Learning-based Adaptive Data Placement for Low Latency in Data Center Networks

Kaiyang Liu[†*], Jingrong Wang[†], Zhuofan Liao[†‡], Boyang Yu[†], Jianping Pan[†]
[†]Department of Computer Science, University of Victoria, Victoria, Canada
[*]School of Information Science and Engineering, Central South University, Changsha, China
[‡]School of Computer and Communication Engineering, Changsha University of Science & Technology, Changsha, China
Email: {liukaiyang, jingrongwang, zfliao, boyangyu, pan}@uvic.ca

***Abstract*—Low-latency data access is an important challenge for data center networks. Proper placement of the data items can reduce the data travel time in the distributed storage systems, which contributes significantly to the latency reduction. Most existing data placement approaches have often assumed the prior distribution of data requests or discovered so through trace analysis. However, the traditional static model-based solutions are less effective to handle the system uncertainties in a dynamic environment. We present DataBot, a reinforcement learning-based adaptive framework, to learn the optimal data placement policies faced with the dynamic network conditions and time-varying request patterns. DataBot utilizes a neural network, trained with a variant of Q-learning, whose input is the real-time data flow measurements and whose output is a value function estimating the near-future latency. For rapid decision making, DataBot is divided into two decoupled production and training components, ensuring that the convergence time of the training will not introduce more overheads to serve the read/write requests. Evaluation results demonstrate that the average write and read latency of the whole system can be lowered by about 35% and 40%, respectively.**

I. Introduction

Data-intensive applications driven by web search, social networks, e-commerce and other data sources have recently generated explosive growth of workloads for the cloud data centers [1]. A defining characteristic of data analytical workloads is the low latency [2]. Major cloud providers, e.g., Amazon, Microsoft and Google, have observed that a slight increase in the overall data access latency may lead to observable fewer user accesses and thus a significant potential revenue loss [2].

In the process of data analytics, data items need to be moved frequently between computing or storage nodes, because data items are not always stored at the nodes where the computation happens. It has been witnessed that the data storage locations can affect the finish time of the distributed computation tasks, since the data movement delay is the main bottleneck when data items are intensively moved to fulfill a task [3]. Various data placement frameworks have been proposed to find the optimal data storage locations for latency reduction. Most existing research efforts focus on the hand-crafted design of optimization models by analyzing the factors that may affect the network latency [4]–[8]. However, different factors contribute to the latency, which could be time-variant. The sources include network latency, disk latency and other types of latency (e.g., RAM, CPU, etc.) [9]. Some of them are even of different proportions in different application scenarios. Hence, the traditional solutions based on static models are not flexible enough to deal with a dynamic environment with many uncertainties, such as unreliable network links, variable user request patterns and evolving system configurations.

Different from existing methods, we propose a generic framework, named DataBot, which learns to optimize the data placement policy with no future information about the system environments. Note that data placement problem can be treated as a finite Markov Decision Process (FMDP) as (1) the number of storage nodes at which the data can be placed is finite; (2) each action of the data placement is independent without storage constraints and the performance of such placement only depends on the current states and decisions. Therefore, the model-free Q-learning, which can find an optimal action-selection policy for any given FMDP [10], is utilized in this paper. DataBot acts as an agent interacting with the system, treated as a complex environment. This agent makes actions of choosing the storage location for each data item when writing or updating data items, and collects the feedback from the environment, including the current state of request patterns and network conditions, and the resultant end-to-end performance metrics (e.g., the read/write latency) due to these actions. Through trials and feedbacks, the DataBot can learn the optimal locations for storing data items or their replicas.

Although Q-learning is a promising approach, it may suffer from the curse of dimensionality and the consequently slow convergence with the increasing number of states/actions. Therefore, we propose to integrate a neural network (NN) into the Q-learning framework, achieving a quick approximation to the optimal solution with high accuracy. Given the current state as an input, NN can learn to calculate an output (i.e., the actions of choosing the storage locations). The resultant read/write latency is then used as a reward signal to train the NN model so that it gives better policies over time.

Furthermore, as the main objective is to make instant data placement decisions, we must ensure that the recurrent training process of the NN will not introduce extra overheads to the data read/write requests. The learning system is then decoupled into two asynchronous components, i.e., the production and training system, in the implementation of our design. The online decision making and offline training methods change

978-1-5386-4413-3/18/$31.00 ©2018 IEEE

the traditional workflow of reinforcement learning (RL), which requires updating the model after each decision. As a result, our proposed framework makes instant decisions only with the newly trained NN for the requests of querying write locations, which manifests the ability to reduce the data placement delay without introducing extra overheads. The main contributions of this paper are summarized as follows:

1) We present a generic and adaptive data placement framework to learn the optimal data placement policy from the environment, without assuming the prior distribution of data requests or the future information about the system dynamics.
2) NN and RL are utilized to reduce the data read/write latency for data center networks. With the increasing number of states/actions, NN achieves a quick approximation when combined with RL. Moreover, the online decision making and offline training overcome the deficiency of the framework in delaying the request handling.
3) Furthermore, large-scale evaluations driven by real-world I/O traces demonstrate that DataBot can lower the average write and read latency of the whole system by about 35% and 40%, respectively.

The remainder of this paper is outlined as follows. Section II surveys the related work. Section III provides the system model and problem statement. Section IV presents the design detail of the learning-based data placement framework named DataBot. Section V evaluates the performance of DataBot. Section VI draws the conclusion and lists the future work.

II. Related Work

Many researchers have pointed out that data placement improves the data locality to ensure a better data read/write performance in data-intensive systems. Given the assumption that queries from clients are known beforehand, Ren et al. [5] formulated the data placement as an integer linear programming problem, and proposed a near-optimal solution to jointly optimize the service cost and latency. Assuming the prior distribution of data requests, Yu et al. [6] designed a general hypergraph-based framework for data placement among geo-distributed storage nodes. Using characteristics of future workloads, Jalaparti et al. [11] presented an offline scheduling framework that jointly places data and tasks to significantly improve the network locality. By analyzing cloud-service traces, Agarwal et al. [12] presented an automated data placement framework for geo-distributed services where the geographical distribution of requests would determine the final data locations. However, all these previous studies investigated the data placement problem in an offline way.

Considering the online data placement problem, Steiner et al. [13] placed the data items used in the same job to the storage nodes of the same rack to ensure the inter-rack traffic can be largely reduced and therefore the job finish time is shortened. Chowdhury et al. [14] proposed that the data write flow could always choose the node with low occupancy links in the writing path as the destination to lower the job finish time. However, the ignorance of the read operations in this work may adversely affect the future read-related performance. Different from existing methods, the proposed framework DataBot takes both the read and write latency into account, and uses RL to adaptively learn and adjust the data placement strategy without future data request information.

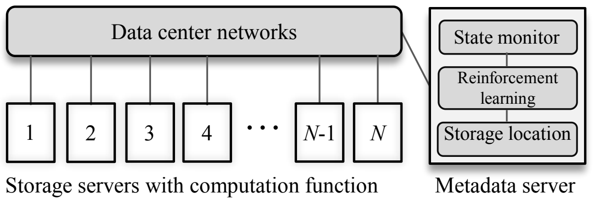


Fig. 1. Data storage system: storage servers, data center networks and metadata server.

Our study is related to the idea of combining NN and RL for joint optimization [15], [16], but we focus on the data placement in data centers. Mao et al. [17] investigated the resource management problems with policy gradients. Nevertheless, they optimized the expected value of a manually designed objective function on the basis of the reward. Unlike this work, Mirhoseini et al. [18] directly utilized the application execution time as the reward of RL to optimize the device placement with no need of designing intermediate cost models. Motivated by previous studies, we directly use the end-to-end performance metric, i.e., the observed read/write latencies as the reward, to reduce the data access latency in distributed storage systems. Furthermore, our asynchronous implementation ensures the training process will not introduce extra overheads to the request handling.

III. System Model and Problem Statement

In this section, we present the system model of the data storage system, and discuss how to dynamically optimize the storage locations of data items with intensive data flows.

A. System Model

Fig. 1 illustrates the model of the data storage system. We consider a distributed storage system consisting of a set of machines or servers $\mathcal{N}$ (with size $N = |\mathcal{N}|$). Each server in the system has both the storage and computation functions. For the storage function, data items are distributed among various servers. For the computation function, applications running on multiple servers may require the data movement among servers. All servers are connected through a data center network (DCN). As our objective is to design a generic data placement solution, we do not focus on any specific topology of the DCN. Owing to the fact that our design is only based on the measurement of the end-to-end network performance, it can support any arbitrary DCN topologies, e.g., the tree-based Clos and Fat-tree, the recursive DCell and BCube, or the flexible Helios and cThrough architectures [19].

A centralized metadata server is deployed to manage the storage locations of data items. Let $\mathcal{M}$ denote the set of data items stored in the system (with size $M = |\mathcal{M}|$). The data items could be files, tables or blocks in practice. Each data item is assigned with a unique hashtag, i.e., the hash output using the index of the data item as the input. When a data item is written into the system, the metadata server maintains the mapping between the hashtag and its storage server. When an application on an ordinary server needs to retrieve a data item, it first asks the metadata server where the storage server is by using the hashtag. Under this framework, the storage location of a data item is flexible and can be changed, whenever the data item is to be written or updated. By this design, no data movement overhead is introduced even though the proposed framework occasionally changes data storage locations.

The metadata server captures the logs of the read/write requests from each storage server through the state monitor module. The format of log entries is defined as

$$(\textbf{TS}, \textbf{R/W}, \textbf{Src}, \textbf{Dst}, \textbf{Lat}), \tag{1}$$

where **TS** is the timestamp, **R/W** represents the operation type (read or write), **Src** and **Dst** are the source and destination location of the requests, and **Lat** is the end-to-end latency of the operations. It is worth noting that the metadata server has all the information by itself except **Lat**. Therefore, it is only necessary to report **Lat** from the storage servers in the system.

B. Problem Statement

Currently, data storage systems need to serve a variety of applications, e.g., the mostly read-only analytical workloads and the high-throughput transactional workloads that both need low latencies [20]. As mentioned before, the data storage locations can affect the finish time of distributed workloads. Therefore, for the purpose of low-latency services, the optimization problem is defined as follows: when a data item and its replicas are to be written or updated, how to choose the optimal storage locations among all available servers?

Within the data center, the end-to-end latency is the sum of a number of components, including transmission latency, propagation latency, processing latency and queuing latency. Therefore, it is difficult to model the system accurately faced with the dynamic environment, i.e., the rapidly changing network conditions and request patterns. Therefore, we utilize a generic method RL to address the formulated problem. RL is inspired by the behaviorist psychology, addressing hard optimization problems through a learning process. In RL, an agent interacts with the environment and then learns the underlying model utilizing the feedback of its actions. Different from the traditional methods aiming at obtaining the analytical models of the physical environments, the adopted RL chooses an alternative, which is to purely fit with the statistical patterns of the environment.

IV. RL-Based Data Placement

We adopt Q-learning, a typical RL algorithm, to address the data placement problem, which is model-free and can be demonstrated to find an optimal policy for any given FMDP. The design overview is presented at first, followed by the design details.

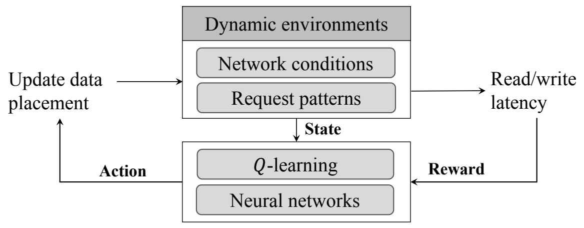


Fig. 2. An overview of the RL-based data placement.

A. Design Overview

Fig. 2 illustrates the design overview of the Q-learning-based data placement. The fundamental principle of Q-learning is captured by the maintained reward function or called Q-function:

$$Q : State(\mathcal{S}) \times Action(\mathcal{A}) \rightarrow Reward(\mathcal{R}).$$

The dynamic information of environments (State $\mathcal{S}$) can be learned from a series of data flows to understand which data item should be placed on which server (Action $\mathcal{A}$) so that the corresponding read/write latencies are reduced. These read/write latencies are then used as Reward $\mathcal{R}$ to train the recurrent model and thus the DataBot outputs better data placement decisions over time in the long-term.

When a specific data item m is going to be written in the system at time t, we first choose the location based on the current state $\boldsymbol{s}$ and then execute the write operation to that location as an action a, $m \in \mathcal{M}$, $\boldsymbol{s} \in \mathcal{S}$, $a \in \mathcal{A}$. Until we start to overwrite or update the same data m at t', we can gather the effect or performance metrics of the last written m at t and the following reads of m between t and t'. The weighted sum of the read and write latencies is used as the immediate reward value r for the action a. Since the write operation occurs at t', the system jumps to another state $\boldsymbol{s}'$. For the immediate reward r at t in Q-learning, it is assumed to still have an impact on the future, and discounted under a discount factor γ for the future moments. According to [21], the optimal solution $Q^*(\boldsymbol{s}, a)$ that maximizes the expected long-term reward satisfies the condition as follows

$$Q^*(\boldsymbol{s}, a) = \mathbb{E}_{\boldsymbol{s}'}[r + \gamma \max_{a'} Q^*(\boldsymbol{s}', a')|\boldsymbol{s}, a], \tag{2}$$

and through value iterations,

$$Q_{t+1}(\boldsymbol{s}, a) = \mathbb{E}_{\boldsymbol{s}'}[r + \gamma \max_{a'} Q_t(\boldsymbol{s}', a')|\boldsymbol{s}, a]. \tag{3}$$

As future rewards are influenced by many factors, such as dynamic network conditions and request patterns, the traditional RL based on temporal differences [22] fails to guarantee a fast convergence to the optimal solution. To address this critical issue, the NN is employed to approximate the Q-function with a high accuracy in our system.

B. Q-Function Design

1) States: The feature of our design is that the decisions are only made on the basis of end-to-end measurements from data flows. The state s consists of three categories of the state information, which can be derived from the stored logs of the read/write requests. They help make a decision on the storage location when serving a write request of data item m from server i, $m \in \mathcal{M}$, $i \in \mathcal{N}$.

Network Conditions: The first category is about the network conditions

$$\left\{L_{ij}^{[\mathsf{R}]}, L_{ij}^{[\mathsf{W}]}, \forall i, j \in \mathcal{N}\right\}, \tag{4}$$

which includes: 1) average latency of read operations on each pair of servers in the network, $L_{ij}^{[\mathsf{R}]}$, where $i, j \in \mathcal{N}$ are the source and destination, respectively; 2) average latency of write operations on each pair of servers in the network, $L_{ij}^{[\mathsf{W}]}$. We measure the information mentioned above, considering that the network performance has a large impact on the objective of improving read/write latency. In our design, the link-based metrics are ignored since the discussed data placement happens on the application layer, where the flow destination, not the path or links, is chosen. Thus our decisions on storage locations can coexist with any underlying link-based or path-based flow scheduling. Note that the measurement of the link-related metrics would introduce a large overhead when compared with the end-to-end ones.

Then, we show how to reconstruct the real-time network condition information from the logs in (1). Let l denote the time for each data movement. Based on the **Lat** l, the Exponential Weighted Moving Average (EWMA) mechanism [23] is utilized to estimate the average read/write latency $L_{ij}^{[\mathsf{R/W}]}$. Specifically, after finishing a data read/write operation, $L_{ij}^{[\mathsf{R/W}]}$ is updated by

$$L_{ij}^{[\mathsf{R/W}]} = \alpha_l l + (1-\alpha_l) L_{ij}^{[\mathsf{R/W}]}, \tag{5}$$

where α_l is the discount factor to lower the importance of the previous requests. In this way, we only need $O(1)$ space to maintain the estimated latency for each pair of servers in the network.

Request Patterns: The second category is about the request patterns

$$\left\{F_{i,m}^{[\mathsf{R}]}, F_{i,m}^{[\mathsf{W}]}, \widetilde{F}_i^{[\mathsf{R}]}, \widetilde{F}_i^{[\mathsf{W}]}, \forall i \in \mathcal{N}\right\}, \tag{6}$$

which includes: 1) read rate or frequency to data m from source server i, denoted by $F_{i,m}^{[\mathsf{R}]}$; 2) write rate to data m from source server i, denoted by $F_{i,m}^{[\mathsf{W}]}$; 3) read rate to all data from source server i, denoted by $\widetilde{F}_i^{[\mathsf{R}]}$; 4) write rate to all data from source server i, denoted by $\widetilde{F}_i^{[\mathsf{W}]}$. With both the request patterns towards all data and the specific requested data, the framework would be able to make a better choice on storage locations, because such information indicates how applications are generating workloads to the system, which is the root cause of the network traffic.

In order to derive the request pattern information from the logs in (1), the Discounting Rate Estimator method [24] is applied here. We maintain a counter for each item in (6), which increases with every read/write operation on each pair of servers in the network, and decreases periodically (every T_r) with a ratio factor $\alpha_r \in (0,1)$. The benefits of this method are that: (i) it reacts quickly to the changes of the request patterns; (ii) it meets the $O(1)$ requirement of both the space and the update time for each maintained counter.

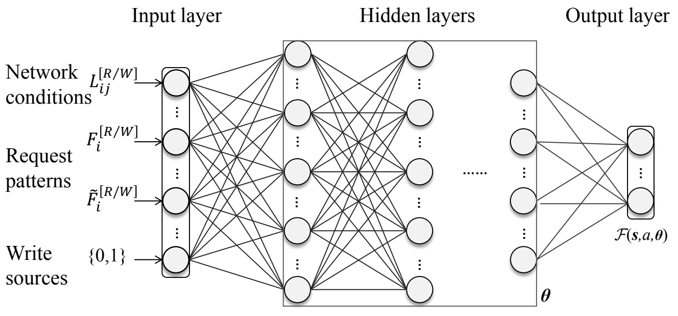


Fig. 3. NN structure for data placement.

Source location: The third category is a 0-1 vector, representing whether each server $i \in \mathcal{N}$ is the source location of the current write operation or not. We assume that the number of replicas for each data item is the same[1], which is denoted as k. Therefore, there are k servers being 1 in the 0-1 vector. This category is necessary because the source locations of the data flow would make a difference to the latency of the write operation on a specific server.

In our design, given the server number N, the size of state s will be

$$|s| = 2N^2 + N + 4N = O(N^2). \tag{7}$$

According to (7), the number of data items in the system will not affect the deployment complexity of the learning-based system.

2) Actions: The action set is also a 0-1 vector, which determines the destination locations of the requested data for the write operation. Similarly, we have k servers being 1 in the 0-1 vector for each data item.

3) Rewards: Our objective is to achieve a low-latency data placement. As the read/write latency can be affected by time-variant factors, the measured read/write latencies are directly used to calculate the immediate reward r_t. It is defined as the weighted sum of the transmission rates measured during time $[t, t')$, where t is the timestamp of writing a data item m and t' is the timestamp of writing the same data item for the next time. Thus, r_t is calculated by

$$r_t = \omega \cdot \frac{1}{l^{[\mathsf{W}]}} + (1-\omega)\frac{1}{|\mathcal{P}|}\sum_{p\in\mathcal{P}} \frac{1}{l_p^{[\mathsf{R}]}}, \tag{8}$$

where $l^{[\mathsf{W}]}$ is the write latency measured at time t, $\mathcal{P}$ represents the set of read operations in $[t, t')$, and $l_p^{[\mathsf{R}]}$ is the latency for the read operation $p \in \mathcal{P}$. Considering the relative importance between read and write may be different in various scenarios, the parameter $\omega \in (0,1)$ is introduced to make the tradeoff.

[1]In this paper, we will not differentiate the data item itself and its replicas, so both of them are treated as replicas below.

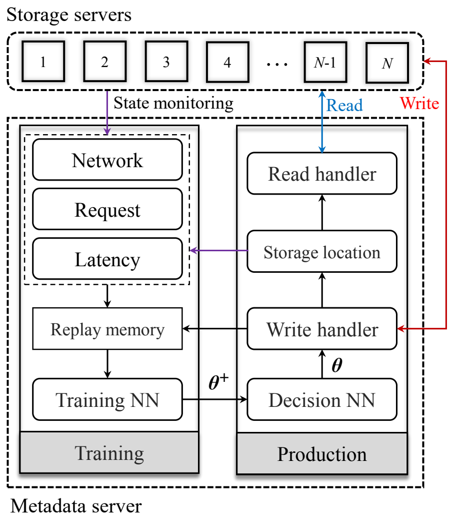


Fig. 4. Production and training system of DataBot.

C. *Asynchronous NN Implementation*

Then, NN techniques are introduced to approximate the action-value function of the Q-learning framework. As shown in Fig. 3, an example NN has three kinds of layers, i.e., input layer, hidden layers and output layer. Each layer contains a number of computing items called neurons. Each neuron receives values from all neurons in its previous layer, and conducts the calculation with the weights of connections $\boldsymbol{\theta}$ between neurons in the adjacent two layers. The weight vector $\boldsymbol{\theta}$ of the NN should be updated periodically to improve the accuracy of the approximation.

It is worth noting that traditional RL is designed as updating the model after each decision [15], [22], which is in serial and therefore will delay the next request handling. Unlike previous work, the objective of DataBot is that the proposed system could make instant decisions for the requests of querying write locations, so that the learning process will not introduce extra delays to the read/write latencies. As shown in Fig. 4, we design the DataBot framework with two components, i.e., the production and training system. They work asynchronously to ensure the training process will not affect the running of the production system.

1) Production System: Here we clarify the design details of the production system. As illustrated in Fig. 4, the main purpose of the production system is to serve the requests of querying write locations through the deployed NN. Given a state $\boldsymbol{s}_t$, an action a_t, and the current weight vector $\boldsymbol{\theta}$ of the maintained NN, we can calculate the output of the NN, denoted by $\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$, which is a vector with the length of the number of storage servers N in the system. Each value in $\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$ represents the expected reward of the corresponding action, or equivalently, the expected reward of writing the data item to the corresponding server.

The pseudo code of the production algorithm is shown in Algorithm 1. Whenever there is a request to the metadata server which queries the write destination for a specific data item at time t, we generate a random number $\eta \in [0, 1]$ that obeys the uniform distribution. If $\eta < \epsilon$, we select the action a_t^* that maximizes the output of function $\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$ to obtain a lower read/write latency. Otherwise, a random action will be selected to search the unexplored portion of the solution space. After applying the action a_t^*, we can observe the reward r_t and system state $\boldsymbol{s}_{t+1}$ at the end of time interval t. A tuple

$$\tau = (\boldsymbol{s}_t, a_t, \boldsymbol{s}_{t+1}, r_t) \tag{9}$$

is stored for each request. The tuples for a certain period constitute the replay memory $\mathcal{R}$, which can be used by the training system [2].

Algorithm 1 Production Algorithm

```
Input: NN weight vector $\boldsymbol{\theta}$, state $\boldsymbol{s}_t$, $\epsilon$.
Output: Data placement action $a_t^*$, reward $r_t$, state $\boldsymbol{s}_{t+1}$.
1: while A request queries the write destination at $t$ do
2:     Generate a random number $\eta \in [0, 1]$;
3:     if $\eta < \epsilon$ then
4:         $a_t^* \leftarrow \arg\max_{a_t \in \mathcal{A}} \mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$;
5:     else
6:         Randomly select action $a_t^* \in \mathcal{A}$;
7:     end if
8:     if Replay memory $\mathcal{R}$ is full, $|\mathcal{R}| = R$ then
9:         Discard the earliest tuple in $\mathcal{R}$;
10:    end if
11:    Store the tuple $(\boldsymbol{s}_t, a_t, \boldsymbol{s}_{t+1}, r_t)$ in replay memory $\mathcal{R}$;
12: end while
```

Algorithm 2 Training Algorithm

```
Input: NN weight vector $\boldsymbol{\theta}$, replay memory $\mathcal{R}$.
Output: Updated weight vector $\boldsymbol{\theta}^+$.
1: if Replay memory $\mathcal{R}$ is full, $|\mathcal{R}| = R$ then
2:     Shuffle all tuples $\tau \in \mathcal{R}$ to generate mini-batches $\mathcal{B}$;
3:     for epoch $i = \{1, 2, ..., I\}$ do
4:         for each tuple $\tau \in \mathcal{R}$ do
5:             $y_t \leftarrow r_t + \gamma \max_{a_{t+1}} \mathcal{F}(\boldsymbol{s}_{t+1}, a_{t+1}, \boldsymbol{\theta})$;
6:         end for
7:         for each mini-batch $b \in \mathcal{B}$ do
8:             Update $\boldsymbol{\theta}^+$ to minimize (11);
9:         end for
10:        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^+$;
11:    end for
12: end if
```

2) Training System: The training system periodically obtains the tuples in the relay memory $\mathcal{R}$ of the production system, and replays them to train an updated weight vector $\boldsymbol{\theta}^+$ for the future data placement.

[2]We define R as the maximum size of the replay memory. If the pool of the replay memory $\mathcal{R}$ is full, i.e., the number of tuples $|\mathcal{R}| = R$, the earliest tuple in $\mathcal{R}$ will be discarded. This ensures only the latest tuples will be stored to reflect the current network conditions and request patterns.

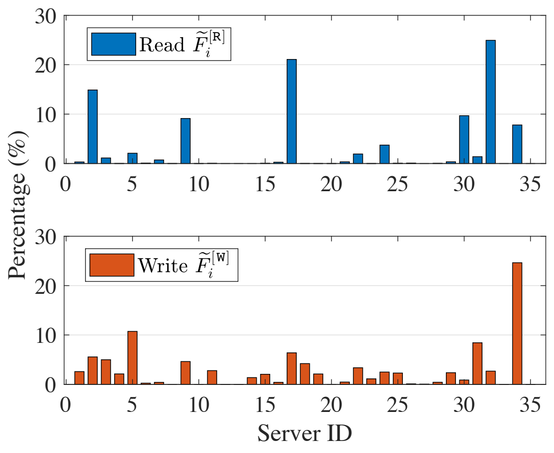


Fig. 5. Read/write request arrival rate of data items based on MSR Cambridge Traces [25]. The distribution is biased among storage servers.

The pseudo code of the training algorithm is shown in Algorithm 2. Fundamentally, the mini-batch stochastic gradient descent (SGD) method [20] is utilized, which updates the weight vector $\boldsymbol{\theta}$ in order to minimize the difference between the output of the NN and the target value. Based on the observed reward r_t as in (8), the target value is defined by

$$y_t = r_t + \gamma \max_{a_{t+1}} \mathcal{F}(\boldsymbol{s}_{t+1}, a_{t+1}, \boldsymbol{\theta}), \tag{10}$$

Then, all tuples in $\mathcal{R}$ are partitioned into subsets, or termed as mini-batches $\mathcal{B}$. For each mini-batch $b \in \mathcal{B}$, we can update the weight vector $\boldsymbol{\theta}^+$ with the gradient method, in order to minimize

$$\mathbb{E}_{(\boldsymbol{s},a)\sim\mathcal{B}} \left[\left(y - \mathcal{F}\left(\boldsymbol{s}, a, \boldsymbol{\theta}^+\right)\right)^2 \right]. \tag{11}$$

In the training method, we have multiple iterations, or termed as epochs, on all mini-batches to converge faster. The number of epochs is denoted by I. We keep the weight vector $\boldsymbol{\theta}$ of the decision NN stable before all records in $\mathcal{R}$ have been processed. Then after a complete round of processing, the weight vector $\boldsymbol{\theta}^+$ of the training NN is transferred to $\boldsymbol{\theta}$. This variant in training makes the optimization objective more stable and therefore avoids fluctuations to some extent.

V. Performance Evaluation

In this section, we perform extensive evaluations driven by large volumes of real-world I/O traces, i.e., MSR Cambridge Traces [25], to evaluate the performance of DataBot.

A. Trace Description and Experiment Settings

MSR Cambridge Traces: These are the I/O traces of an enterprise data center at Microsoft Research Cambridge, where data read/write requests are captured from 36 storage volumes for one week. Fig. 5 illustrates the arrival rates of the read/write requests $\widetilde{F}_i^{[R/W]}$ of all data items. The distribution is biased among storage servers due to real applications. For each request, the hostname, request type (read/write), and timestamp are given. However, for the reason of confidentiality, most publicly available traces, including the utilized MSR Cambridge Traces, do not specify the detailed data item for each read/write request. We assume that the total number of data items is $M = 10,000$ in the system. The request rates of data items in server i $F_{i,m}^{[R/W]}$ follow a Zipf distribution with $\widetilde{F}_i^{[R/W]} = \sum_{m=1}^{M} F_{i,m}^{[R/W]}$ as in Fig. 5, just similar to [5], [6].

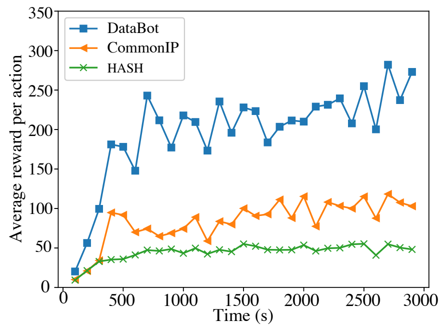
Experiment Settings: Mininet is used to emulate the data center network, which can create a virtual network running real Linux-based applications [26]. All servers are in a representative Fat-Tree topology with 3 levels of switches. The capacity of links is set to 1 Gbps. The client program is implemented at each server, which initiates the data read and write requests according to the traces. In order to improve the efficiency of intensive data access, Memcached [27] is used as the end of data flows and caches data items in RAM. Simply speaking, each storage server has a client being the source of requests and a Memcached process being the destination of requests. The default data block size is set to 64 MB, just similar to the widely used Hadoop Distributed File System (HDFS) [28]. A metadata server program is also implemented to handle control flows, whose functions include state monitoring, write destination decision and NN training. Multilayer perceptron (MLP) [30] with one kernel is adopted as the structure of the NN used in our system.

Performance Baselines: In the experiments, two other data placement frameworks are introduced for a fair performance comparison. The first is `commonIP` [12]—places data as close as possible to the IP address that most commonly accesses it. The second is `HASH`—hashes data to servers so as to optimize for load-balancing, which has been adopted in most distributed storage systems today, such as HDFS [28] and Cassandra [29].

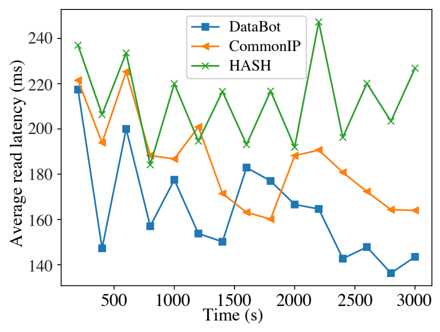
B. Experiment Results

We start by showing the experiment results of weight $\omega = 0.2$ without replica $k = 1$ scenario in the 3,000 s running time of the system. Note that the time of 3,000 s is long enough to reach a steady performance improvement ratio. For each round of training, the system obtains the last $R = 2,000$ samples from the replay memory and processes them with $I = 6$ epochs and $|b| = 300$ batch size, which averagely takes 60 s until the new weight vector $\boldsymbol{\theta}^+$ takes effect in the production system. Fig. 6(a) shows the average reward per action of data placement as defined in (8). At the beginning of the data service, the data read/write latencies reduce as Memcached needs to be warmed up with data items. Therefore, the average rewards with `commonIP` and `HASH` increase for the first hundreds of seconds, and keep fairly stable then. When compared with the hand-crafted heuristics above, DataBot continuously learns better data placement policies through trials and feedbacks over time. As shown in Fig. 6(a), the average reward is in an increasing trend with the learning process of DataBot. After multiple rounds of training for convergence, the improvement ratio through DataBot will become stable after 1,000 s.

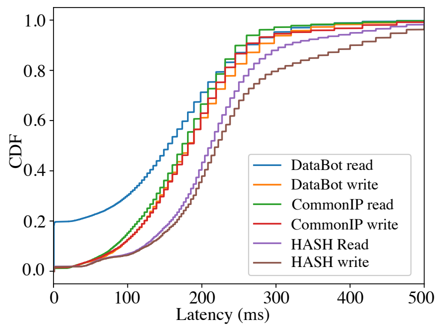
As shown in Fig. 6(b), we also measure the average read latencies in every 200 s of the experiment duration. `HASH` can be intuitively understood as random placement without



(a) Average reward per action r_t



(b) Read latency



(c) CDF of latency

Fig. 6. Read optimized $\omega = 0.2$.

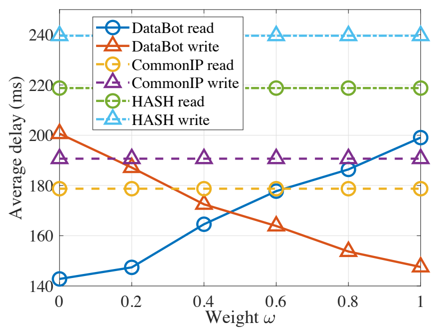


Fig. 7. Impact of weight ω.

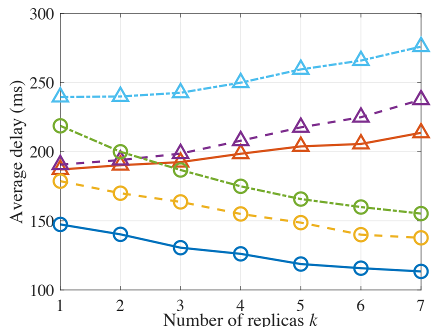


Fig. 8. Impact of replica number k.

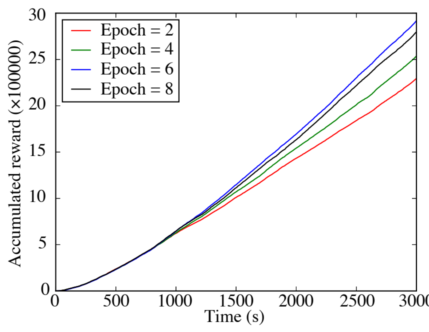


Fig. 9. Impact of epoch number I.

considering the network conditions or request patterns, and thus achieves the highest average read/write latencies 218.8 ms and 239.6 ms for the last 1,000 s, respectively. `commonIP` places the data item on the server that has the largest request rate. However, with the biased distribution of data items among servers as shown in Fig. 5, the queuing latency will inevitably increase with the average read/write latencies 174.5 ms and 185.6 ms, respectively. For the last 1,000 s with the optimized placement policy, DataBot achieves the lowest read latencies 147.4 ms with the write latency 187.1 ms. Moreover, as shown in Fig. 6(a) and (b), the performance shows some variance because of the fluctuations of the request patterns.

We also show how the read/write latencies of data requests are distributed for the last 1,000 s in Fig. 6(c). The latency distribution could reveal how the performance is changed for a certain percentage of requests after introducing DataBot. It can be observed that more than 40% of the write requests are finished in less than 128.1 ms by the proposed scheme while being 163.6 ms and 198.6 ms for `commonIP` and `HASH`. This confirms that DataBot is effective in ensuring more requests with lower read latencies in this scenario.

C. Parameter Impacts

In order to fully evaluate the performance of the proposed DataBot framework, several factors, including the weight ω and the number of replicas k, which may affect the data placement process, are also considered.

Weight ω: As ω is the tradeoff parameter between the importance of write and read, larger ω indicates a higher priority of write requests but with a less concern of read requests. From Fig. 7, we can see that when ω increases from 0 to 1.0, the average write latency is decreased by 28.26% (from 199.06 ms to 147.56 ms), while the read latency is increased by 39.40% (from 142.80 ms to 199.06 ms). For the read-optimized scenario ($\omega = 0$), compared with `commonIP` and `HASH`, the read latency can be reduced by 20.1% and 34.73%, respectively. Furthermore, for the write-optimized scenario ($\omega = 1.0$), the write latency can be reduced by 22.61% and 38.42%, respectively. As shown in Fig. 7, balanced read/write latencies can be achieved when ω is between 0.4 and 0.5.

Number of Replicas k: Data replicas can improve the reliability, fault-tolerance and accessibility of the system. When the number of replicas increases from 1 to 7 with $\omega = 0.2$, computing servers have more choices to access needed data items. Therefore, the network congestion of read requests can be eased. The read latency can be reduced from 147.41 ms to 113.40 ms. Nevertheless, as the data item needs to be written to k different storage servers, the overall network congestion will be inevitably increased with more data flows. The write latency is increased from 187.13 ms to 213.72 ms. Fig. 8 demonstrates that DataBot can effectively choose the nodes with lower read/write latencies under the replication setting.

Number of Training Epochs I: Note that the number of training epochs I affects the training process of the NN and the resultant data placement performance. As shown in Fig. 9, with the increase of I from 2 to 6, the accumulated reward increases as more rounds of training tend to reduce

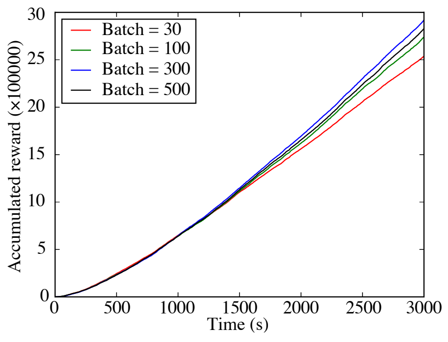


Fig. 10. Impact of batch size $|b|$.

the difference between the expected reward and the output given by the NN. However, when I is further increased to 8, a degradation of reward can be observed due to over-fitting. It means that even though the return of the loss function in the NN training can be further decreased with more training rounds, the obtained model may not be effective for the future samples, and thus we let $I = 6$ in the performance evaluation.

Batch Size $|b|$: Then, the batch size, which represents how frequently the weight vector θ^+ is updated in the training process, is also numerically tested. As shown in Fig. 10, when the batch size is 300, the highest reward can be achieved compared with the other settings. Fig. 9 and 10 suggest that a careful selection of the training parameters can help to improve the performance of the learning-based system. With these discoveries, we are interested in finding a more systematic way to properly set and fine tune the parameters and to avoid over-fitting in the follow-on work.

VI. Conclusion and Future Work

In this paper, we propose a Q-learning-based data placement framework, DataBot, to automatically learn the optimal data placement policies in order to handle the uncertainties of the dynamic system. The NN is utilized to estimate the near-future latency by training the weight vector with the Q-values, thus speeding up the convergence to the optimal solution. Moreover, two asynchronous components, i.e., the online decision making and offline training, are integrated seamlessly to ensure that no extra overheads are introduced to the data request handling. Evaluation results show that the average write and read latencies are reduced when compared with the existing, often-used solutions. For the scalability in the future work, the distributed RL solutions can be further explored to speed up the convergence of the learning process in the data placement problem, with no need of aggregating raw data to a centralized metadata server for training.

References

[1] Y. Mansouri, A.N. Toosi, and R. Buyya, "Data storage management in cloud environments: Taxonomy, survey, and future directions," *ACM Comput. Surv.*, vol. 50, no.6, pp. 1–51, 2018.

[2] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *in Proc. of ACM SIGCOMM*, pp. 421–434, 2015.

[3] M.Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: Flexible data placement and its exploitation in Hadoop," *in Proc. VLDB Endow.*, vol. 4, no. 9, pp. 575–585, 2011.

[4] Y. Xiang, T. Lan, V. Aggarwal, Y.F.R. Chen, "Joint latency and cost optimization for erasure-coded data center storage," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 1063-6692, 2016.

[5] X. Ren, P. London, J. Ziani, and A. Wierman, "Datum: Managing data purchasing and data placement in a geo-distributed data market," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 893–905, 2018.

[6] B. Yu, and J. Pan, "A framework of hypergraph-based data placement among geo-distributed datacenters," *IEEE Trans. Serv. Comput.*, 2017.

[7] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang, "Latency reduction and load balancing in coded storage systems," *in Proc. of ACM SoCC*, pp. 365–377, 2017.

[8] Y. Fan, H. Ding, L. Wang, and X. Yuan, "Green latency-aware data placement in data centers," *Comput. Netw.*, vol. 110, pp. 46–57, 2016.

[9] "Latency Definition." [Online]: https://techterms.com/definition/latency

[10] L.P. Kaelbling, M.L. Littman and A.W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[11] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *in Proc. of ACM SIGCOMM*, pp. 407-420, 2015.

[12] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," *in Proc. of USENIX NSDI*, 2010.

[13] M. Steiner, B.G. Gaglianello, V. Gurbani, V. Hilt, W.D. Roome, M. Scharf, and T. Voith, "Network-aware service placement in a distributed cloud environment" *in Proc. of ACM SIGCOMM*, pp. 73–74, 2012.

[14] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," *in Proc. of ACM SIGCOMM*, pp. 231–242, 2013.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *in NIPS Deep Learning Workshop*, 2013.

[16] I. Bello, H. Pham, Q.V. Le,, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[17] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," *in Proc. of ACM HotNets*, pp. 50–56, 2016.

[18] A. Mirhoseini, H. Pham, Q.V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio and J. Dean, "Device placement optimization with reinforcement learning," *in Proc. of the ICML*, 2017.

[19] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A survey on data center networking (DCN): Infrastructure and operations," *IEEE Commun. Surv. Tuts.*, vol. 19, no. 1, pp. 640–656, 2017.

[20] K.A. Kumar, A. Quamar, A. Deshpande, and S. Khuller, "SWORD: Workload-aware data placement and replica selection for cloud data management systems," *VLDB J.*, vol. 23, no. 6, pp. 845–870, 2014.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[22] C. Xu, K. Wang, P. Li, R. Xia, S. Guo, and M. Guo, "Renewable energy-aware big data analytics in geo-distributed data centers with reinforcement learning," *IEEE Trans. Netw. Sci. Eng.*, 2018.

[23] J.S. Hunter, "The exponentially weighted moving average," *J. Qual. Technol.*, vol. 18, no. 4, pp. 203–210, 1986.

[24] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," *in Proc. of ACM SIGCOMM*, pp. 503–514, 2014.

[25] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, 2008.

[26] "Mininet." [Online]: http://mininet.org/

[27] "Memcached" [Online]: http://memcached.org/

[28] "HDFS Architecture Guide." [Online]: https://hadoop.apache.org/

[29] "Cassandra." [Online]: http://cassandra.apache.org/

[30] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.