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 in $2/\alpha$ 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—Cloud computing, competitive ratio, complexity analysis, data caching, data packing

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, thus they 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.

We study the problem of 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], [3], [7], 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 $\alpha$ 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 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 $\alpha$ 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, Khanfer 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 \( D = \{d_1, d_2, \ldots, d_k\} \), which will be cached in a fully connected network with m cache servers, denoted by \( S = \{s_1, s_2, \ldots, s_m\} \). A sequence of data requests, \( R = \{r_1, r_2, \ldots, r_r\} \), are made to request these data items, where the tuple \( r_i = <s_i, t_i, D_i> \) represents that request \( r_i \) is made at server \( s_i (s_i \in S) \) 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., \( D_i = \{d_j\}, d_j \subseteq D \), or can request two data items, i.e., \( D_i = \{d_{j1}, d_{j2}\}, d_{j1}, d_{j2} \subseteq D \).

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], [3], [4], [7], so we can use \( t \) to represent the arrival time of the request \( r \).

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 \( \alpha, \alpha \leq 1 \), 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.

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 \( \mu \), and the transfer costs between any pair of servers are also identical, denoted by \( \lambda \). Suppose \( C^p_{ij} (1 \leq p \leq k) \) represents the cost to serve a data item \( d_j \) contained in request \( r_j = \langle s_j, t_j, D_j \rangle \), i.e., \( d_j \in D_j \) and request \( r_j = \langle s_j, t_j, D_j \rangle \), is the most recent request for data item \( d_j \) before \( r_j \). If \( r_j \) is the first request for data \( d_p \), \( t_1 = 0 \). We give a formal definition of \( C^p_{ij} \) as follows:

\[
C^p_{ij} = \begin{cases} 
(t_j - t_i)\mu + \epsilon \lambda, & t_j > t_i, \\
+\infty, & \text{Otherwise}
\end{cases}
\]

(1)

where \( \epsilon \) is a variable with value in \([0, 1]\) to signify how to treat the transfer cost during the computation of \( C^p_{ij} \). Specifically, if the request \( r_j \) is served by a local data cache resulted from \( d_j \), i.e., \( s_j = s_i \), then \( \epsilon = 0 \). Otherwise, \( \epsilon = 1 \), implying the requested data item \( d_j \) is first cached from \( s_j \) to \( s_i \) on server \( s_i \), and then transferred from \( s_i \) to \( s_j \) to serve request \( r_j \). If \( d_j \) 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 \( \lambda \) will have the discount factor \( \alpha \) being applied, i.e., \( \alpha \lambda \). 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 generality, we assume these costs are free as in many previous studies [2], [3], [4], [7].

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 \( \phi_i(R) \) as the optimal for this \( n \)-length request sequence, i.e., \(|R| = n \). Each schedule \( \phi_i(R) \) has a cost \( cost(\phi_i(R)) \), which is defined as follows:

\[
\text{cost}(\phi_i(R)) = \sum_{r_j \in R} \text{cost}(r_j) = \sum_{r_j \in R} \sum_{d_j \in D_j} \text{cost}(d_j),
\]

(2)

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

\[
\phi_i(R) = \arg \min_{\phi(R) \in \Pi(i)} \text{cost}(\phi_i(R)),
\]

(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 3. Function serve_request(rk)

Input: an incoming request in form rK = < si, tk, Di >
Output: the transfer cost of serving this request
1: if request rK arrives at time tk
2: if (request rK contains a single data item dk) then
3: else if (E[di] = 0) then
4: serve rK by the local copy on sj
5: else if (E[di] ≠ 0) then
6: serve request rK, by local copy on sj
7: E[di] ← tK + Δt
8: c[di] ← c[di] + 1
9: else if (E[di] = 0 and E[dj] = 0) then
10: serve request rK, by the local copy on sj
11: E[di] ← tK + Δt
12: c[di] ← c[di] + 1
13: else if (E[di] ≠ 0 and E[dj] ≠ 0) then
14: rK serves request rK, by local copy on sj
15: E[di] ← tK + Δt
16: c[di] ← c[di] + 1
17: if transferred individually then
18: CTP ← CTP + 2λ
19: else
20: CTP ← CTP + 2αλ
21: end if
22: dK & dJ 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.
23: if transferred individually then
24: CTP ← CTP + 2λ
25: else
26: CTP ← CTP + 2αλ
27: end if
28: else if (request rK contains two data items di and dj) then
29: dK & dJ 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.
30: if transferred individually then
31: CTP ← CTP + 2λ
32: else
33: CTP ← CTP + 2αλ
34: end if
35: else if (E[di] = 0 and E[dj] = 0) then
36: serve request rK, by the local copy on sj
37: E[di], E[dj] ← tK + Δt
38: c[di] ← c[di] + 1, c[dj] ← c[dj] + 1
39: else if (E[di] ≠ 0 and E[dj] ≠ 0) then
40: serve request rK, by local copy on sj
41: E[di], E[dj] ← tK + Δt
42: c[di] ← c[di] + 1, c[dj] + 1
43: end if
44: else
45: dK & dJ are missed locally, trans. from sk, k ≠ j;
46: CTP ← CTP + λ
47: end if
48: end if
49: rK serves request rK, by local copy on sj
50: E[di], E[dj] ← tK + Δt
51: c[di] ← c[di] + 1, c[dj] + 1
52: if transferred individually then
53: CTP ← CTP + 2λ
54: else
55: CTP ← CTP + 2αλ
56: end if
57: else if (E[di] = 0 and E[dj] = 0) then
58: serve request rK, by the local copy on sj
59: E[di], E[dj] ← tK + Δt
60: c[di] ← c[di] + 1, c[dj] + 1
61: else if (E[di] ≠ 0 and E[dj] ≠ 0) then
62: serve request rK, by the local copy on sj
63: E[di], E[dj] ← tK + Δt
64: c[di] ← c[di] + 1, c[dj] + 1
65: end if
66: end if
67: return CTP

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 rK is updated.

Algorithm 4. Function copy_expire(dK, sj, tk)

Input: the copy expiration event information
Output: no output will be yielded by this function
1: if (c[di] = 1) then
2: E[di] = tK + Δt
3: if (c[di] = 1) then
4: drop the dK copy at sj
5: E[di] = 0
6: c[di] = c[di] - 1
7: end if
8: end if

For the copy expiration handler, we adopt the mechanism inspired by the anticipatory caching concept. The period Δt = λ/μ is calculated as shown in line 1 in Algorithm 4 and is also used in Algorithm 4. 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[di] > 1, i.e., there exists more than one alive copies of data item di, then no copy can survive for more than Δt on any server sj, j ∈ [1, m].

Observation 2. Since the algorithm has no knowledge about future requests, in the worst case, it needs to cache di after satisfying its request for Δt period. However, if no subsequent request of di 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 3/2-competitive.

Proof. In the proof, we denote COPT as the cost of serving data request rK by the proposed PackCache algorithm, and denote CDP 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 rK = < sj, tk, {dk} > is the first request that arrives at server sj, j ∈ [1, m], then we have CDP = μΔt + λd and CDP = λ. Hence we have CDP/CDP = 2 ≤ 2/3, given that 0 < A ≤ 1.
Case 2. When the request $r_1 = \langle s_1, t_1, \{d_i\} \rangle$ arrives, two scenarios should be considered as follows:

Case 2.1. If $t_1 \in \{r_1' \in \mathbb{R} : r_1' < t_1 \}$ and $E[d_i] \neq 0$, which means there is a local copy ready to serve the request and the caching period is within $\Delta_f$, therefore we have $C_{ON}^{OPT} = \mu \Delta_f \leq 2\lambda$, $C_{OPT} = \mu \Delta_f \leq 2\lambda$, hence $C_{ON}^{OPT} / C_{OPT} = 1 < 2\alpha$.

Case 2.2. If $t_1 > r_1' < \mathbb{R} > t_1 + \Delta_f$ and $E[d_i] = 0$, i.e., there exists no local cache of the data item to server the request. Then the transfer will occur, i.e., $C_{ON}^{OPT} = \mu \Delta_f + 2\lambda$, $C_{OPT} = \mu \Delta_f$, hence we have $C_{ON}^{OPT} / C_{OPT} = 2 < 2\alpha$.

Now let's 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 $0 < \alpha < 1$ can be applied during data item transferring, otherwise, the discount factor is not applicable. First, 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 $C_{ON}^{OPT} = 2\mu \Delta_f + 2\lambda = 4\lambda$. On the other hand, $C_{OPT} = 2\alpha \lambda$. Hence, $C_{ON}^{OPT} / C_{OPT} = 2\alpha$.

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., $E[d_i] = 0$, then we have $C_{ON}^{OPT} = 2\mu \Delta_f + 2\lambda = 4\lambda$, while $C_{OPT} = 2\alpha \lambda$. Hence, we have $C_{ON}^{OPT} / C_{OPT} = 2\alpha$.

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 $C_{ON}^{OPT} = \mu \Delta_f + 2\lambda = 2\lambda$, while $C_{OPT} = \lambda$. Therefore, we have $C_{ON}^{OPT} / C_{OPT} = 2 < 2\alpha$ given that $0 < \alpha \leq 1$.

Since the $C_{ON}^{OPT}$ is the cost of serving request $r_1$, same for $C_{OPT}$ which is the cost of the corresponding optimal algorithm, therefore, for the entire request sequence $R$, $|R| = n$.

$$
\frac{C_{ON}^{OPT}}{C_{OPT}} = \sum_{1 \leq i \leq n} \frac{C_{ON}^{OPT}}{C_{OPT}} = \sum_{1 \leq i \leq n} \frac{C_{ON}^{OPT}}{C_{OPT}}.
$$

Hence, we can conclude the theorem with the following result

$$
\lim_{n \to \infty} \frac{C_{ON}^{OPT}}{C_{OPT}} = \frac{2}{\alpha}.
$$

The following theorem shows that there does not exist any deterministic online algorithm that can yield a better performance than $\frac{2}{\alpha}$ 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 $\lambda$ and $\mu$ are set to 1, and thus $\Delta_f$ is 1. Initially, at least one server has both data items, and at least one server has each single data item. We define $l_1$ to be length of local caching after satisfying request $r_1$. 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 $r_1 = \langle s_1, 0, \{d_i, d_j\} \rangle$ 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:

1. If $l_1 = 1$, then there is no subsequent requests comes after $r_1$, which yields $\frac{C_{ON}}{C_{OPT}} = \frac{2\mu \Delta_f}{\alpha \lambda} = \frac{2}{\alpha}$.

2. If $l_1 < 1$, it means there is a request comes in $\epsilon$, i.e., there exists another request $r_2$, hence, we have $C_{ON}^{OPT} = \mu \Delta_f + \lambda\epsilon$, $C_{OPT} = \mu \Delta_f$, hence we have $C_{ON}^{OPT} / C_{OPT} = \frac{2}{\alpha}$ holds, the 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 4-4 in Section 4. In terms of the space complexity, maintaining $E[d_i]$ 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 $E[d_i]'$, $E[d_i]$ and $r_1', r_2', \ldots$ 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 $\alpha$, the minimum support threshold $\gamma$ for the FP-Tree frequent data itemset mining, the caching cost $\mu$ and the transfer cost $\lambda$. 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 \((d_1, d_2, \ldots, d_{10})\) 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 \(\lambda\) and caching cost \(\mu\) 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 \(\rho\) ratios, we set \(\lambda + \mu = 6\) intentionally. The experimental results of the transfer cost is illustrated in Fig. 2a.

According to Fig. 2a, the average transfer cost increases as the ratio \(\rho\) increases. It is natural to observe because as the ratio \(\rho\) increases, the transfer cost \(\lambda\) is gradually emphasised while the caching cost \(\mu\) is gradually declined. Under all \(\rho\) 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 \(\rho\) 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 a linear trend as the \(\rho\) ratio raises. The reason is that when the ratio \(\rho\) 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 \(\gamma\)

The effectiveness of the PackCache algorithm under different minimum support \(\gamma\) during FP-Tree construction has been illustrated in Fig. 2b. The minimum support value controls the threshold of whether two data items will be considered frequent or not. The higher the \(\gamma\) is, the more frequently should both data items be co-utilised in order to be considered as frequent. A higher minimum support \(\gamma\) will discourage the discounted packing from being applied and vice versa. Hence, by observing Fig. 2b, 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 \(\gamma\) 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 \(\gamma\) 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 \(\gamma\) 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 \(\gamma\) settings.

### 5.3 Impact of Discount Factor \(\alpha\)

The performance of PackCache algorithm under different discount factor \(\alpha\) is shown in Fig. 2c. The discount factor \(\alpha\) 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. 2c, the average transfer cost increases linearly as the discount factor \(\alpha\) raises. Despite the raise of the average transfer cost, the PackCache algorithm still outperforms its individual counterpart by a large margin under all \(\alpha\) settings. For instance, when \(\alpha = 0.6\), 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 Figs. 3, 2d and 2e, respectively.

**Scalability (Number of requests):** From Fig. 3a, 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 Figs. 3b and 3c. 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. 2d. 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. 2e, 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 5.1%, 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 is 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 $\alpha$ varies between 0.5 and 2. When $\alpha$ varies from 0.6 to 0.8, the cost is 1.13 and 1.41 times higher than the offline version. The PackCache algorithm 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.

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, Berlin, Germany: 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 Proc. 46th Int. Conf. Parallel Process. 2017, pp. 412–421.
[3] Y. Wang, B. Veeravalli, and C.-K. Tham, “On data staging algorithms for shared data accesses in clouds,” IEEE Trans. Parallel Distrib. Syst., vol. 24, no. 4, pp. 825–838, Apr. 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 Proc. IEEE Int. Conf. Cluster Comput., 2019, pp. 1–10.
[5] M. Bury, C. Schwingelshohn, and M. Sorella, "Similarity search for dynamic data streams," IEEE Trans. Knowl. Data Eng., vol. 32, no. 11, pp. 2241–2253, Nov. 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 Trans. Mobile Comput., to be published, doi: 10.1109/TMC.2021.3108130.
[7] B. Veeravalli, “Network caching strategies for a shared data distribution for a predefined service demand sequence,” IEEE Trans. Knowl. Data Eng., vol. 15, no. 6, pp. 1487–1497, Nov. /Dec. 2003.
[8] Y. Mansouri, A. N. Toosi, and R. Buyya, “Cost optimization for dynamic replication and migration of data in cloud data centers,” IEEE Trans. Cloud Comput., vol. 7, no. 3, pp. 705–718, Jul.–Sep. 2019.
[9] S. Zhang, P. He, K. Suto, P. Yang, L. Zhao, and X. Shen, “Cooperative edge caching in user-centric clustered mobile networks,” IEEE Trans. Mobile Comput., vol. 17, no. 8, pp. 1791–1805, Aug. 2018.
[10] X. Zhang and Q. Zhu, “Collaborative hierarchical caching over 5G edge computing mobile wireless networks,” in Proc. IEEE Int. Conf. Commun., 2018, pp. 1–6.
[11] J. George and S. Sebastian, “Cooperative caching strategy for video streaming in mobile networks,” in Proc. Int. Conf. Emerg. Technol. Trends, 2016, pp. 1–7.
[12] S. Li, M. Song, C. Yu, and Y. Luo, “Mobility and marginal gain based content caching and placement for cooperative edge-cloud computing,” Int. J. Inf. Sci., vol. 548, pp. 153–176, 2021.
[13] A. Kanafer, M. Kodialam, and K. P. N. Puthusserywamy, “The constrained skitrental problem and its application to online cloud cost optimization,” in Proc. IEEE INFOCOM, 2013, pp. 1492–1500.
[14] K. P. Puthusserywamy, T. Nandagopal, and M. Kodialam, “Frugal storage for cloud file systems,” in Proc. 7th ACM Eur. Conf. Comput. Syst., 2012, pp. 71–84.
[15] A. Erradi and Y. Mansouri, “Online cost optimization algorithms for tiered cloud storage services,” Inf. Sci., vol. 160, 2012, pp. 71–84.
[16] C. Li, M. Song, C. Yu, and Y. Luo, “Mobility and marginal gain based content caching and placement for cooperative edge-cloud computing,” Int. J. Inf. Sci., vol. 548, pp. 153–176, 2021.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.