Chen and Tan *EURASIP Journal on Wireless Communications and Networking* (2018) 2018:268
https://doi.org/10.1186/s13638-018-1268-1

EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

A Q-learning-based network content caching method

Haijun Chen[1,2*] and Guanzheng Tan[1]

Abstract

Cloud computing provides users with a distributed computing environment offering on-demand services. As its technologies become gradually mature and its application becomes more universal, cloud computing greatly reduces users' costs while increasing working efficiency of enterprises and individuals (Futur Gener Comput Syst 25:599–616, 2009). Software as a service (SaaS), as a kind of information servicing model based on cloud platforms, is rising with the developments of Internet technologies and the maturing of application software. The responsibility of a SaaS server is to timely and accurately satisfy users' needs for information. An intelligent and efficient content caching solution or method plays a vital role in that. This paper proposes a reinforcement learning (RL)-based content caching method named time-based Q Cacher (TQC) which effectively solves the problem of low hit ratio of server caching and ultimately achieves an intelligent, flexible, and highly adaptable content caching model.

1 Introduction

Cloud computing is regarded as a revolution in enterprise application deployment and software configuration. Traditional software sales model requires users to purchase, deploy, use, and maintain software permanently. However, this puts high requirements on users' technical experience and initial costs. At the same time, software also needs to be regularly updated and maintained, which increases users' investment in manpower and material resources.

Internet, through Web services, provides enterprises and individuals with an innovative business model and flexible calculation modes. With the emergence of software as a service (SaaS), applications are moving away from PC-based or ownership-based programs to Web delivered-hosted services [1]. The software services are provisioned on a pay-as-you-go basis to overcome the limitation of the traditional software sales model. Due to its flexibility, scalability, and cost-effectiveness, SaaS model has been increasingly adopted for distributing enterprise software systems, such as banking and e-commerce business software [2, 3]. The number of SaaS services is available in the markets such as Twitter, Gmail, Saleforce.com, and Google Maps to configure SaaS-based Web service systems.

To use cloud computer server facility, it is much faster, more reliable solution than private client-server model (c/s). This is the biggest factor to use cloud computer server and run SaaS today [4]. SaaS is also involving PaaS and IaaS. But there are several issues to achieving these requirements, scalable, configurable, multi-tenant-efficient SaaS model. To meet these requirements, SaaS infrastructure is required to have scale-out architecture with data interoperable function without any data contamination. It also needs short latency of time service because Web service is online and follows the remote client-server model. If service response time does not meet these timeliness requirements, users will not use SaaS application services again.

In addition, Web-based applications also need more intensive I/O access, putting higher requirements on I/O performance. To meet this requirement, we are facing issues of how to create effectiveness for the total computer system with amount of data and how to reduce computer facilities in data center. To do this, available memory space is the big performance factor. If memory space on server is insufficient, guest OS and application programs will be flushing between memory and storage. Once VMM server encounters this situation, the VMM environment would become very slow [4]. Wilhelm et al. [5] prove that rationally using caching technologies is necessary and important for reducing worst-case execution times (WCETs) and improving system performance.

* Correspondence: 18710822156@163.com
[1]Department of School of Information Science & Engineering, Central South University, Changsha, China
[2]Hunan University of Commerce, Changsha, China

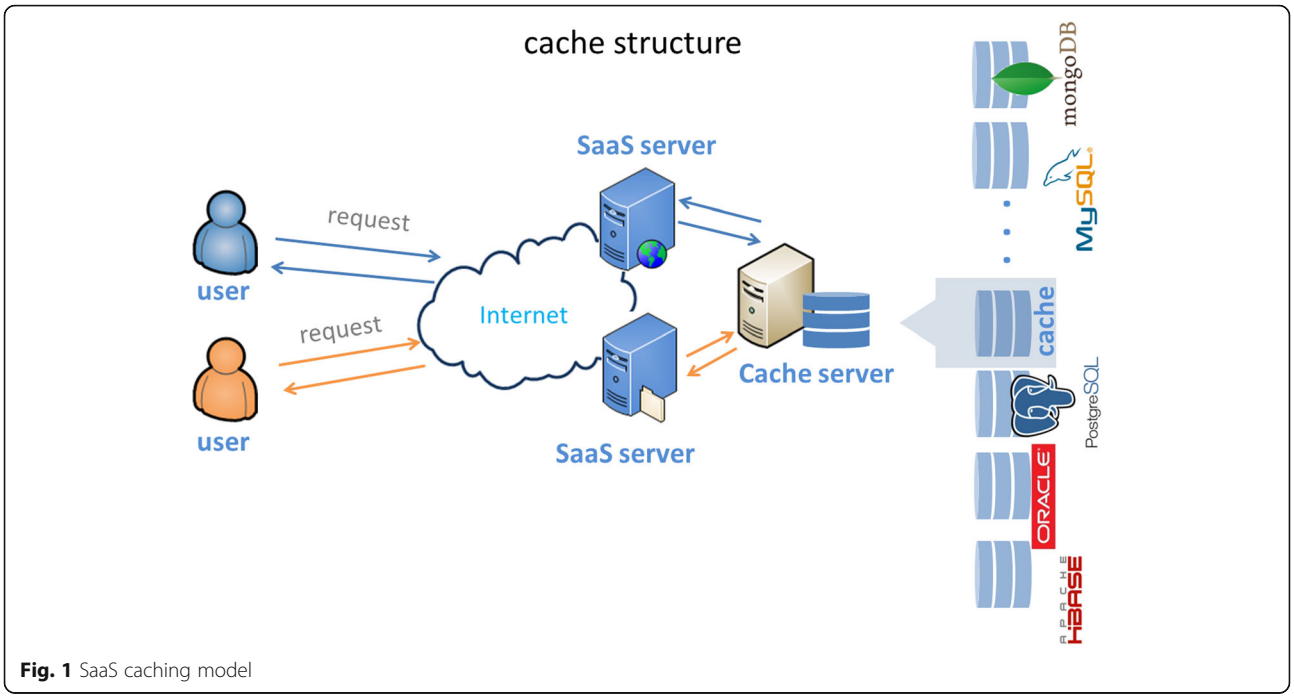

Fig. 1 SaaS caching model

SaaS caching model for serving customers in cloud is shown in Fig. 1. SaaS caching framework can be divided into four parts: user, SaaS server, caching server, and storage system. As shown in Fig. 1, users periodically send data acquisition requests to SaaS services via the Internet. The requests are diverse, dynamic, in large quantity, and random. Based on a cloud platform, SaaS server provides information acquisition service for users in a unified, on-demand, and efficient way. The server first listens to and analyses users' requests and then checks whether the cache server has cached the data that users need to acquire. If it has not, the SaaS server then sends the query request

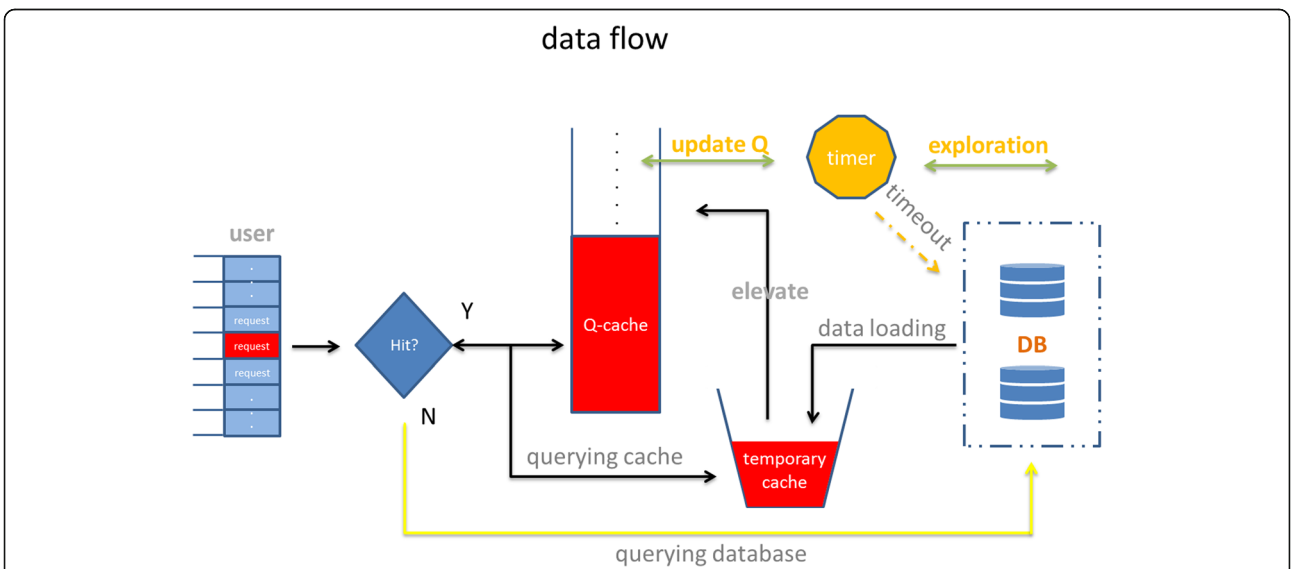

Fig. 2 TQC algorithm structure. On the one hand, TQC listens to the user request to determine whether the cache is hit. **a** If it is hit, the result will be returned directly to the user and the Q value of the action will be updated. **b** If it is missed, the data in the file system or DB will be loaded to the temporary cache and one or several pieces of data with the smallest Q value in the temporary cache will be replaced. On the other hand, TQC periodically (at a specific stride) adds data to the temporary cache from the list of known contents, replaces one or several files with the smallest Q value, and then observes the change of cache hit ratio so as to allow TQC to learn user requests with low frequency and long period and achieve direct hits

Table 1 File type and size (MB)

Type	Size (MB)	M
Minimum	1	20,000
Smaller	8	20,000
Average	16	20,000
Larger	32	20,000
Maximum	64	20,000

to the database. The cache server replaces file system or database to respond to the data query request sent to the SaaS server so as to increase the response speed of SaaS platform. An important indicator of the cache server is cache hit ratio. In such a caching system, one of the most important design decisions is the content replacement policy that determines which content is to be replaced when the cache is full. The storage system stores all data of the SaaS platform. In SaaS environments, databases are typically relational or non-relational such as Oracle, MySQL, PostgreSQL, mongoDB, and Hbase.

Meanwhile, Wang et al. [6] analyzed where to cache, what to cache, and how to cache. They point out that caching policies, deciding what to cache and when to release cache, are crucial for overall caching performance. The increases in the number and types of SaaS service requests not only put higher requirements on cache hit ratio, adaptability, and scalability, but also bring severe challenges to first in first out (FIFO), least recently/frequently used (LRFU), and other traditional caching algorithms.

Given the important role of cache in increasing the response speed of SaaS platform and the shortcomings of current caching algorithms, the focus of this paper is on exploring policies to maximize the cache hit ratio and minimize the required latency. To achieve this, this paper, based on Q-learning, implements an intelligent caching policy, which we call time-based Q cacher (TQC). The specific contributions of this paper are as follows:

a) Realizes an intelligent and model-free cache updating and replacement policy.
b) Provides a cache hit ratio 8–12% higher than that of current algorithms with the same cache size.
c) The algorithm not only has short-term memory capability but can also learn the pattern of users' long-term and low-frequency requests.
d) The algorithm can not only passively learn the caching policy based on users' requests but also autonomously load data into cache and generate a caching policy by observing the cache hit ratio.

Table 2 Simulation parameter settings

Duration	T	24 h
Interval	T	1 h
Cache capacity	C	32 to 64 GB
File request rate	B	250 to 500 times/min

The rest of this paper consists of the following parts: the second part, which introduces the caching algorithm and the caching model; the third part, which introduces the TQC algorithm; the fourth part, which evaluates the performance of the algorithm; and the fifth part, which provides a summary and the future research direction.

2 Related work

Common methods used to increase the cache hit ratio in reality are mainly improved based on FIFO and LRFU [7] algorithm proposed by Lee et al. Wang et al. [8] propose a practically feasible centrality-based heuristic method that does not depend on global content distribution information as required by the optimal solution. The algorithm searches for the latest caching policy based on shortest path tree (SPT) and in a heuristic manner.

Psaras et al. [9] propose the ProbCache algorithm which approximates the caching capability of a path and caches contents probabilistically to leave caching space for other flows sharing (part of) the same path and to fairly multiplex contents in caches along the path from the server to the client. ProbCache considers each path of caching entities as a pool of caching resources and tries to find optimal ways of distributing content in these caches. Jeswani et al. [10] propose the DiffCache algorithm which computes a cache composition with the objective of minimizing transfers from repository, thereby reducing request service time. The algorithm is based on the phenomenon that image templates often have high degree of commonality. They exploit the presence of this commonality among template files to generate different files or patches between two templates. A patch file can be applied on another template to generate a new template. Instead of caching large templates, they can cache patches and templates and effectively cater to a larger set of template requests by paying a small cost of patching time, while saving the time to fetch the complete template file from the repository. Neumann et al. [11] propose the hybrid cloud storage framework (HCSF) which includes cache synchronization with table storage and provides cloud application developers with single point of data access. Their article demonstrates how the data consistency and persistence of tabular storage can be combined with a volatile but fast distributed cache, while adhering to the CAP theorem.

Identifying and predicting user's behavior and interested content are often more valuable than simply providing services to users. Traditional caching decisions are driven by user requests. Statistical decision-making based on the popularity, size, type, and location of

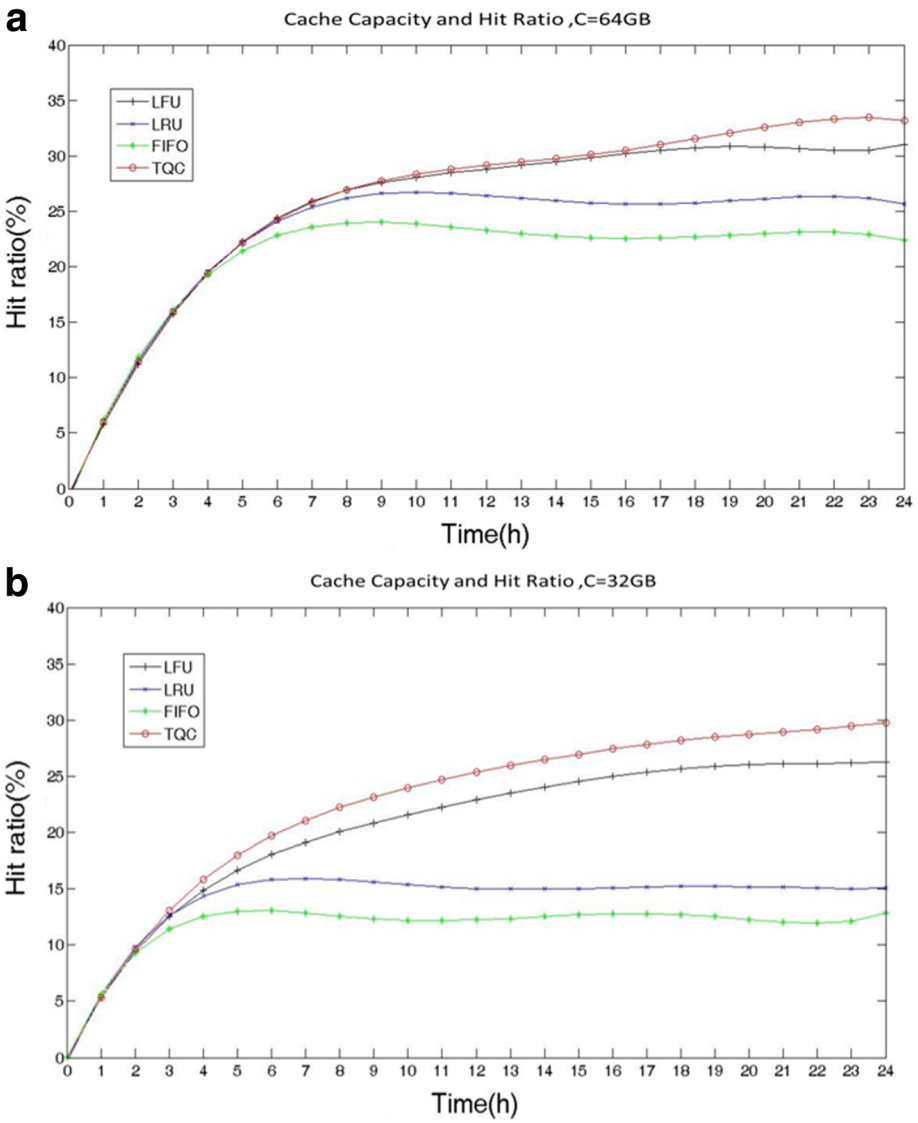

Fig. 3 Effect of cache capacity on hit ratio. The figure shows the comparison for the cache hit ratios of different policies when request rate is 500 times/min and the cache capacity *C* is 64 GB and 32 GB respectively. It can be seen that LRU, LFU, FIFO, and TQC policies all have the same hit ratios when the cache is not full. When t_full = 4.5 h, *C* = 64 GB, t_full = 2.5 h, *C* = 32 GB, and the cache is full, different replacement policies start replacing the contents. **a** and **b**, consequently, the hit ratios of LRU, TQC, and LFU begin to increase and that of FIFO gradually decreases. In general, the performance of TQC is better than the other three policies. When there is hotspot data, the efficiency of LRU is very good. But sporadic and periodic batch operations will lead to a sharp drop of LRU hit ratio and a serious pollution of the cache. FIFO does not take into account the characteristics of data popularity, so its hit ratio reaches peak of 25% and 12.5% at t_full, then begins to decline and fluctuates around 22.5% and 12.5%. Although the hit ratios of the two are not very different, compared with TQC, LFU records all the access counts of files and as the cache capacity increases, LFU needs to cost more. Therefore, TQC saves more time than LFU and the hit ratio of TQC is also about 12% higher than that of traditional methods

content has high requirements on hardware resources and poor flexibility. Its algorithm does not offer long-term memory and intelligence. Recently, Mnih et al. [12] combine deep learning and reinforcement learning algorithms, using game video images (high-dimensional sensory information) and game scores as inputs to simulate a human player playing the Atari 2600 game. In the absence of explicitly defined game rules and human experience, the intelligent agent, through studying human behavior, ultimately attains the level of human professional players and surpasses all similar algorithms. Besides, Silver et al. [13] propose a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, their program AlphaGo achieved a 99.8% winning rate against other Go programs and defeated the human European Go champion by 5 games to 0. The above two achievements have aroused great concern of AI researchers.

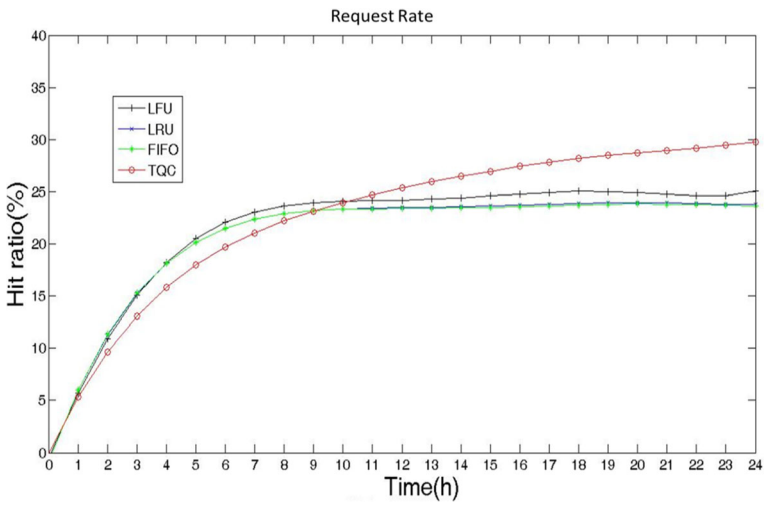

Fig. 4 Request rate evaluation. The figure shows the hit ratios of different policies when cache capacity C is 32 GB and request rate is 250 times/min

Heess et al. [14] study the partially observable state problem in reinforcement learning. They extend two related, model-free algorithms for continuous control-deterministic policy gradient and stochastic value gradient to solve partially observed domains using recurrent neural networks trained with back-propagation through time, to solve challenging memory problems such as the Morris water maze. Blundell et al. [15] restrict the size of Q table by removing the entries of the least and recently accessed updates as soon as a large state sequence is received. Meanwhile, they combine the non-parametric nearest-neighbors model and solve the two problems that reinforcement learning consumes large amounts of memory and lacks a way to generalize across similar states. Schaul et al. [16] make improvements based on the fact that experience transitions were uniformly sampled from a replay memory [17] and propose the prioritized experience replay (PER) method. They applied the two methods to DQN [13] algorithm and made comparison. The results showed that the performance of PER was far better than common experience replay algorithm.

Our work is close to that of Chiocchetti et al. [18]. In dynamic network conditions, they made several improvements to the Q-routing algorithm in order to apply it to various network contexts. Their article achieved a distributed reinforcement learning based on the reward information exchanged between routers in the network so as to improve the cache hit ratio. Caarls et al. [19], combining the Q-routing algorithm and MEC algorithm, propose the Q-caching algorithm aiming at the minimum user download time. They exploit the fact that Q-routing computes the cost-to-go and use it to make not only routing but also caching decisions in a weighted least frequently used (WLFU) manner, evicting the item with the minimum expected cost (MEC) to retrieve. Then, Q-routing and MEC are combined in order to efficiently route requests and store content with the goal of minimizing the download time experienced by users.

Based on user's location, Q-routing and Q-caching passively calculate the frequency of content access and generates the Q table. In reality, acquisition of SaaS data by users or applications is often in large amounts, diverse, low-frequency, and in long periodicity, for example, acquiring SaaS content in every 15 min or a longer period. Making caching or replacement decisions driven by user access behaviors and calculating the recent frequency of content access put serious challenges on computing performance and storage space and even are impossible. To solve the problem of caching solution by directly calculating and analyzing user behaviors, we propose viewing from caching server, using Q-learning to improve cache hit ratio, and letting caching server execute a specific action in a specific time in order to indirectly learn user behaviors and interested content.

3 Methods/experimental

Q-learning algorithm was proposed by Watkings and Dayan. It uses a direct approximation approach to solve Bellman's optimal equation in order to obtain the optimal value function [20]. Rummery and Niranjan made some modifications to Q-learning and proposed the state action reward state action (Sarsa) algorithm [21]. Both Q-learning and Sarsa interact with environment ε by constructing an agent. In a specific environment status, the agent generates a specific action and the environment returns different rewards to the agent. The purpose of training an agent is to maximize its reward.

In this section, we describe the design of the TQC caching policy. We make decisions on content caching and releasing by manipulating the cache space of the

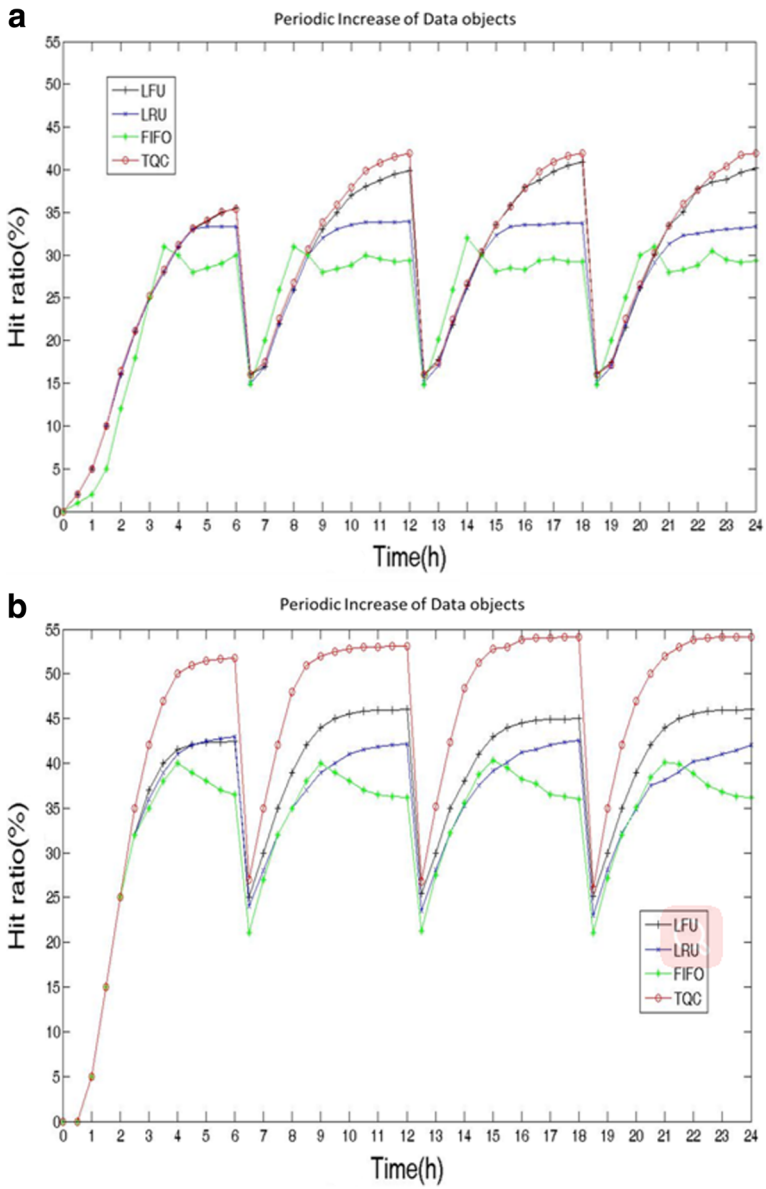

Fig. 5 Periodic increase of data objects. **a** The changes of hit ratio at cache capacity $C = 64$GB and $C = 32$GB. **3a** shows that when the cache capacity is 32 GB, at $t = 6$, 12, and 18 h, each time new data is generated, LFU robustness is the best and TQC efficiency will decline, because TQC needs some time to learn new files. **b** indicates that TQC outperforms LFU over a long period of time when the cache increases to 64 GB. This is mainly because LFU accumulates a large number of access requests for a long time which requires more time to judge new data and replace old data. Based on the learning mode of time series, with very little storage space and computing resources, TQC can learn low-frequency and long-period request patterns so as to enhance the overall performance of the algorithm

server. The goal of TQC is twofold: (i) minimize manual involvement in the caching policy and implement an intelligent model-free caching policy and (ii) increase the cache hit ratio and reduce the time required for users to obtain data from the SaaS platform. In particular, due to the size of the cache, we only add the contents of the group <*time*, *request*, *action*, *hitrate*> with the highest Q value (the highest cache hit ratio) in the table Q to the cache. The TQC periodically updates the Q value of each record in the table Q and replaces the contents of the cache according to the change of the Q value (Fig. 2). In order to guarantee the exploration ability of the algorithm, we divide the cache into two parts. One is 20% and the other one is 80%. Twenty percent of the space stores the requests, missing the contents of the cache. The advantages of this design can speed up the convergence rate of the algorithm and ensure the algorithm's exploration ability.

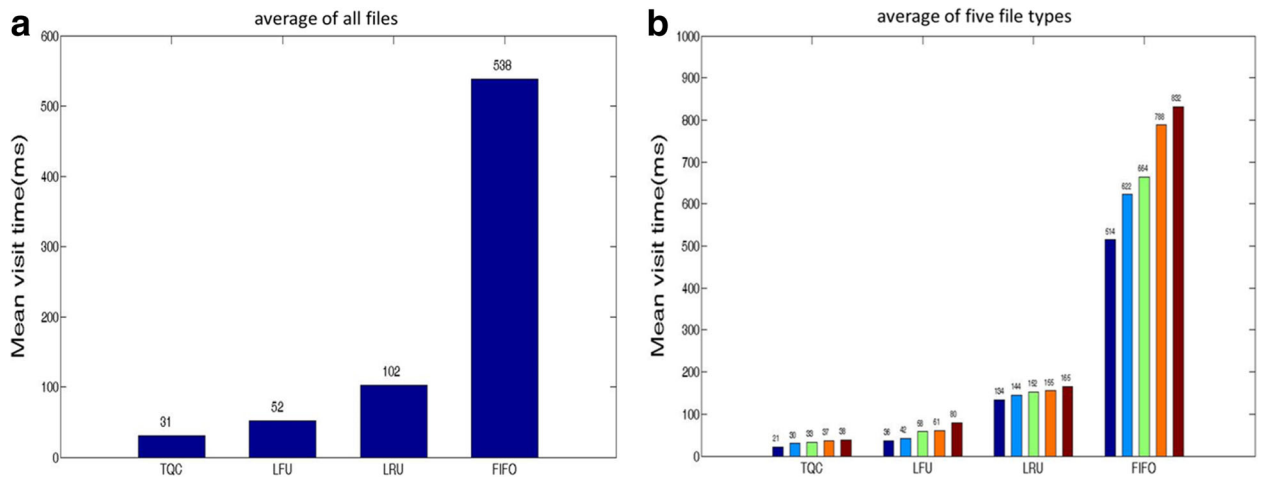

Fig. 6 File average access time. **a** The average response time for four cache policies to retrieve all files. It can be seen that TQC is better than LFU and LRU, and FIFO is the worst policy. **b** The average access time of five file types. The average response time of FIFO for all objects is the highest. And the time cost of LFU and TQC is the smallest. However, TQC has five types whose response time is not more than 50 ms, LFU has two while LRU has none. TQC can eventually achieve a stable state for each file after a long time of learning

TQC uses time series as the basis of learning. By convention, we calculate the hit ratio of the cache at $t+1$ based on the discount factor γ. The calculation formula of the hit ratio is $H_t = \sum_{t'=t}^{T} \gamma^{t'-t} h_{t'}$, $t = stride(s, \tau)$ $t \in oneday$ which denotes the discrete time node with the stride of s and the unit of τ in a certain time range T (1 day). $h = hitrate_t(q, a)$ indicates the cache hit ratio when the user request q is received and the action a is executed at time t. We define an optimal value function $Q_t^*(q, a)$ (i.e., the maximal cache hit ratio) where $data = map(q)$ indicates that the user request q is mapped to an actual data block by the *map* function and a represents the action sequence. The specific formula is $Q_t^*(d, a) = \max_\pi E[H_t | d_t = d, a_t = a, \pi]$ where π is a caching policy which maps the time series and the request sequence to the action sequence $a = \pi(t, q)$. The optimal Q value function indicates that at time t each caching policy selects a valid caching action from the action sequence a for the request sequence q and maximizes the hit ratio of the entire cache. The optimal value function obeys an important and widely used principle (Bellman 1957) of optimality. An optimal policy has the property that the residual decision must be the

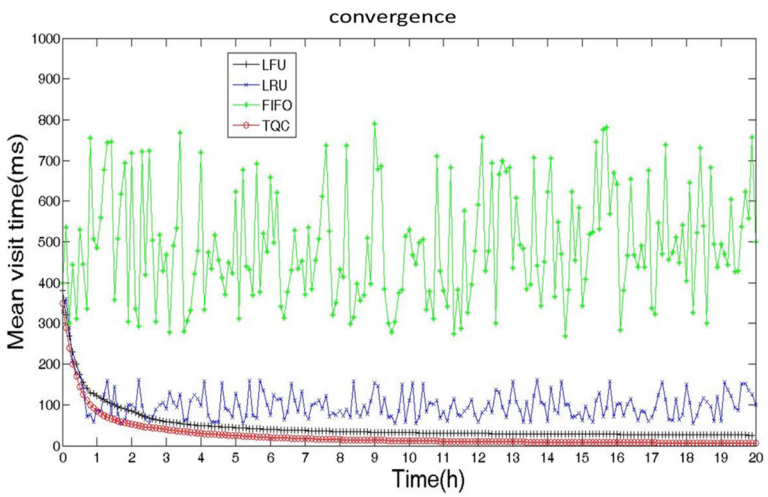

Fig. 7 Variation trend of average file response time over time. The figure shows the changes of average response time for several policies. The average access time for LRU and FIFO is constantly changing due to the variability of the cache contents of LRU and FIFO, resulting in the loss of cache data. Conversely, TQC and LFU do not exhibit this behavior, thus increasing stability. It can be seen that the convergence of TQC is slower than LFU but the average time cost of TQC after convergence is 6 ms and that of LFU is 26 ms. This indicates that the performance of TQC is superior to that of LFU for dynamic changes of file access

optimal policy for the state resulting from the first decision regardless of the initial state and the initial decision. That is, if the optional caching action that the optimal function $Q_t^*(d', a')$ executes for each request is known, the optimal caching policy is to select a caching action sequence that maximizes the subsequent cache hit ratio based on these request states sequences: $h_t + \gamma Q_{t'}{}^*(q', a')$. The optimal Q value function is shown in Eq. 1:

$$Q_t^*(q,a) = E_{q' \sim \varepsilon}\left[h_t(q,a) + \gamma \max_{a'} Q_{t'}{}^*\left(q', a'\right) | q, a\right] \quad (1)$$

In many reinforcement learning algorithms, the estimation of the action-value function is done by iteratively updating the Bellman equation as shown in Eq. 2. Richard Sutton [11] proved that iterative updating can converge to the optimal value function when $Q_t \rightarrow Q^*$ as $t \rightarrow \infty$.

$$Q_{t+1}(q,a) = E\left[hitrate_t(q,a) + \gamma \max_{a'} Q_t\left(q', a'\right) | q, a\right] \quad (2)$$

Based on the Bellman equation, the calculation method of TQC optimal cache hit ratio is shown in Eq. 3 where $count(t, q)$ represents the access counts of user request q at time t:

$$Q_t^*(q,a) = \sum_{t=1}^{T} count(t,q)\left[hitrate_t(q,a) + \gamma \max_{a'} Q^*\left(q', a'\right)\right] \quad (3)$$

Our updating method of the cache hit ratio divides the Bellman equation into instantaneous hit ratio and the sum of historical hit ratios based on the time series and finally multiplies them by the time t and the access counts of the contents corresponding to user request q. The updating method of Formula 4 uses Robins-Monro stochastic approximation method to iteratively update the Bellman equation so as to estimate the action-value function:

$$Q_{t+1}(q,a) = \left(1 - \eta_t(q,a)\right) Q_t(q,a) + \eta_t(q,a) \left[hitrate_t(q,a) + \gamma \max_{a' \in Action} Q_{t'}\left(q', a'\right)\right) \quad (4)$$

where η is the learning rate. We set the value of η to 0.6 in the initial period of learning to make the algorithm tend to exploration. Then, we increased the exploitation probability of TQC by linearly reducing η. Finally, we fix the value of η to 0.2 to make the algorithm maintain 20% of the exploration capacity.

Because SaaS requests have a certain time periodicity, compared with LFU, Q-routing and Q-caching, TQC focuses on cache hit ratio. The outer loop of Algorithm 1 initializes the cache and table Q then processes each user request. For requests that have hit the cache, TQC updates the hit ratio, access counts, and the Q value of the contents. The inner loop of the algorithm (a) maps the user request to the actual data of the database or the file system through the map function; (b) adds the data directly to the cache when the cache is not full; (c) replaces the contents of the cache using $\varepsilon - policy$ when the cache is full; (d) executes action a_t, observes the number of hits, and calculates the Q value; and (e) selects the valid action from the action list after the timeout of the updating timer, actively loads the contents into the cache, and updates the whole Q table.

Algorithm 1 Time-based Q Cacher

Initialize the primary cache M_1 and the temporary cache M_2 with the size of S_1 and S_2 respectively.

Initialize action-value function Q with random value

for user request = 1, Q do

Initialize sequence $q_1 = \{data_1\}$ and preprocess sequenced $data_1 = map(q_1)$

If the cache is hit

Update cache hit ratio

Update content access counts

Update action Q value based on equation 4

Return data

Else

for t = 1, T do

Query the database or file system, $data_1 = map(q_1)$

If the temporary cache is not full

Cache the data directly

Else

with probability ε select a random action a_t

otherwise replace the content which has the minimum Q value $a_t = \min_{a \in A} Q_t^*(q,a)$

Execute action a_t and observe $hitrate_t(q,a)$ and q_{t+1}

Store transition (t, q, a, h) in table Q

If timeout

with probability select a random action a_t

otherwise replace the content which has the maximum Q value $a_t = \max_{a \in A} Q_t^*(q,a)$

Set $Q_t^* = \begin{cases} hitrate_j & \text{for terminal } q_{t+1} \\ hitrate_j + \gamma \max_{a' \in A} Q(q_{t+1}, a') & \text{for non-terminal } q_{t+1} \end{cases}$

Execute the operation of updating Q according to equation 4

else

continue

end for

end for

Compared with existing algorithms, TQC has the following advantages: first, the end-to-end way of learning is a model-free algorithm to improve the cache hit ratio. Without passing any artificial experience to the

algorithm, TQC can show a very good performance. Its hit ratio continues to increase until convergence. Second, the partitioned cache accelerates the convergence of the algorithm while ensuring the exploration of unknown contents and learning ability. Third, the learning style based on the time series only needs very little computing time and storage space. It will be able to learn long-period and low-frequency user request behaviors. Fourth, the learning style of active exploration allows user requests to directly hit the cache, which greatly reduces the time that a user's first request needs for reading data from the database or file system.

4 Results

The following simulation experiment evaluated four different caching policies namely LFU, LRU, FIFO, and TQC from cache space size, request rate, data amount, and average access time and got the digital analysis results. We selected 100,000 files of different sizes and types as the objects to be cached as shown in Table 1. The cache space size ranges from 32 to 64 GB. The time detection range of hit ratio is from 0 to 24 h. And the stride size is 1 h. File request rate ranges from 250 to 500 times/min. The specific parameters are shown in Table 2.

Comparing Fig. 4 with Fig. 3b, it can be seen that with the increase of request times, TQC hit ratio increases the fastest followed by LFU, LRU, and FIFO. It shows that TQC can achieve good learning effect with a large number of request samples. Comparing Fig. 4 with 3a, it can be found that simultaneous increases of cache capacity and request rate can make the hit ratio of TQC reach the steady state faster. For example, at $t = 5$ h, LFU, TQC, and LRU achieve the steady state at 500 times/min and need nearly twice the time to achieve the steady state at 250 times/min.

In a real network, the number of contents is increasing continuously. Suppose at t = 0, the number of files M = 100,000.1000 new files are generated every 6 h and these new files reach the highest degree of accesses within 6 h (Figs. 5, 6, 7, and 8).

5 Discussion

The methods based on statistics and manual caching policy not only put high requirements on computational resources and storage space but also have poor scalability and flexibility, which usually only adapt to specific environments. This paper proposes the cache management policy TQC based on reinforcement learning (Q-learning). Compared with traditional cache management methods, TQC not only eliminates the need to manually customize the cache rules but also has strong adaptability. In addition, TQC caching policy is more intelligent. The algorithm can well sense and predict low-frequency and long-period user requests, taking the first step in active caching and direct hit of requests. In the future, TQC will be based on distributed architecture and combined with neural network technology for prediction of continuous state spaces (user requests) so that the TQC hit ratio and SaaS response speed can reach a higher level.

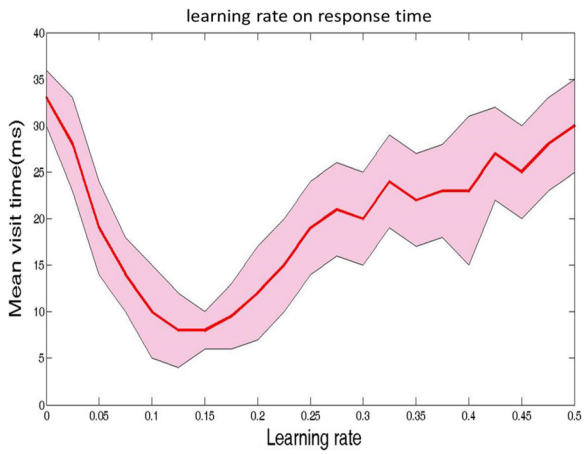

Fig. 8 Learning rate detection. We investigated the impact of learning rate on response time. It can be seen that the average response time of TQC reaches the minimum value of 8 ms when the learning rate is about 0.15

Abbreviations

FIFO: First in first out; IaaS: Infrastructure-as-a-service; LFU: Least frequently used; LRU: Least recently used; MEC: Minimum expected cost; PaaS: Platform-as-a-service; PER: Prioritized experience replay; RL: Reinforcement learning; SaaS: Software as a service; Sarsa: State action reward state action; SPT: Shortest path tree; TQC: Time-based Q Cacher; WCETs: Worst-case execution times; WLFU: Weighted least frequently used

Acknowledgements

Not applicable.

Funding

This research is supported by Natural Science Foundation of Hunan Province of China (No. 2016JJ4045) and Educational Commission of Hunan Province of China (No. 17A114). We thank the National Supercomputing Center in Changsha for providing with the technical support of this research.

Availability of data and materials

The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Authors' contributions

HC made substantial contributions to conception and design, or acquisition of data, or analysis and interpretation of data and been involved in drafting the manuscript or revising it critically for important intellectual content, and he has also given final approval of the version to be published. Each author should have participated sufficiently in the work to take public responsibility for appropriate portions of the content; GT agreed to be accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved. Both authors read and approved the final manuscript.

Authors' information

Haijun Chen received B.S. degrees at the School of Computer Science and Technology, National University of Defense Technology, Changsha, China, in 1997, and M.S. degree in software engineering from Hunan University,

Changsha, China, in 2006. He is currently pursuing Ph.D. degree at Central South University, Changsha, China. His research interests include wireless sensor networks, machine learning, and neural network.
GuanZheng Tan received B.S degree in aeronautical power plant control engineering from Nanjing University Of Aeronautics and Astronautics, Nanjing, China, in 1983, and M.S. degrees in automatic control theory and application from National University of Defense Technology, Changsha, China, in 1988, and Ph.D. degrees in mechanical manufacture and automation from Nanjing University Of Aeronautics and Astronautics, Nanjing, China, in 1992; he worked as a professor at the School of Information Science and Engineering of Central South University, Changsha, China. His research interests include artificial intelligence and application, bionic robot and intelligent bionic system, advanced control theory and advanced calculation, and biomedical image processing.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 28 July 2018 Accepted: 10 October 2018
Published online: 27 November 2018

References

1. L. Wu, S.K. Garg, R. Buyya, SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments[C]// Ieee/acm international symposium on cluster, cloud and grid IEEE Computer Society 2011:195–204
2. R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Futur. Gener. Comput. Syst. **25**(6), 599–616 (2009) Elsevier Science, Amsterdam, The Netherlands
3. M. A. Vouk, "Cloud Computing-Issues, Research and Implementation". In Proceedings of 30th International Conference on Information Technology Interfaces (ITI 2008), Dubrovnik, Croatia
4. H. Takahashi, K. Mori, H.F. Ahmad, Efficient I/O intensive multi tenant SaaS system using L4 level cache[C]// IEEE international symposium on service oriented system engineering. IEEE Computer Society, 2010:222–228
5. R. Wilhelm, J. Engblom, A. Ermedahl, et al., The worst-case execution-time problem—overview of methods and survey of tools. Cheminform **7**(3), 36 (2008)
6. X. Wang, M. Chen, T. Taleb, et al., Cache in the air: exploiting content caching and delivery techniques for 5G systems. IEEE Commun. Mag. **52**(2), 131–139 (2014)
7. D. Lee, J. Choi, J.H. Kim, et al., LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. Acm Sigmetrics Perform. Eval. Rev. **50**(12), 1352–1361 (2001)
8. Y. Wang, Z. Li, G. Tyson, et al., Design and evaluation of the optimal cache allocation for content-centric networking. IEEE Trans. Comput. **65**(1), 95–107 (2016)
9. I. Psaras, K.C. Wei, G. Pavlou, In-network cache management and resource allocation for information-centric networks. IEEE Trans. Parallel Distrib. Syst. **25**(11), 2920–2931 (2014)
10. D. Jeswani, M. Gupta, P. De, et al., Minimizing latency in serving requests through differential template caching in a cloud[C]// IEEE, International conference on cloud computing. IEEE, 2012:269–276
11. R. Neumann, S. Taggeselle, R. Dumke, et al., *Combining query performance with data integrity in the cloud: a hybrid cloud storage framework to enhance data access on the Windows Azure platform* (2012), pp. 518–525
12. V. Mnih, K. Kavukcuoglu, D. Silver, et al., Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
13. D. Silver, A. Huang, C.J. Maddison, et al., Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
14. N. Heess, J.J. Hunt, T.P. Lillicrap, et al., Memory-based control with recurrent neural networks. Comp. Sci. (2015)
15. C. Blundell, B. Uria, A. Pritzel, et al., Model-Free Episodic Control. 2016
16. T. Schaul, J. Quan, I. Antonoglou, et al., Prioritized experience replay. Comp. Sci. (2015)
17. L.J. Lin, Self-improving reactive agents based on reinforcement learning, planning and teaching. Mach. Learn. **8**(3), 293–321 (1992)
18. R. Chiocchetti, D. Perino, G. Carofiglio, et al., INFORM: a dynamic interest forwarding mechanism for information centric networking[C]// ACM SIGCOMM Workshop on Information-Centric NETWORKING. 2013:9–14
19. W. Caarls, E. Hargreaves, D.S. Menasché, Q-caching: an integrated reinforcement-learning approach for caching and routing in information-centric networks. Comp. Sci. (2015)
20. C.J.C.H. Watkins, P. Dayan, Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992)
21. M. Riedmiller, Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method[C]// European conference on machine learning. Springer-Verlag, 2005:317–328.