A Learning-based Data Placement Framework for Low Latency in Data Center Networks

Kaiyang Liu, *Student Member, IEEE,* Jun Peng, *Member, IEEE,* Jingrong Wang, *Student Member, IEEE,* Boyang Yu, *Student Member, IEEE,* Zhuofan Liao, Zhiwu Huang, *Member, IEEE,* and Jianping Pan, *Senior Member, IEEE*

Abstract—Low-latency data service is an increasingly critical challenge for data center applications. In the modern distributed storage systems, proper data placement helps reduce the data movement delay, which can contribute to the service latency reduction tremendously. Existing data placement solutions have often assumed the prior distribution of data requests or discovered it via trace analysis. However, data placement is a difficult online decision-making problem faced with dynamic network conditions and time-varying user request patterns. The conventional static model-based solutions are less effective to handle the dynamic system. With an overall consideration of data movement and analytical latency, we develop a reinforcement learning-based framework DataBot+, automatically learning the optimal placement policies. DataBot+ adopts neural networks, 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 instantaneous decision making, DataBot+ is decoupled into two asynchronous production and training components, ensuring that the training delay will not introduce extra overheads to handle the data flows. Evaluation results driven by real-world traces demonstrate the effectiveness of our design.

Index Terms—Data center network, data placement, reinforcement learning, neural networks.

1 Introduction

CURRENTLY, we have witnessed the explosive growth of workloads driven by data-intensive applications, e.g., web search, social networks, and e-commerce [2]. The key challenge is to perform low-latency services with real-time workloads. Cloud service providers, e.g., Amazon and Google, have reported that a slight increase in the overall service latency may cause observable fewer user accesses and thus a considerable revenue loss [3].

The user-experienced service latency mainly consists of the data movement (for both read and write) and analytical latencies. In order to perform data analytics, data items should be moved intensively among computing or storage nodes, as they are not always stored at the locations where the execution happens. It has been reported that the storage locations of data items can influence the completion duration of distributed analytics, because the movement latency could be the major bottleneck when data are frequently moved among storage nodes [4]. Various data placement solutions have been proposed to find the optimal data storage locations for data movement latency reduction.

Many previous works focus on analyzing various factors that may influence the data movement latency with the hand-crafted design of optimization models [5]–[7], [9], [10]. However, time-variant factors contribute to the latency, including network latency, disk latency and other types of latency (e.g., RAM, CPU, etc.) [11]. Therefore, these static optimization model-based methods are not flexible enough to handle a dynamic system with many uncertainties, such as unreliable network links, changing user request patterns, and evolving system configurations.

Moreover, all research efforts above only consider the data movement latency for the storage location selection. However, the user-experienced service latency is jointly determined by data movement and the follow-on data analytics. Data analytics is the process of inspecting, cleansing, transforming, and modeling raw data to discover useful information for decision-making. Different data analytics frameworks (e.g., MapReduce [12], Dremel [13], and Spark [14]) have been proposed to analyze large volumes of data. For a particular analytical task at a certain scale, the analytical latencies could be different with various frameworks, e.g., tens of minutes for MapReduce, several seconds for Dremel, and sub-seconds for Spark. The analytical latency influences the overall service latency, which should be considered in the data placement problem.

Unlike previous solutions, a generic learning-based framework DataBot+ is proposed, which automatically optimizes the data placement policy with no need for future dynamic information. The investigated data placement problem can be considered as a finite Markov Decision Process (FMDP) as (1) the number of nodes for data storage is limited; (2) each data placement action is independent, and the performance only depends on the current states and placement decisions. Hence, the model-free Q-learning, which is proven to find the optimal action-selection policy for any given FMDP [15], is used in this work. DataBot+

- *K. Liu, J. Peng and Z. Huang are with the School of Information Science and Engineering, Central South University, Changsha, 410075, China. (E-mail: {liukaiyang, pengj, hzw}@csu.edu.cn)*
- *K. Liu, J. Wang, B. Yu, Z. Liao, and J. Pan are with the Department of Computer Science, University of Victoria, Victoria, BC, V8W 2Y2, Canada. (E-mail: {liukaiyang, jingrongwang, boyangyu, zfliao, pan}@uvic.ca)*
- *Z. Liao is with the College of Electrical and Information Engineering, Changsha University of Science & Technology, Changsha, 410114, China.*
- *A preliminary version of this paper appears at IEEE LCN'18 [1].*

can be considered as an agent interacting with the complex environment. This agent selects the storage locations of data items and collects the feedback from the environment, including the current state of request patterns, network conditions, and the resultant end-to-end performance metrics (e.g., the read/write and analytical latencies) due to these actions. Data items with short task execution duration are assigned with higher priorities to optimize the data movement latency, such that the user-experienced latency can be reduced. Through trials and feedbacks, DataBot+ learns the optimal storage locations of data items.

Although as a promising technique, Q-learning may suffer from the curse of dimensionality and the consequently slow convergence with the increasing scale of state/action space. Therefore, a neural network (NN) is maintained in the learning-based framework, approximating to the optimal results with high efficiency and accuracy. Given the current state information as input, NN learns to output the expected rewards of data placement actions. The resultant data read/write and analytical latencies are then utilized as rewards to train the recurrent model, outputting better data placement policies over time.

As the major purpose of our design is to make instantaneous decisions, it must be ensured that the recurrent NN training will not incur extra latency to handle the data flows. The learning-based framework is then decoupled into two asynchronous components, i.e., the production and training system, in the implementation. The online decision making and offline training in parallel change the traditional workflow of reinforcement learning (RL), which requires updating the model after each decision. Therefore, DataBot+ makes instantaneous decisions to query write locations only with the newly trained NN, without introducing extra overheads. The main contributions of this paper are summarized as follows:

1) A generic framework is proposed to learn the optimal data placement policy without assuming the prior request distribution or future dynamic information.
2) Both data read/write and analytical latencies are considered in the storage location selection. Data items with short analytical latency are with higher priorities to optimize the data movement latency.
3) With the increasing number of states/actions, RL is integrated with NN to achieve a quick approximation. Moreover, the online decision making and offline training overcome the deficiency of the framework in delaying the request handling.
4) Driven by real-world I/O traces, large-scale evaluations demonstrate that DataBot+ can lower the user-experienced latency of data service by about 24%.

The rest of this work is organized as follows. Section 2 surveys the related work. Section 3 presents the system architecture of the data placement problem. Section 4 provides the design details of the learning-based data placement framework DataBot+. Section 5 evaluates the performance. Section 6 draws the conclusion and lists future work.

2 Related Work

Existing research efforts have indicated that data placement can enhance the data locality to provide better read/write performances in data-intensive systems. Assuming that the data requests can be predicted accurately, Ren et al. [6] formulated the data purchasing and placement as an integer linear programming problem and designed a close-to-optimal solution to jointly reduce the service cost and latency. By analyzing the workload features, Jalaparti et al. [16] proposed an offline scheduling scheme to jointly places data and tasks to significantly improve the network locality. Through trace analysis, Agarwal et al. [19] proposed Volley, an automated data placement scheme to place data items near end users. Cui et al. [20] constructed a tripartite graph to formulate the data placement problem and proposed a genetic solution to reduce data traffic and latency in clouds. Assuming data request traffic is fairly steady for a certain time, Yu et al. [7] proposed a hypergraph-based data placement scheme among geo-distributed data centers. As a follow-up work, Yu et al. [8] proposed a sketch-based solution for hypergraph sparsification, reducing the algorithm running time. However, all listed previous studies are offline solutions without considering the dynamic information of the system.

As an online scheme, Steiner et al. [17] placed the data items used in the same task to the same location, reducing the inter-rack traffic and task completion time. Chowdhury et al. [18] selected the server with low occupancy links as the storage location of data write flows to lower the task completion time. However, the neglect of the following data read requests may detrimentally influence future read-related performance. Unlike existing solutions, DataBot+ considers both the read/write and data analytical latencies, and uses RL to adaptively learn a better data placement policy with no future assumption about the user requests.

Our work is related to the idea of combining RL and NN to solve complex online decision-making problems [21], [22], but we focus on the data placement problem in the data center network (DCN). Mao et al. [23] used deep reinforcement learning to solve the resource management problem. However, they optimized the expected value of a manually designed objective function on the basis of the reward. Different from this study, Mirhoseini et al. [24] directly used the application execution time as the reward of RL to optimize the device placement with no need of designing intermediate cost models. By collecting the flow level performance metrics in real time, Nie et al. [25] proposed to use group-based RL method to reduce the TCP response latency. Chen et al. [26] developed a two-level deep reinforcement learning system to handle the flow-level traffic optimization. Motivated by previous studies, the early version of our work [1] used end-to-end performance metric as the reward to reduce the overall service latency in the DCN. Metrics such as measured read/write latencies are easier to obtain at lower costs when compared with link-related metrics. DataBot+ extends the scenario in [1] by addressing the considerable influence of data analytics on data placement.

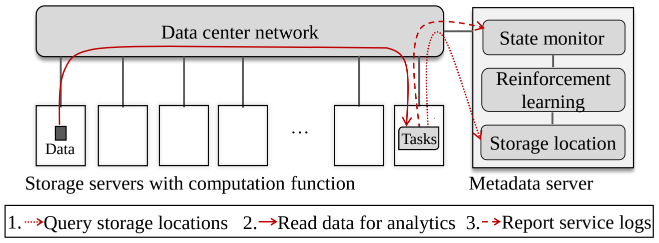

Data center network
State monitor
Reinforcement learning
Storage location
Data
Tasks
Storage servers with computation function
Metadata server
1. Query storage locations 2. Read data for analytics 3. Report service logs

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

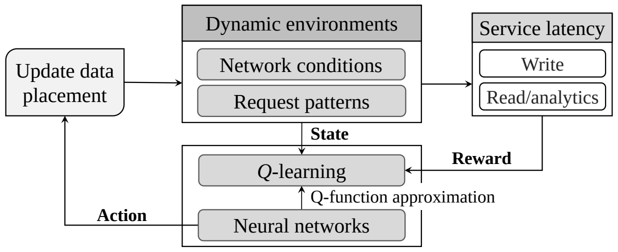


Fig. 2. An overview of the RL-based data placement.

3 System Model and Problem Statement

In this section, we describe the architecture of the storage system and identify the major challenges of the data placement problem.

3.1 System Architecture

The architecture of the storage system is shown in Fig. 1. A set of storage servers or nodes $\mathcal{N}$ (with size $|\mathcal{N}| = N$) is deployed in the distributed storage system. All data items are distributed among storage servers. Each storage server also has computational functions for data analytics. Analytical applications involving multiple data items may require the data movement among storage servers. All servers are connected via a DCN. In order to design a generic data placement solution, we do not rely on any specific DCN topology. Note that our design is only on the basis of the end-to-end measurements of data flows. Our design 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 [27].

Just as in existing systems [28], a centralized metadata server is employed to handle the data storage locations. Let $\mathcal{M}$ represent the set of all data items (with size $|\mathcal{M}| = M$), which can be files, blocks or tables in the system. Let χ_m denote the file size of data item m, $m \in \mathcal{M}$. 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 locations. When an application on a storage server needs to retrieve a data item, it first asks the metadata server where the storage location is through the hashtag. This design ensures that the data storage location is flexible and can be changed with no extra data movement overhead.

As shown in Fig. 1, the metadata server also captures the service logs of the data requests through the state monitor module. As the data request could be read or write, the format of log entries is defined as

$$\begin{cases} (\mathbf{TS}, \mathbf{Src}, \mathbf{Dst}, l^{[\mathrm{R}]}_{ij,m}, l^{[\mathrm{A}]}_{j,m}), \text{if } \mathbf{Request\ type} = \mathbf{Read}, \\ (\mathbf{TS}, \mathbf{Src}, \mathbf{Dst}, l^{[\mathrm{W}]}_{ij,m}), \text{if } \mathbf{Request\ type} = \mathbf{Write}, \end{cases} \tag{1}$$

where **TS** is the timestamp. **Src** and **Dst** are the source and destination nodes of the requests. If the request type is **Read** for data analytics, the end-to-end read latency $l^{[\mathrm{R}]}_{ij,m}$ and the follow-up analytical latency $l^{[\mathrm{A}]}_{j,m}$ are included, where i, j are the source and destination server, respectively, $i, j \in \mathcal{N}$. Otherwise, if the request type is **Write**, the write latency $l^{[\mathrm{W}]}_{ij,m}$ is recorded. The metadata server has all the information by itself except the latencies. Therefore, it is only necessary to report $l^{[\mathrm{R}]}_{ij,m}$, $l^{[\mathrm{A}]}_{j,m}$ and $l^{[\mathrm{W}]}_{ij,m}$ from the storage servers.

The storage system updates the data storage locations when users generate data write requests. For the data-intensive system where the data items are frequently fetched and updated, the proposed learning system is adaptive to the network dynamics. However, for some read-intensive data items with rare writes, hotspots may occur due to the fluctuation of user request patterns. In order to further reduce the data read latency, the storage system can periodically issue "pseudo" write request (i.e., the request issued by the system, not users) to mitigate hotspots. In each time interval with length ϕ, if the data item has not been updated by users, a "pseudo" write operation is executed to update the storage locations. Note that issuing a "pseudo" write request is not an actual data write operation. It triggers the system to recalculate the storage locations of read-intensive data items with rare writes. In this design, extra system overheads introduced by periodical "pseudo" write operations are limited.

3.2 Problem Statement

Due to the non-negligible data movement latency, the storage locations of data items can influence the finish time of distributed analytical tasks. In order to perform low-latency analytical services, the data placement problem can be clarified as follows: how to select the optimal storage locations among all available servers when a data item is to be written or updated?

In the DCN, the service latency includes the data movement latency and the data analytical latency for computing. Moreover, the data movement latency is the sum of a number of components, including the transmission, propagation, processing and queuing latency in the network. It is hard to obtain a precise latency model of the entire system faced with the dynamic scenario, e.g., the changing network conditions and user request patterns. Hence, a generic solution RL is adopted to solve the data placement problem. With RL, the placement of data items can be considered as an agent interacting with the environment and learning the underlying model through the feedback. Unlike the traditional solutions aiming at formulating the mathematical latency models, the used RL chooses an alternative, which is to purely fit with the statistical patterns of the dynamic environment.

4 Q-Learning-based Data Placement

In this section, DataBot+ is presented, which is basically a Q-learning-based solution, to solve the data placement problem. Q-learning is a classic model-free RL technique, which has been demonstrated to find an optimal policy for any given FMDP. The design overview of DataBot+ is first presented, followed by the design details.

4.1 Design Overview

The design overview of DataBot+ is shown in Fig. 2. The main principle of the Q-learning-based data placement can be described by the maintained Q-function:

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

The dynamic information of the storage system (State $\mathcal{S}$) can be learned through intensive data flows to understand which data item should be placed on which node (Action $\mathcal{A}$) so that the corresponding service latencies can be reduced. The observed read/write and analytical latencies are then used as Reward $\mathcal{R}$ to train the recurrent model. In this way, DataBot+ outputs better data placement policies over time in the long term.

More specifically, before a data item m is written into the storage system at time t, its storage locations are chosen according to the current state $\boldsymbol{s}$ and the data placement policy π, $\boldsymbol{s} \in \mathcal{S}$. Then, the action a is executed to place data m to that location, $a \in \mathcal{A}$. Until data item m is updated at t', the latencies of the last written at t and the following read and analytical operations between t and t' can be measured. The weighted sum of the read/write and analytical latencies is used as the immediate reward r_t of the action a. After the update operation at t', the system jumps to another state $\boldsymbol{s}'$. According to [29], the immediate reward r_t at time t still has an impact on the future moments. The optimal Q-value function $Q^*(\boldsymbol{s}, a)$ is the maximum expectation of the long-term reward:

$$Q^*(\boldsymbol{s}, a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \boldsymbol{s}_0 = \boldsymbol{s}, a_0 = a, \pi\right], \quad (2)$$

where $\gamma \in (0, 1)$ is a discount factor. $Q^*(\boldsymbol{s}, a)$ can be achieved through the Bellman equation as follows

$$Q^*(\boldsymbol{s}, a) = \mathbb{E}_{\boldsymbol{s}'}\left[r + \gamma \max_{a'} Q^*(\boldsymbol{s}', a') | \boldsymbol{s}, a\right]. \quad (3)$$

As shown in Fig. 2, future rewards may be affected by many factors in a dynamic environment, such as network conditions and request patterns. The classical RL methods based on temporal differences [30] fail to guarantee a fast convergence to the optimal solution. To solve this challenge, the NN is deployed to approximate the Q-function with high efficiency and accuracy.

4.2 Q-Function Design

4.2.1 States

The key feature of DataBot+ is that the data placement decisions are made only based on the end-to-end measurements of data flows, i.e., the measured read/write latencies. Five categories of state information are included in $\boldsymbol{s}$, which can be derived from the service logs.

Network Conditions: The first category is the network condition, which drastically affects the objective of reducing the read/write latencies:

$$\left\{L_{ij}^{[\mathrm{R}]}, L_{ij}^{[\mathrm{W}]}, i, j \in \mathcal{N}\right\}, \quad (4)$$

where $L_{ij}^{[\mathrm{R}]}$ and $L_{ij}^{[\mathrm{W}]}$ are the average latencies of read/write operations on each pair of servers. The link-based metrics, e.g., bit error rates, are not considered as data placement happens on the application layer. The source/destination of the data flow, not the path or links, is chosen in this paper. Note that the measurement of the link-related metrics would introduce an extra overhead when compared with the end-to-end method. Furthermore, unlike the link-related measurements, the end-to-end method can support any arbitrary data center network topologies. However, our design can coexist with any underlying link-based or path-based flow scheduling.

Then, the real-time network condition is constructed from the logs in (1). Using $l_{ij,m}^{[\mathrm{R/W}]}$ and data size χ_m, the Exponentially Weighted Moving Average (EWMA) method [31] is adopted to estimate the average read/write latency per unit size of data $L_{ij}^{[\mathrm{R/W}]}$. Specifically, after a data read/write operation, $L_{ij}^{[\mathrm{R/W}]}$ is updated by

$$L_{ij}^{[\mathrm{R/W}]} = \alpha_l \frac{l_{ij,m}^{[\mathrm{R/W}]}}{\chi_m} + (1 - \alpha_l) L_{ij}^{[\mathrm{R/W}]}, \quad (5)$$

where α_l is the discount factor to lower the importance of the previous data requests. The advantage of EWMA is that it only needs $O(1)$ space to maintain the prediction for each pair of storage servers.

Data Analytical Latency: The second category is the data analytical latency

$$\left\{L_{j,m}^{[\mathrm{A}]}, j \in \mathcal{N}\right\}, \quad (6)$$

where $L_{j,m}^{[\mathrm{A}]}$ is the estimated analytical latency of data m on the destination server j. The analytical latency is determined by the task type. Computation-intensive tasks generally need more analytical time. Moreover, the analytical latency is also affected by the assigned task priority and the computing workload level of the server. Based on the $l_m^{[\mathrm{A}]}$ in (1), EWMA is also used to estimate the analytical latency of data m on each server

$$L_{j,m}^{[\mathrm{A}]} = \alpha_l l_{j,m}^{[\mathrm{A}]} + (1 - \alpha_l) L_{j,m}^{[\mathrm{A}]}. \quad (7)$$

Note that the storage location i of data item m does not influence the analytical latency on the computing server j. This means the analytical latency cannot be directly optimized with the data placement scheme. How to reduce analytical latency is beyond the scope of this paper. However, the analytical latency should also be considered in the data placement problem as it has non-negligible impacts on the overall user-experienced service latency.

Request Patterns: The third category is the patterns of data requests

$$\left\{F_{i,m}^{[\mathrm{R}]}, F_{i,m}^{[\mathrm{W}]}, \widetilde{F}_{i}^{[\mathrm{R}]}, \widetilde{F}_{i}^{[\mathrm{W}]}, i \in \mathcal{N}\right\}, \quad (8)$$

which contains: 1) read rate to data item m from source server i, $F_{i,m}^{[\mathrm{R}]}$; 2) write rate to m from i, $F_{i,m}^{[\mathrm{W}]}$; 3) read rate to

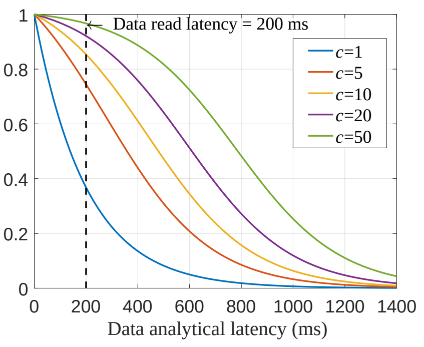


Fig. 3. An example of the weight function $f(l_p^{[R]}, l_p^{[A]})$.

all data items from i, $\widetilde{F}_i^{[R]}$; 4) write rate to all data items from i, $\widetilde{F}_i^{[W]}$. The request patterns reveal how analytical tasks are generating workloads to the storage system, which are the primary cause of the network traffic. With the request information to all data items and the specific data, our design can make a better decision to select storage locations.

The Discounting Rate Estimator (DRE) method [32] is used to construct real-time request pattern information. A counter is maintained for each item in (8), which increases with every read/write operation on each pair of servers, and decreases periodically with a ratio factor $\alpha_r \in (0, 1)$. The benefits of DRE are as follows: (1) it reacts quickly to the changes of the request patterns; (2) it also only requires $O(1)$ space and update time to maintain the prediction for each counter.

Data size: The fourth category is the data size χ_m as it affects the read/write latencies of data m.

Source locations: A 0-1 vector is introduced in state $\boldsymbol{s}$ to indicate whether each storage server is the source of the data write operation or not. The number of data replicas is denoted by k [1]. So we have k servers being 1 in this 0-1 vector. This category should be included as the source locations of the data flow will influence the latency of the write operation.

Each data item has a state with the same size. Given the server number N, the size of state space is

$$|\boldsymbol{s}| = 2N^2 + 6N + 1 = O(N^2). \tag{9}$$

From (9), the number of data items M will not influence the complexity of the learning-based storage system.

4.2.2 Actions

The action set a is also a 0-1 vector, which decides the destination locations of the write operation, $a \in \mathcal{A}$. Similar to the source locations, we have k servers being 1 in the action set for each data item.

4.2.3 Rewards

The main purpose of DataBot+ is to achieve a low-latency service with proper data placement. The read/write and analytical latencies are influenced by time-varying factors. The measured latencies, which include these factors, are directly

1. Similar to the widely used Hadoop Distributed File System (HDFS) [33], the replica number is assumed to be the same for each data item. The data item itself and its replicas are not differentiated in this work, so all of them are considered as data replicas then.

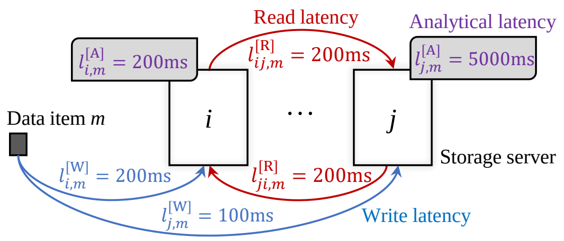


Fig. 4. An example of the storage location selection between server i and j for a data item m when the data read/write and analytical latencies are considered.

utilized to derive the immediate reward. The reward r_t is defined as the weighted sum of latencies measured during time period $[t, t')$, where t is the time of writing data m and t' is the time of updating the same data for the next time:

$$r_t = \omega \cdot \frac{1}{l^{[W]} + \sigma} + (1-\omega) \cdot \frac{1}{|\mathcal{P}|} \cdot \sum_{p \in \mathcal{P}} \frac{1}{l_p^{[R]} + \sigma} \cdot f(l_p^{[R]}, l_p^{[A]}), \tag{10}$$

where $l^{[W]}$ is the write latency at time t, and σ is a pre-defined positive number [2]. $\mathcal{P}$ represents the set of all read operations to data m in $[t, t')$, and $l_p^{[R]}$ is the latency of the read operation $p \in \mathcal{P}$ [3]. As the importance between read and write may be different in various scenarios, the parameter $\omega \in (0, 1)$ is introduced to make the tradeoff. Moreover, $f(l_p^{[R]}, l_p^{[A]})$ is also a weight function which is defined as follows:

$$f(l_p^{[R]}, l_p^{[A]}) = \frac{c}{c - 1 + e^{l_p^{[A]}/l_p^{[R]}}}, \tag{11}$$

where c is also a pre-defined positive integer, and $l_p^{[A]}$ is the analytical latency [4]. Fig. 3 illustrates an example of the weight function with the variation of c. Assuming the data read latency $l_p^{[R]}$ is 200 ms, the value of the weight function decreases with the increase of analytical latency $l_p^{[A]}$. This means that the data item with a lower analytical latency has a higher priority to minimize the user-experienced service latency of data read. In contrast, the data item with longer analytical latency is more focussed on the data write operation. In the extreme case when $l_p^{[A]} \gg l_p^{[R]}$, according to (10), only the data write will be considered. A server with fewer data requests may be selected as the write location to minimize the write latency. In this way, the overall network congestion can be eased to benefit the latency reduction of other data items with short analytical latencies.

Moreover, Fig. 4 illustrates an example of the storage location selection for a data item m. With no data replication, m can be written into storage node i or j with the write latency of 200 ms and 100 ms, respectively. The read latencies are assumed to be 200 ms if m needs to

2. The positive number σ is introduced to prevent $\frac{1}{l^{[R/W]}} \to \infty$ when $l^{[R/W]} = 0$.

3. If no data read happens in $[t, t')$, i.e., $\mathcal{P} = \varnothing$, only data write is considered, $r_t = \omega \cdot \frac{1}{l^{[W]}}$.

4. Please note that not all user-generated requests will incur data analytics after data movement, e.g., pre-processed data access. If no analytics is incurred, i.e., $l_p^{[A]} = 0$, the weight function $f(l_p^{[R]}, l_p^{[A]}) = 1$. According to (10), only data read/write latencies will be considered and optimized in this case. Therefore, the proposed DataBot+ can be applied to various scenarios with or without data analytics.

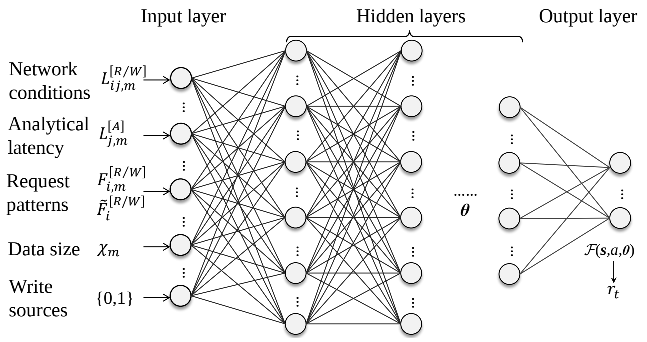

Input layer
Hidden layers
Output layer
Network conditions $L_{ij,m}^{[R/W]}$
Analytical latency $L_{j,m}^{[A]}$
Request patterns $F_{i,m}^{[R/W]}$ $\bar{F}_{i}^{[R/W]}$
Data size χ_m
Write sources {0,1}
θ
$\mathcal{F}(s,a,\theta)$
r_t

Fig. 5. NN structure for data placement.

be fetched from the storage node (i or j). The follow-up analytical latencies of m at node i and j are 200 ms and 5,000 ms, respectively. For the read-optimized scenario $\omega = 0.2$, without considering the data analytical latency, node j will be selected as the storage location to maximize the reward (with $c = 50$ and $\sigma = 0.1$). The overall service latencies of the analytical tasks at node i and j are 400 ms and 5,000 ms, respectively. In contrast, by considering the analytical latency in the reward function, our design tends to reduce the data movement latency of the task with short analytical latency. Therefore, node i is selected as the storage location. The service latencies will be 200 ms and 5,200 ms, respectively. This means that the service latency of the long analytical task is only increased by 4%. In return, the service latency of the short analytical task can be reduced by 50%. This example shows the benefits of the reward function design.

4.3 Asynchronous Implementation

Then, we show how to reduce the size of state space caused by the number of servers N. For Q-function approximation, an NN is maintained in the Q-learning-based system. Fig. 5 illustrates the structure of NN, which contains three kinds of layers, i.e., the input layer, hidden layers, and the output layer. Each layer has a number of computing neurons. Given the current state $\boldsymbol{s}$ as input and the reward of Q-learning r_t as output, the NN updates the weights of connections $\boldsymbol{\theta}$ between layers of neurons. With the training process of approximation, the NN learns the weights to output the expected rewards of data placement actions with high efficiency and accuracy.

The traditional workflow of RL requires updating the model after each decision [21], [30]. The recurrent training process will introduce extra latencies to handle the requests, which are undesirable for data center applications. Unlike previous studies, the design objective is that DataBot+ can make instantaneous decisions for the requests of querying write locations. As illustrated in Fig. 6, the learning system is decoupled into two components, i.e., the production and training system, in the implementation. They work asynchronously to ensure that the training process of NN will not introduce extra overheads to handle the data requests in the production system.

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
```

4.3.1 *Production System*

As shown in Fig. 6, the production system serves the requests for updating the storage locations via the decision NN. Given a state $\boldsymbol{s}_t$ as input and the current weight vector $\boldsymbol{\theta}$, the maintained NN can output a vector $\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$ (with size $|\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})| = N$). Each element in $\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$ represents the estimated reward of writing the data item to the corresponding servers.

Algorithm 1 lists the pseudo code of the production system. When a data write/update request is submitted to the metadata server at time t, the ϵ-greedy method is applied here to select the action. Following the uniform distribution, a random variable $\eta \in [0, 1]$ is generated. If $\eta < \epsilon$, the action a_t^* that maximizes the output value of $\mathcal{F}(\boldsymbol{s}_t, a_t, \boldsymbol{\theta})$ is selected to obtain a lower read/write latency. Otherwise, a random action will be selected to search for the unexplored solution space. When the storage locations are updated with action a_t, the system is transitioned into a new state $\boldsymbol{s}_{t+1}$. The reward r_t can be observed at the end of time interval t. A tuple

$$\tau = (\boldsymbol{s}_t, a_t, \boldsymbol{s}_{t+1}, r_t) \tag{12}$$

is stored for each update of the data storage location. All tuples in a certain period constitute the replay memory $\mathcal{R}$, which can be used for NN training in the training system [5].

4.3.2 *Training System*

The training system replays the tuples in the relay memory $\mathcal{R}$ periodically to train an updated weight vector $\boldsymbol{\theta}^+$ for the future decisions in the production system. Algorithm 2 lists the pseudo code of the training system. The mini-batch stochastic gradient descent (SGD) method [29] is adopted to update the weight vector, minimizing the difference between the NN output and the target value. According to the observed reward r_t in (10), the target value is calculated as follows

$$y_t = r_t + \gamma \max_{a_{t+1}} \mathcal{F}(\boldsymbol{s}_{t+1}, a_{t+1}, \boldsymbol{\theta}). \tag{13}$$

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

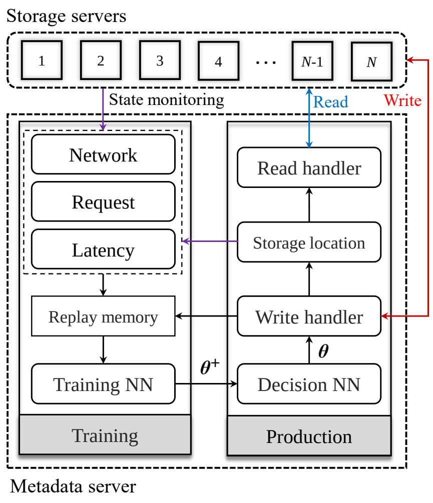


Fig. 6. Production and training system of DataBot+.

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 $i \in \mathcal{I}$ 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 (14);
9:     end for
10:    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^+$;
11:  end for
12: end if
```

With SGD, all tuples in $\mathcal{R}$ are split into several subsets, i.e., mini-batches $\mathcal{B}$. For each mini-batch $b \in \mathcal{B}$, the weight vector $\boldsymbol{\theta}^+$ is updated by the gradient method to minimize the difference between the expected reward and the output of NN

$$\mathbb{E}_{(\boldsymbol{s},a)\sim\mathcal{B}} \left[\left(y - \mathcal{F}\left(\boldsymbol{s}, a, \boldsymbol{\theta}^+\right) \right)^2 \right]. \tag{14}$$

In the training process, all mini-batches are trained with multiple iterations to converge faster. The iteration is termed as epoch i, $i \in \mathcal{I}$. The weight vector $\boldsymbol{\theta}$ of the decision NN keeps 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 the weight of decision NN $\boldsymbol{\theta}$. This variation of the training process makes the optimization objective more stable and therefore avoids fluctuations to some degree.

Using the SGD training method, [34] proved that the NN can converge to the global optimum at a linear rate if the initial weights of NN are approximately balanced and the initial end-to-end matrix has positive deficiency margin. This ensures that the maintained NN can approximate to the optimal results with high accuracy and efficiency.

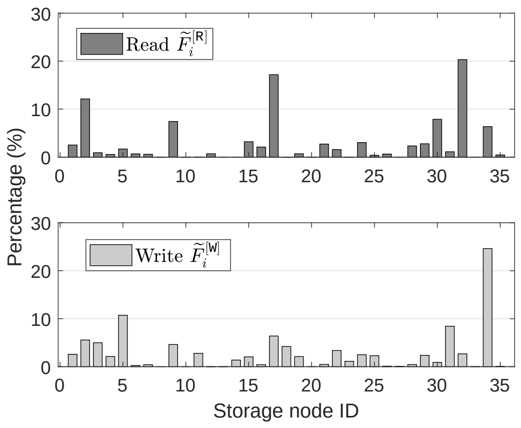


Fig. 7. The read/write request arrival rates of data items based on the MSR Cambridge Traces are shown. The request distribution is biased among 36 storage servers.

5 Performance Evaluation

Extensive evaluations driven by real-world I/O traces, i.e., MSR Cambridge Traces [35], are conducted to evaluate the performance of the proposed DataBot+.

5.1 Data Trace Description and Experiment Settings

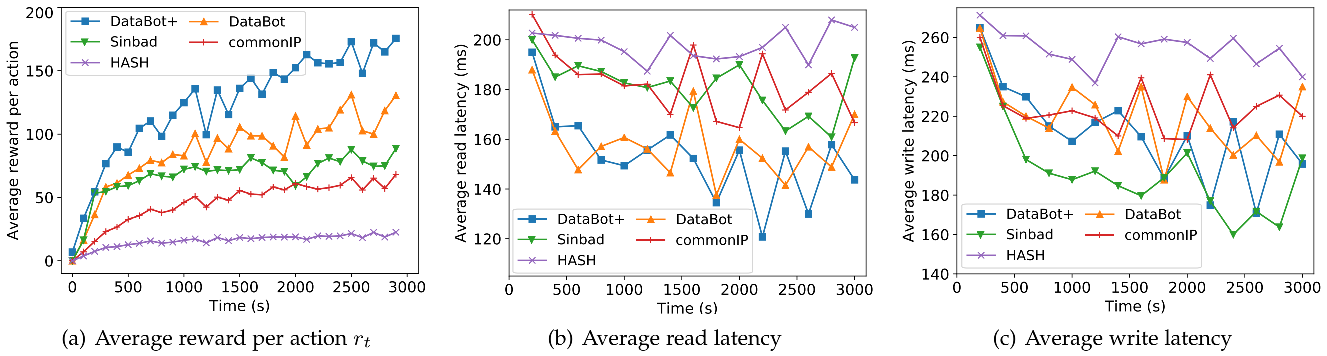
MSR Cambridge Traces: These traces are gathered from an enterprise data center at Microsoft Research Cambridge, where data read/write requests are recorded from 36 servers for a week. The arrival rates of the read/write requests to all data items $\widetilde{F}_i^{[\mathsf{R/W}]}$ are shown in Fig. 7. The request distribution is biased among 36 servers due to real applications. The hostname, request type (read/write), transferred traffic size, and timestamp information are provided for each request. For the reason of confidentiality, most publicly available traces, including MSR Cambridge Traces, do not specify the detailed data items for each request. The total number of data items is assumed to be $M = 10,000$ in the storage system. Similar to the previous studies [6], [7], the request rates of data items $F_{i,m}^{[\mathsf{R/W}]}$ in server i follow a Zipf distribution with $\widetilde{F}_i^{[\mathsf{R/W}]} = \sum_{m=1}^{M} F_{i,m}^{[\mathsf{R/W}]}$.

Furthermore, the short-lived tasks occupy most of the tasks in the cloud analytical system [36]. An experimental study of response time on Amazon EC2 illustrated that the latency distribution has long tails [37]. A workload analysis at Microsoft Bing also demonstrated that data analytical latencies have long tail features [38]. Therefore, the data analytical latencies are generated following the long tail power-law distribution, which ranges widely from 50 ms to 300 s [39].

Experiment Settings: The experiments are conducted on a Dell XPS 15 9560 with an Intel(R) Core i7-7700 processor running at 2.8 GHz. This machine features 16 GB of RAM and an NVIDIA GeForce GTX 1050 graphics card. Mininet is adopted to emulate the DCN, which can create a high fidelity network running Linux-based applications [40]. All nodes are implemented in a typical Fat-Tree topology with 3 layers of network switches. All link capacities in the topology are set to 1 Gbps. At each storage node, a client program is deployed to initiate the read/write requests

TABLE 1
Average data read latencies in different analytical latency intervals (ms)

Interval of analytical latency	50 ∼ 200	200 ∼ 400	400 ∼ 800	800 ∼ 1400	1400 ∼ ∞
Average analytical latency	95.6	277.3	534.5	1,059.3	15,628.3
HASH	200.2	199.0	200.5	202.5	202.5
commonIP	175.6	172.5	175.4	179.0	175.3
Sinbad	172.2	173.1	172.9	173.8	171.4
DataBot	155.2	154.3	156.9	157.3	155.4
DataBot+	129.8	141.8	143.6	144.2	149.6



(a) Average reward per action r_t (b) Average read latency (c) Average write latency

Fig. 8. Read-optimized scenario $\omega = 0.2$ with replica $k = 3$.

based on the MSR Cambridge Traces. To enhance the access efficiency for intensive data flows, Memcached [41] module is adopted as the end of data flows and caches data items in RAM. In brief, each node has a client being the request source and a Memcached being the destination. A metadata server program is also implemented to handle the control flows, whose modules include state monitoring, write destination decision and NN training.

TensorFlow-GPU [42] is used as the learning platform to deploy DataBot+. Keras [43] is used as the framework for NN implementation. Multilayer perceptron (MLP) [44] with one kernel is used as the structure of NN. Determined by (9), the NN has 2,809 features in the input layer and 36 features in the output layer. The NN is initialized based on [34] to ensure the training convergence at a linear rate. Firstly, the dimensions of hidden layers should be at least the minimum dimension of the input and output layers. Therefore, three hidden layers are deployed, which have 2,000, 1,000 and 400 neutrons, respectively. Then, the weight matrices are initialized following the random Gaussian distribution with zero means. In order to mitigate hotspots with “pseudo” write, the time length ϕ is set to 30 s.

Performance Baselines: In order to evaluate the performance of DataBot+, four baselines HASH, commonIP, Sinbad, and DataBot are investigated. HASH hashes data items to storage nodes for load-balancing, which has been widely used in the current distributed storage systems, such as HDFS [33] and Cassandra [45]. commonIP [19] places data items to the IP address that requests them the most. As an online solution, Sinbad [18] leverages the network flexibility to avoid congested links. Therefore, the network hotspots can be eased in replica placement during data writes. DataBot [1] was proposed in the conference version of our paper without considering the influence of data analytical latency.

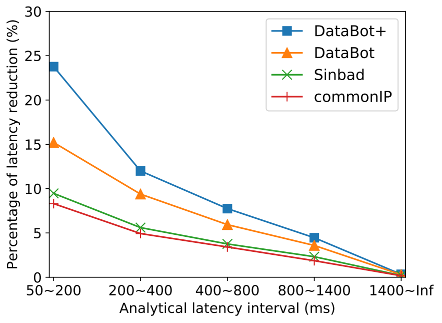
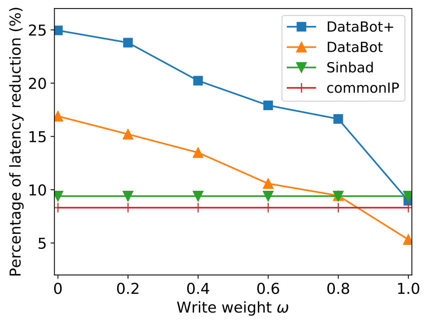


Fig. 9. Percentage of user-experienced latency reduction when compared with the worst performance obtained by HASH in different analytical latency intervals.

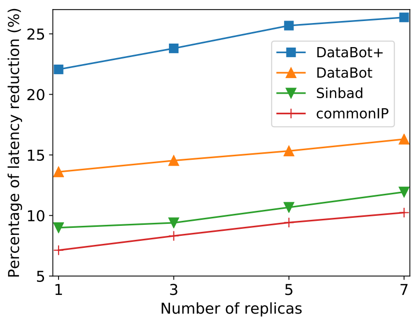
5.2 Experiment Results

To start with, the read-optimized scenario (weight $\omega = 0.2$ and replica $k = 3$) is evaluated with the running time of 3,000 s. According to the evaluation results, 3,000 s is a long enough period to obtain a steady performance improvement ratio. The constant c in the reward function is set to 5. The latest $R = 2,000$ tuples are captured in the replay memory and trained with $|\mathcal{I}| = 6$ epochs and $|\mathcal{B}| = 300$ batch size. Each round of training needs 8.498 s on average before the updated weight vector $\boldsymbol{\theta}^+$ is transferred to the decision NN in the production system. This means that without the asynchronous implementation, the NN training will introduce extra 8.498 s of latency to the data write requests, which is undesirable for data center applications. This demonstrates the benefits of the asynchronous implementation.

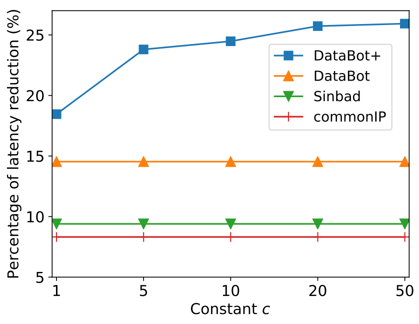
Fig. 8(a) illustrates the average reward per data placement decision. In the beginning, the read/write latencies decrease as Memcached should be warmed up with data



(a) Impact of write weight ω

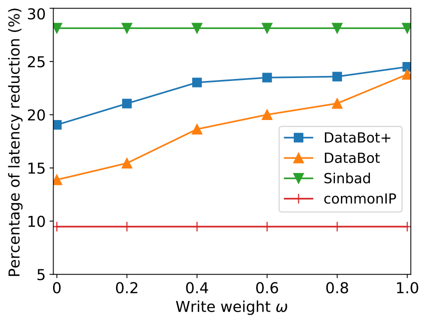


(b) Impact of replica number k

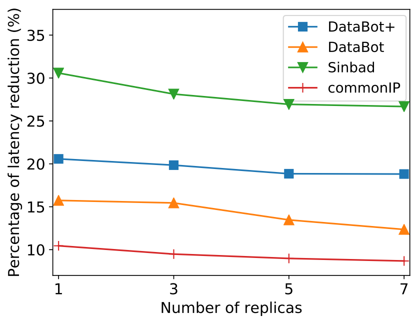


(c) Impact of constant c

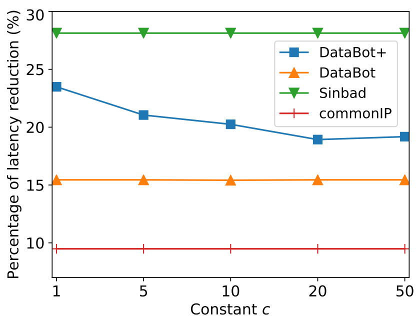
Fig. 10. Percentage of **read and analytical** latency reduction when compared with the worst performance obtained by HASH for data items with short analytical latency (ranging from 50 ms to 200 ms).



(a) Impact of write weight ω



(b) Impact of replica number k



(c) Impact of constant c

Fig. 11. Percentage of **write** latency reduction when compared with the worst performance obtained by HASH for data items with short analytical latency (ranging from 50 ms to 200 ms).

items. Hence, the average rewards of Sinbad, commonIP and HASH increase for the first hundreds of seconds, and remain fairly stable then. In comparison with the heuristic solution, DataBot+ and DataBot learn better data placement policies through trials and feedbacks continuously. The average reward is in an increasing trend with the learning-based process. After multiple rounds of training for convergence, the performance improvement ratio with DataBot+ and DataBot becomes stable after 1,000 s.

Fig. 8(b) and (c) show the average read/write latencies measured in every 200 s of the experimental period. HASH can be intuitively treated as random storage location selection regardless of the user request patterns or network conditions. Therefore, for the last 1,000 s, HASH incurs the highest read/write latencies at about 201.0 ms and 249.9 ms, respectively. commonIP places the data to the storage servers which request them the most. However, with the more queuing delay caused by the biased distribution of data requests among servers, the average read/write latencies are 175.6 ms and 226.2 ms, respectively. Sinbad places the data to the server with low occupancy links with the lowest write latency 179.6 ms. However, the following data read is neglected with the read latency 172.4 ms.

With the learning process for a better placement policy, DataBot+ and DataBot achieve lower read latencies than HASH, commonIP, and Sinbad. As our previous work, DataBot treats all data items equally to reduce the average read/write latencies to 156.8 ms and 211.3 ms, respectively. In contrast, as shown in Table 1, DataBot+ tends to reduce the data movement latency of the data item with a short analytical latency. All data items are divided into 5 different groups according to the data analytical latencies. Without considering the analytical latency, the average read/write latencies achieved by HASH, commonIP, and DataBot are barely the same for all data groups. According to (10), with DataBot+, data items with shorter analytical durations are assigned with higher priorities to reduce the read latency. Data items with long analytical latencies pay more attention to reduce the write latency by selecting a server with fewer data requests as the write location. In this way, the overall network congestion can be eased, which also benefits the read/write latency reduction of other data items with short analytical latency. Specifically, as shown in Fig. 9, for data items with analytical latency ranging from 50 ms to 200 ms, the user-experienced latency of data read and analytics can be reduced the most with DataBot+ (up to 23.8%). Furthermore, compared with DataBot, DataBot+ periodically issues "pseudo" write operations to mitigate hotspots. The average read/write latencies of all data items can be reduced to 141.5 ms and 193.9 ms, respectively.

Furthermore, as shown in Fig. 8, the performance shows some variance due to 1) the fluctuations of the user request patterns, and 2) the needed exploration with the ϵ-greedy method. The current data placement decisions may not always yield the optimal data read/write latencies due to the dynamic user requests and the exploration process. The zigzag range becomes smaller with the learning process, showing its converging adaptivity.

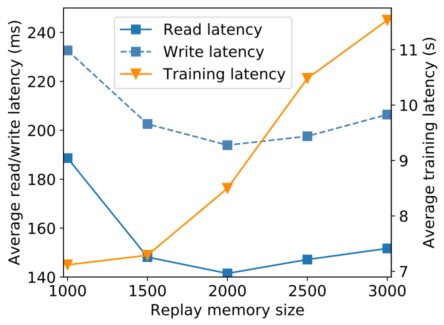


Fig. 12. Impact of replay memory size $|\mathcal{R}|$.

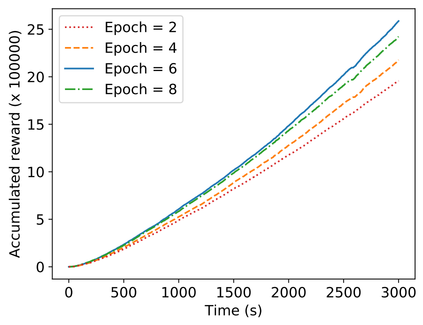


Fig. 13. Impact of epoch number $|\mathcal{I}|$.

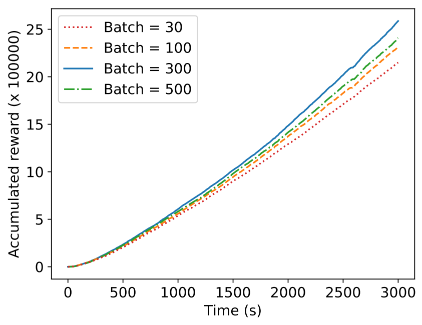


Fig. 14. Impact of batch size $|\mathcal{B}|$.

5.3 Parameter Impacts

In order to fully evaluate the performance of DataBot+, several factors which may affect the data placement process are also considered. Compared with the worst performance obtained by HASH, Fig. 10 and 11 show the percentage of latency reduction for data items with short analytical latency (ranging from 50 ms to 200 ms).

Write weight ω: Fig. 10(a) and 11(a) illustrate the impact of write weight ω. The average read/write latencies with HASH, commonIP, and Sinbad remain roughly stable as ω is not considered in the heuristic solutions. When ω increases from 0 to 1.0, the priority of write requests with becomes higher and higher for DataBot+ and DataBot. With DataBot+, the average write latency is decreased by 6.72% (from 202.3 ms to 188.7 ms), while the read latency is increased by 37.42% (from 126.4 ms to 173.7 ms). For the read-optimized scenario ($\omega = 0$), compared with HASH, the user-experienced latency of data read and analytics with DataBot+, DataBot, Sinbad, and commonIP can be reduced by 24.94%, 16.9%, 9.4%, and 8.32%, respectively. For the write-optimized scenario ($\omega = 1.0$), compared with HASH, the write latency can be reduced by 24.5%, 23.79%, 28.13%, and 9.48% for DataBot+, DataBot, Sinbad, and commonIP, respectively.

Number of Replicas k: Data replication can enhance the reliability, accessibility, and fault-tolerance of the data service. When the replica number is increased from 1 to 7, the network congestion due to read requests can be eased with more options to access needed data. The average read latency of DataBot+ is decreased from 151.2 ms to 103.6 ms.

Data writes are synchronous to provide strong consistency in this work. Once a data item is updated, its storage node acts as the source to synchronize the updated data with all nodes having data replicas. The write latency is increased from 195.3 ms to 218.9 ms as the data item must be written into k different locations with more data write flows. In the future work, the asynchronous writing model will be investigated to choose the relay nodes among all storage nodes, reducing the data write latency. Fig. 10(b) and 11(b) illustrates that under the replication setting, DataBot+ can choose better storage locations than other schemes with lower data service latencies.

Constant c: As shown in Fig. 3, for a data item with the same analytical latency, the importance of read request in (11) goes up with the increase of c. This means that a larger c indicates a higher priority of read requests but with less concern for write requests. When c is increased from 1 to 50, the average read latency with DataBot+ is decreased from 145.5 ms to 123.5 ms, while the write latency is increased from 191.2 ms to 201.9 ms. In contrast, the read/write latencies with DataBot, Sinbad, commonIP, and HASH keep stable without considering c. As shown in Fig. 10(c), with the increase of c, DataBot+ achieves more and more user-experienced latency reduction for the data read and analytical operation. At the same time, the reduction of data write latency will be decreased.

Replay Memory Size $|\mathcal{R}|$: Fig. 12 illustrates the impact of replay memory size $|\mathcal{R}|$. When $|\mathcal{R}|$ is increased from 1,000 to 2,000, the learning system can approximate the optimal solution more precisely with more tuples. The average read/write latencies are decreased by 25% and 16.6%, respectively. In the meantime, the training latency is increased from 7.117 s to 8.498 s.

When $|\mathcal{R}|$ is further increased from 2,000 to 3,000, the training latency is increased rapidly from 8.498 s to 11.533 s. The adaptability of the learning system to network dynamics decreases accordingly. The average read/write latencies are increased by 7.2% and 6.4%, respectively.

Number of Training Epochs $|\mathcal{I}|$: The NN training and the resultant performance of data placement are influenced by the number of training epochs. Fig. 13 illustrates the change of the accumulated reward with the variation of $|\mathcal{I}|$. When $|\mathcal{I}|$ is increased from 2 to 6, the accumulated reward increases because more rounds of training processes lower the difference between the expected reward of learning and the output of NN. Nevertheless, when $|\mathcal{I}|$ is set to 8, performance degradation can be observed due to over-fitting. During the training process, even though the loss function can be further reduced with more training rounds, the obtained model may lose its generalization capability for the future samples. Therefore, $|\mathcal{I}|$ is set to 6 in the experiments.

Batch Size $|\mathcal{B}|$: The batch size determines how frequently the weight vector θ is updated during the training process. As can be seen in Fig. 14, when $|\mathcal{B}|$ is set to 300, the highest accumulated reward can be achieved due to the same reason above. Fig. 13 and 14 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.

6 Conclusion and Future Work

In order to handle the uncertainties of the dynamic system, this paper proposes a novel learning-based data placement framework DataBot+, to automatically learn the optimal data placement policies. The NN is utilized to estimate the near-future latency by training the weight vector with the Q-values, thus avoiding the complexity of the huge state space and speeding up the convergence to the solution. Data items with short analytical latencies are more sensitive to the variation of the data movement latency. They are assigned with higher priorities to maximize the reduction of the user-experienced service latency. Furthermore, two asynchronous components, i.e., the online decision making and offline training, are integrated seamlessly to ensure that no extra delays will be introduced to handle the intensive data flows. Performance evaluation demonstrates that the user-experienced service latencies are reduced when compared with the state-of-the-art solutions. For the scalability in the future work, the distributed RL solutions can be explored to further 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] K. Liu, J. Wang, Z. Liao, B. Yu, and J. Pan, "Learning-based adaptive data placement for low latency in data center networks," *in Proc. of IEEE LCN*, pp. 142–149, 2018.
[2] 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.
[3] Q. Pu, G. Ananthanarayanan, oP. 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.
[4] 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. of VLDB Endow.*, vol. 4, no. 9, pp. 575–585, 2011.
[5] Y. Xiang, T. Lan, V. Aggarwal, and 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.
[6] 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.
[7] B. Yu and J. Pan, "A framework of hypergraph-based data placement among geo-distributed datacenters," *IEEE Trans. Serv. Comput.*, 2017.
[8] B. Yu and J. Pan, "Sketch-based data placement among geo-distributed datacenters for cloud storages," *in Proc. of IEEE INFOCOM*, pp. 1–9, 2016.
[9] 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.
[10] 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.
[11] "Latency Definition." [Online]: https://techterms.com/definition/latency
[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[13] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *Commun. ACM*, vol. 54, no. 6, pp. 114–123, 2011.
[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *in Proc. of USENIX NSDI*, pp. 15–28, 2012.
[15] L.P. Kaelbling, M.L. Littman, and A.W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.
[16] 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.
[17] 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.
[18] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," *in Proc. of ACM SIGCOMM*, pp. 231–242, 2013.
[19] 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.
[20] L. Cui, J. Zhang, L. Yue, Y. Shi, H. Li, and D. Yuan, "A genetic algorithm based data replica placement strategy for scientific applications in clouds," *IEEE Trans. Serv. Comput.*, vol. 11, no. 4, pp. 727–739, 2018.
[21] 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.
[22] I. Bello, H. Pham, Q.V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2017.
[23] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," *in Proc. of ACM HotNets*, pp. 50–56, 2016.
[24] 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 ICML*, pp. 2430–2439, 2017.
[25] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, and J. Zhang, "Reducing web latency through dynamically setting TCP initial window with reinforcement learning," *in Proc. of IEEE/ACM IWQoS*, pp. 1–10, 2018.
[26] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," *in Proc. of ACM SIGCOMM*, pp. 191–205, 2018.
[27] 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.
[28] Q. Xu, R.V. Arumugam, K.L. Yong, and S. Mahadevan, "Efficient and scalable metadata management in EB-scale file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 2840–2850, 2014.
[29] 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.
[30] 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.
[31] J.S. Hunter, "The exponentially weighted moving average," *J. Qual. Technol.*, vol. 18, no. 4, pp. 203–210, 1986.
[32] 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.
[33] "HDFS Architecture Guide." [Online]: https://hadoop.apache.org/
[34] S. Arora, N. Cohen, N. Golowich, and W. Hu, "A convergence analysis of gradient descent for deep linear neural networks," *in in Proc. of ICLR*, 2019.
[35] 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.
[36] P. Delgado, F. Dinu, A. M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," *in Proc. of USENIX NSDI*, pp. 499–510, 2015.
[37] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," *in Proc. of USENIX NSDI*, pp. 329–341, 2013.
[38] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," *in Proc. of ACM SIGCOMM*, pp. 219–230, 2013.
[39] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," *in Proc. of ACM SOSP*, pp. 69–84, 2013.

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

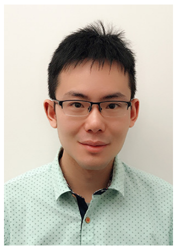
[41] "Memcached." [Online]: http://memcached.org/

[42] "TensorFlow," [Online]. Available: https://www.tensorflow.org

[43] "Keras," [Online]. Available: http://keras.io/

[44] 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.

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



Kaiyang Liu (S'14) received his Ph.D. degree in the School of Information Science and Engineering, Central South University in 2019. During 2016–2018, he was a research assistant at University of Victoria, Canada, with Prof. Jianping Pan. His current research areas include networked systems, distributed systems and cloud/edge computing, with a special focus on the analysis and optimization of the data-intensive services. One of his paper is one of three IEEE LCN 2018 Best Paper Award candidates.

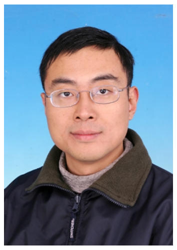
Jun Peng (M'08) received the B.S. Degree from Xiangtan University, Xiangtan, China, in 1987, the M.Sc. degree the National University of Defense Technology, Changsha, China, in 1990, and the Ph.D degree from Central South University, Changsha, China in 2005. In April 1990, she joined the Central South University. From 2006 to 2007, she was with School of Electrical and Computer Science, University of Central Florida, USA, as a visiting Scholar. She is a Professor with the School of Computer Science and Engineering, the Central South of University, China. Her research interests include cooperative control, cloud computing, and wireless communications.

Jingrong Wang (S'18) received her B.S. degrees in 2017 in the School of Electronic and Information Technology, Beijing Jiaotong University. She is currently an MSc. student in the Dept. of Computer Science at the University of Victoria. Her research interests cover wireless communications, mobile edge computing, and machine learning. She is a coauthor of one of three IEEE LCN 2018 Best Paper Award candidates.

Boyang Yu received his PhD degree in the Department of Computer Science at the University of Victoria, Canada, in 2016. He received his Bachelor Degree and Master Degree in Computer Science from Nankai University, China, in 2006 and 2009, respectively. His current research areas include networked systems, distributed systems and cloud computing, with special focus on the analysis and optimization in the data-intensive services.

Zhuofan Liao received the Ph.D. degree in computer science from Central South University, China, in 2012. Supported by the China Scholarship Council, she was a Visiting Scholar with the University of Victoria, from 2017 to 2018. She is currently an Assistant Professor with the School of Computer and Communication Engineering, Changsha University of Science and Technology, China. Her research interests include wireless networks optimization, big data, and edge computing.

Zhiwu Huang (M'08) received the B.S. in industrial automation degree from Xiangtan University, Xiangtan, China, in 1987, the M.S. degree in industrial automation from the Department of Automatic Control, University of Science and Technology Beijing, Beijing, China, in 1989, and the Ph.D degree in control theory and control engineering from Central South University, Changsha, China, in 2006. In October 1994, he joined the Central South University. From 2008 to 2009, he was with School of Computer Science and Electronic Engineering, University of Essex, U.K., as a visiting Scholar. He is currently a Professor with the School of Automation, Central South University, China. His research interests include fault diagnostic technique and cooperative control.



Jianping Pan (S'96-M'98-SM'08) is currently a professor of computer science at the University of Victoria, Victoria, British Columbia, Canada. He received his Bachelor's and PhD degrees in computer science from Southeast University, Nanjing, Jiangsu, China, and he did his postdoctoral research at the University of Waterloo, Waterloo, Ontario, Canada. He also worked at Fujitsu Labs and NTT Labs. His area of specialization is computer networks and distributed systems, and his current research interests include protocols for advanced networking, performance analysis of networked systems, and applied network security. He received the IEICE Best Paper Award in 2009, the Telecommunications Advancement Foundation's Telesys Award in 2010, the WCSP 2011 Best Paper Award, the IEEE Globecom 2011 Best Paper Award, the JSPS Invitation Fellowship in 2012, the IEEE ICC 2013 Best Paper Award, and the NSERC DAS Award in 2016, is a coauthor of one of three IEEE LCN 2018 Best Paper Award candidates, and has been serving on the technical program committees of major computer communications and networking conferences including IEEE INFOCOM, ICC, Globecom, WCNC and CCNC. He was the Ad Hoc and Sensor Networking Symposium Co-Chair of IEEE Globecom 2012 and an Associate Editor of IEEE Transactions on Vehicular Technology. He is a senior member of the ACM and a senior member of the IEEE.