *PackCache* algorithm achieves a cost drop around 4.3%, which further verifies the effectiveness of the *PackCache* algorithm.

Scalability (Number of data items): Finally, the *PackCache* algorithm is evaluated under different number of shared data items. From Fig. 2(e), we can observe a relatively stable increasing trend of the average transfer cost when the number of data items increases. Under all settings, the *PackCache* algorithm constantly achieves superior performance. More specifically, cost saving of 5.7%, 5.1% and 3.8% are achieved when the number of data item is set to be 2, 5 and 10, respectively. The reason why the cost reduction gradually decreases is that as the number of data items increases, more double data item pairs can be formed, hence, less pairs become frequent and hence the packing mechanism is less frequently applied. Therefore, as the number of data items keeps growing, the performance of the *PackCache* algorithm will gradually approach its individually-served counterpart, as the benefit brought by the packing mechanism will gradually diminish. However, when the number of data items are relatively small, we can observe significant cost reduction, which verifies the effectiveness and scalability of the *PackCache* algorithm.

5.5 Comparison with Offline Algorithm

To demonstrate the effectiveness, we also compare the *PackCache* algorithm with its offline counterpart. The cost yielded by the *PackCache* algorithm is 1.04, 1.41, and 1.31 times higher than its offline counterpart when ρ varies between 0.5 and 2. When α varies from 0.6 to 0.8, the cost is 1.13 and 1.41 times higher than the offline version. The *PackCache* achieves costs that are 1.04, 1.27 and 1.41 times higher than its offline counterpart in settings with 10, 30 and 50 servers, and are 1.12 and 1.41 times higher in settings with 5 and 10 data items. Finally, 1.5 and 1.41 times higher cost are produced when the number of requests varied from 500 to 1000. As we can observe, although the performance of *PackCache* is lower bounded by $2/\alpha$ (1.6 under default $\alpha = 0.8$), in various settings, this theoretical lower bound is usually not reached, which demonstrates the effectiveness of the *PackCache* algorithm.

5.6 Performance under Real Cost Model

We use the real service prices to demonstrate that *PackCache* algorithm can achieve a significant cost saving under real deployment. Google Cloud charges $\$0.04$ and $\$0.08$ for caching and transferring a GB of data. We suppose the algorithm daily serves 1000 requests for total 1000GB of data items. Given a reasonable range of discount factor between 0.6 and 0.9 depending on the selected compression technique, we use α of 0.6 and 0.8 as representative scenarios. The system can yield a yearly cost saving of $\$1306.7$ out of $\$11468$ and $\$449$ out of $\$12002$, which is approximately equivalent to reducing 21778 and 7483 GB of data being handled, respectively. Hence, the *PackCache* algorithm is beneficial in real settings.

6 Conclusion

In this paper, we studied a data caching problem in the cloud with cost minimisation as the goal. Given that serving data requests in a packable manner is usually more cost effective than in its individual counterpart (i.e., non-packing), we are among the first to propose a time-space efficient *PackCache* algorithm, which leverages FP-Tree to mine frequently co-utilised data in an online setting and exploits the concept of anticipatory caching for service cost reduction. We showed the algorithm is $2/\alpha$ competitive with respect to a homogeneous cost mode, reaching the lower bound of the competitive ratio for any deterministic online algorithm on this problem. Finally, we evaluated the performance of the algorithm via experimental studies to show its actual cost-effectiveness and scalability in practice.

References

[1] M. Anandarajan, C. Hill, and T. Nolan, “Text preprocessing,” in *Practical Text Analytics*. Springer, 2019, pp. 45–59.

[2] Y. Wang, S. He, X. Fan, C. Xu, J. Culberson, and J. Horton, “Data caching in next generation mobile cloud services, online vs. off-line,” in *Parallel Processing (ICPP), 2017 46th International Conference on*, 2017, pp. 412–421.

[3] Y. Wang, B. Veeravalli, and C.-K. Tham, “On data staging algorithms for shared data accesses in clouds,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 4, pp. 825–838, 2013.

[4] D. Huang, X. Fan, Y. Wang, S. He, and C. Xu, “Dp_greedy: A two-phase caching algorithm for mobile cloud services,” in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10.

[5] M. Bury, C. Schwiegelshohn, and M. Sorella, “Similarity search for dynamic data streams,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 11, pp. 2241–2253, 2019.

[6] Y. Wang, H. Dai, X. Han, P. Wang, Y. Zhang, and C.-Z. Xu, “Cost-driven data caching in edge-based content delivery networks,” *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[7] B. Veeravalli, “Network caching strategies for a shared data distribution for a predefined service demand sequence,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 6, pp. 1487–1497, 2003.

[8] Y. Mansouri, A. N. Toosi, and R. Buyya, “Cost optimization for dynamic replication and migration of data in cloud data centers,” *IEEE Transactions on Cloud Computing*, vol. 99, no. 99, pp. 1–1, 2018.

[9] S. Zhang, P. He, K. Suto, P. Yang, L. Zhao, and X. Shen, “Cooperative edge caching in user-centric clustered mobile networks,” *IEEE Transactions on Mobile Computing*, vol. 17, no. 8, pp. 1791–1805, 2018.

[10] X. Zhang and Q. Zhu, “Collaborative hierarchical caching over 5g edge computing mobile wireless networks,” in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.

[11] J. George and S. Sebastian, “Cooperative caching strategy for video streaming in mobile networks,” in *2016 International Conference on Emerging Technological Trends (ICETT)*, 2016, pp. 1–7.

[12] C. Li, M. Song, C. Yu, and Y. Luo, “Mobility and marginal gain based content caching and placement for cooperative edge-cloud computing,” *Information Sciences*, vol. 548, pp. 153–176, 2021.

[13] A. Khanafer, M. Kodialam, and K. P. N. Puttaswamy, “The constrained ski-rental problem and its application to online cloud cost optimization,” in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 1492–1500.

[14] K. P. Puttaswamy, T. Nandagopal, and M. Kodialam, “Frugal storage for cloud file systems,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 71–84.

[15] A. Erradi and Y. Mansouri, “Online cost optimization algorithms for tiered cloud storage services,” *Journal of Systems and Software*, vol. 160, p. 110457, 2020.

[16] R. Moulton and Y. Jiang, “Maximally consistent sampling and the jaccard index of probability distributions,” in *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2018, pp. 1094–1103.

Jiashu Wu received BSc. degree in Computer Science and Financial Mathematics & Statistics from the University of Sydney, Australia (2018), and M.IT degree in Artificial Intelligence from the University of Melbourne, Australia (2020). He is currently pursuing his Ph.D at the University of Chinese Academy of Sciences (Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences). His research interests including transfer learning and cloud computing.

Hao Dai received the BS and M.Sc. degrees in Communication and Electronic Technology from the Wuhan University of Technology in 2015 and 2017, respectively. He is currently working toward the Ph.D. degree in the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include mobile edge computing, federated learning and deep reinforcement learning.

Yang Wang received the BSc degree in applied mathematics from Ocean University of China, in 1989, and the M.Sc. degree in computer science from Carleton University, in 2001, and the Ph.D degree in computer science from the University of Alberta, Canada, in 2008. He is currently in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, as a professor. His research interest includes cloud computing, big data analytics, and Java virtual machine on multicores. He is an Alberta Industry R&D Associate (2009-2011), and a Canadian Fulbright Scholar (2014-2015).

Yong Zhang received his Ph.D. in the Department of Computer Science and Engineering at Fudan University in 2007. He is now a Professor in SIAT, CAS, Honorary Professor at the University of Hong Kong. Before joining SIAT, he worked as Post-Doctoral Fellow and Senior Researcher in TU-Berlin and HKU. He has published more than 100 papers in refereed journals and conferences. His research interests include design and analysis of algorithms, combinatorial optimization, and wireless networks.

Dong Huang received the M.Eng degree from University of Chinese Academy of Sciences (Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences) in 2019. His research interest includes cloud computing.

Chengzhong Xu received the Ph.D. degree from the University of Hong Kong in 1993. He is currently the Dean of Faculty of Science and Technology, University of Macau, China, and the Director of the Institute of Advanced Computing and Data Engineering, Shenzhen Institute of Advanced Technology of Chinese Academy of Sciences.His research interest includes parallel and distributed systems and cloud computing. He has published more than 400 papers in journals and conferences. He serves on a number of journal editorial boards, including IEEE TC, IEEE TPDS, IEEE TCC, JPDC and China Science Information Sciences. He is a fellow of the IEEE.