Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

CLQLMRS: improving cache locality in MapReduce job scheduling using Q-learning

Rana Ghazali[1], Sahar Adabi[1*], Ali Rezaee[2], Douglas G. Down[3] and Ali Movaghar[4]

Abstract

Scheduling of MapReduce jobs is an integral part of Hadoop and effective job scheduling has a direct impact on Hadoop performance. Data locality is one of the most important factors to be considered in order to improve efficiency, as it affects data transmission through the system. A number of researchers have suggested approaches for improving data locality, but few have considered cache locality. In this paper, we present a state-of-the-art job scheduler, CLQLMRS (Cache Locality with Q-Learning in MapReduce Scheduler) for improving both data locality and cache locality using reinforcement learning. The proposed algorithm is evaluated by various experiments in a heterogeneous environment. Experimental results show significantly decreased execution time compared with FIFO, Delay, and the Adaptive Cache Local scheduler.

Keywords: Job scheduler, Data locality, Cache locality, Reinforcement learning, Q-learning

Introduction

Hadoop [1] is one of the most popular frameworks for parallel processing in a distributed environment. The two main components of Hadoop are HDFS and MapReduce. HDFS is a file system for storing identically sized blocks of data and MapReduce is a programming model for analyzing large amounts of data consisting of two types of tasks: Map and Reduce. Hadoop employs a master/worker architecture where a cluster includes a single *NameNode* as a master node, while the remaining nodes are DataNodes that serve as worker nodes. The *NameNode* stores metadata and determines the mapping of data blocks and their replicas into DataNodes. The job scheduler has a pivotal role in Hadoop performance with one of its main goals being minimizing job execution time and overhead, as a means for maximizing throughput and resource utilization. The main responsibility of the job scheduler is to allocate tasks to appropriate nodes, in particular attempting to choose those that have input data for processing these tasks. In other words, the scheduler moves processes instead of data and attempts to decrease data transmission through the network. One of the critical factors with a direct impact on improving performance is data locality, the distance between the input data node and the processing node. As a result, the improvement of data locality in job schedulers has become an important research topic. One aspect of data locality that has achieved little attention is cache locality.

In-memory caching [2] has been applied in Hadoop 2.3.0 to address the I/O bottleneck in HDFS. Caching of the input data leads to maximizing memory

*Correspondence: Sahar_adabi@iau-tnb.ac.ir

[1] Department of Computer Engineering, North Tehran Branch, Islamic Azad University, Tehran, Iran
Full list of author information is available at the end of the article

locality and provides high data availability. Requested data can be retrieved from the cache avoiding the significant overhead of retrieving the data from HDFS. In this case, the *NameNode* manages and updates cluster cache states by collecting cache reports from DataNodes. Also, the *NameNode* receives heartbeat messages from DataNodes to update the data location information in its metadata. Therefore, the job scheduler itself does not have any information about data location or cached data and uses these metadata that exist in the *NameNode*. Furthermore, the job scheduler can utilize cache locality in its decision to assign tasks to nodes that have cached the required input data, leading to reduced data transmission and reduced job completion times. If most of the requested data is cached, a significant performance improvement ensues. There are a few job scheduler mechanisms that take into account cache locality in making decisions, but to the best of our knowledge, none have applied machine learning methods.

In this paper, we propose a novel job scheduler based on Q-learning for improving data and cache locality. The main reason for using the Q-learning algorithm is that it does not require any training data or prior information about the environment; it learns through interaction with the environment. As a result, it generates minimal overhead. The fact that no information is required about the environment (Q learning is model-free) is crucial for our setting, as being workload-dependent, the environment is very difficult to model.

Our contributions to this paper are:

- We provide a brief overview of job schedulers that consider cache locality and discuss their advantages and disadvantages.
- We introduce reinforcement learning concepts, particularly Q-learning and the SARSA algorithm which are used in our proposed job scheduler.
- We propose a novel MapReduce job scheduler, CLQLMRS, by applying Q-learning to train a scheduling policy that maximizes the local tasks rate.
- We design CLQLMRS based on an associated MDP model.
- We describe the CLQLMRS implementation.
- We evaluate CLQLMRS's performance and compare it with other job schedulers including *FIFO, Delay, and Adaptive Cache Local scheduling*.
- We carry out experiments to investigate the impact of different factors on locality rates.

The structure of this article is as follows. Section 2 defines the problem and our solution approach. Related work overviews related work that considers cache locality in scheduling decisions. In Reinforcement learning, we perform a literature review and introduce necessary basic concepts that are used in the proposed scheduler. We describe our novel job scheduler based on Q-learning for improving data and cache locality in Reinforcement learning and Markov Decision Process (MDP) model. The CLQLMRS implementation steps are presented in Q-Learning and the SARSA algorithm. We evaluate the performance of the proposed job scheduler via different experiments in Design of CLQLMRS. Finally, Modeling cache-locality scheduling as an MDP contains the conclusion and future work.

Problem definition

Reducing job execution time has a positive impact on improving Hadoop performance. It is also one of the important goals of a job scheduler in assigning tasks to the worker nodes. Increasing the local tasks rate is the one way to achieve this goal, therefore we should take into account different levels of locality in assigning tasks to the nodes as an approach to tackle this problem.

Although there are many job schedulers that benefit from data locality in their scheduling decisions. These opportunities exist due to the fact that existing MapReduce schedulers do not consider all levels of locality for both Map and Reduce tasks and rarely use cache locality in their scheduling mechanisms. To address this issue, we introduce an intelligent job scheduler (CLQLMRS) that trains a scheduling policy to maximize local tasks rate by increasing both cache and data locality. For this purpose, we use a flavor of reinforcement learning, Q-learning, which is model-free and appropriate for this complex environment. Not only is no training dataset required, but it can also learn by interacting with the environment. In this method, we use metadata that exists in the NameNode to find the location of data and cached data, meaning extra computation is not necessary, leading to reduced overhead.

Related work

There are several surveys [3–7] of existing job schedulers that discuss their features, advantages, and limitations. They have classified job schedulers based on different aspects: strategy (static/dynamic), environment (homogenous/heterogeneous), time (deadline/delay), etc. Also, there are various job schedulers that consider data locality. These job schedulers can be

classified as to whether they consider data locality at the Map task level, data locality at the Reduce task level, or data locality at the job level (both Map and Reduce tasks) [8]. For instance, Hybrid scheduling MapReduce priority (HybSMRP) [9] was presented by Ghandomi et al. in 2019 and is a hybrid scheduler that combines dynamic job priority and data localization. It determines job priority based on three parameters: running time, job size, and waiting time. HybSMRP uses a localization marker for each node to give a fair chance to be assigned a local task. After two unsuccessful attempts to obtain a local task, a node obtains a non-local task. Increasing the data locality rate for map tasks, decreasing completion time, and avoiding wastage of resources are merits of this approach. However, it does not consider some environmental features for job priority and it assumes only one queue for jobs.

Zhang et al. [10] developed a job scheduler that utilizes a weighted bipartite graph maximum matching algorithm for improving Data Locality (DL) and Cache Locality (CL) for Map tasks. Map tasks and resources are the vertices of this graph, and the weights of the edges are determined according to the relationship between the tasks, resources, and the data locality level. This strategy sorts Map tasks by considering their priorities and then builds a selection matrix based on the location of task input data. Finally, it applies a maximum matching algorithm to select a suitable node for processing tasks. In this method, the data locality rate improves as the number of Map tasks increases, however, the focus is only on Map tasks, and the overall workload of the cluster is not considered.

In [11], Locality Aware Request Distribution (LARD) was proposed to schedule tasks by considering disk buffer caches in a distributed environment. LARD predicts cached data location based on the node where the previous request has been processed. This mechanism is suitable for small files and read-only workloads but presents poor performance for large datasets that yield frequently changing cached data. Moreover, this scheduler depends on cache prediction such that an incorrect prediction leads to increased disk access time. This last issue is the main weak point of this strategy.

Adaptive Cache Local Scheduling Algorithm (ACL) [12] was designed in 2017 and uses a cache affinity aware replacement algorithm for evicting data blocks from the in-memory cache based on the cache affinity of applications. This algorithm calculates the number of times that a job scheduler skips a task (C times) in order to obtain cache locality, this number is correlated with the current percentage of cached input data of the job. Similar to *Delay* [13], to prevent starvation, if the job scheduler skips a task D times the task is launched on a node that contains input data and has a free slot. However, it does not consider the performance impact of co-running applications in a workload. An additional disadvantage of this method is related to induced latency for scheduling tasks and launching them on cache local and data local nodes. Moreover, this strategy needs to tune the parameters C and D.

In 2017, *Cache Aware Task Scheduling (CATS)* [14] was introduced to exploit the operating system's buffer cache and take into account cached data in scheduling decisions. This technique includes two phases: a buffer cache probe, and a task scheduler. First of all, it requires cached data information from each worker node and then updates cache states based on the obtained information. Second, the job scheduler searches for a task where the greatest amount of target data is cached and then assigns it to a node that contains these cached data. The limitation of the CATS method is that it does not take into account the locality of data such that if it cannot launch cache-local tasks it uses disk scheduling, resulting in additional overhead.

The *Cache-Affinity and Virtualization Aware (CAVA)* [15] algorithm was developed by Wang et al. in 2018. The main idea of this technique is to consider a run-time priority for the MapReduce applications based on their cache affinities. *CAVA* uses memory locality-aware scheduling that considers different levels of memory locality in virtual machines: Virtual Machine Memory Locality (VMML), Physical Machine Memory Locality (PMML), Rack Machine Memory Locality (RMML), and Virtual Machine Disk Locality (VMDL). This technique reduces execution time and is suitable for virtualized clusters, however, it focuses only on Map tasks.

In Table 1, we compare these scheduling strategies in terms of their technique and locality level and summarize their advantages and disadvantages.

The main challenge of existing methods is that cached data management generates overhead. For solving this problem, we propose a solution based on reinforcement learning for improving data and cache locality to balance the load in the cluster and avoid overhead. We note that reinforcement learning has been used for various scheduling problems (resource utilization, performance, and job execution time) in different environments including cloud computing, distributed environments, and HPC. In the following, we mention some recent research in this field.

In [16] , Naik et al. designed a job scheduler that uses a type of reinforcement learning to reduce job completion time in a heterogeneous environment. For this purpose,

Table 1 Job scheduler comparisons

Scheduling method	Locality level	Technique	Advantages	Disadvantages
HybSMRP	DL (Map tasks)	Localization marker, Job priority	Avoids wastage of resources	Does not consider some environmental features for job priority
LARD	CL	Predict cached data location	It is suitable for small files	Poor performance for large dataset
Improved CL, and DL for map tasks	DL, CL (Map tasks)	Weighted bipartite graph, Maximum matching algorithm	Improves data locality rate for Map tasks	Only considers Map tasks
ACL	DL, CL	Delay tasks to launch them on local nodes	Increase tasks locality rate	Latency to schedule tasks
CATS	CL	Buffer cache probe and find tasks with the greatest amount of cached data	Increases cache locality rate	Only considers cache locality
CAVA	VMM, PMML, RMML, VMDL	Memory locality-aware and application's cache affinity	It is suitable for virtualized clusters	Only focuses on Map tasks

they apply the SARSA algorithm to identify straggler tasks and assign them to fast nodes.

In [17], Rashmi et al. proposed a Q-learning-based job scheduler to optimize resource utilization in a cloud computing workflow by considering parameters such as VMs consumed, bandwidth at the data center, and power consumption. This job scheduler consists of two steps: 1) Job preprocessing calculates the earliest start time and the deadline for each task in the workflow. 2) Scheduling using Q-learning minimizes the cost of VM creation, cost of data transfer, energy increment cost, etc.

In [18] Orhean et al. presented a job scheduler based on the SARSA algorithm for a distributed system to minimize execution time. This method considers node heterogeneity and uses a DAG to model task dependencies. In this method, a model called MBox considers the underlying heterogeneity to allocate appropriate resources to tasks.

A Q-learning-based framework for energy-efficient cloud computing (QEEC) [19] has been proposed for energy-efficient task scheduling in a cloud environment. This two-phase method combines a centralized task dispatcher and a Q-learning-based job scheduler. In the first phase, an M/M/S queuing model is used to assign requested tasks to the appropriate server and to provide a priority list of tasks based on task laxity and task lifetime. Minimizing task response time and maximizing CPU utilization is the goal of the Q-learning-based job scheduler. For this purpose, task deadlines and waiting times are considered in the reward function design.

In [20] Asghari et al. applied the SARSA algorithm to task scheduling, load balancing, and resource provisioning of workloads in a cloud environment. Their strategy includes two independent agents such that the first agent schedules tasks in a workload by using a DAG task model. The second agent uses the results of the first agent and assigns tasks to the appropriate resources considering the underlying heterogeneity. Finally, a genetic algorithm is utilized to combine the two agents.

Reinforcement learning

In this section, we introduce some useful reinforcement learning concepts that will be used in our proposed algorithm.

Reinforcement learning and Markov Decision Process (MDP) model

Reinforcement learning (RL) [21, 22] is a machine learning technique where an agent learns in an interactive environment by receiving rewards as feedback from its actions. In other words, training consists of a series of episodes with the goal of learning the optimal policy. During each episode, the agent starts at a random state and executes actions, receiving an appropriate reward as the result of each action. Maximizing the cumulative reward is the ultimate goal of this approach, yielding a general approach to automated decision-making.

The RL algorithm is based on a Markov Decision Process (MDP) [22, 23] model. An MDP is denoted by the tuple *(S, A, P, R)*, where each of the elements is as follows:

- *State (S)*: The set of possible system states.
- *Action (A):* The set of possible actions that the agent is allowed to perform.
- *Transition function (P):* A function that determines the probability of transitioning between states as a result of the selected action. It is defined as $P: S \times S \times A \rightarrow [0,1)$ where the values correspond to

the conditional probabilities $P(S_{t+1} | S = S_t, A = A_t)$, S_{t+1} is the next state, S_t is the current state, and A_t is the selected action.

- *Reward (R):* The reward is the feedback from the environment as the result of a selected action. Formally, $R: S \times A \rightarrow R$.

Q-learning and the SARSA algorithm

Q-learning [24, 25] is a value-based reinforcement learning approach that iteratively updates a Q-value for each (state, action) pair. The Q-value is updated based on the reward received for the selected (state, action) pair which leads to selecting the action with the best Q-value for subsequent states. In this case, the agent can learn the policy and the value function simultaneously. Q-learning is model-free, and it is appropriate when the system model is too complex and we are not able to estimate the transition function.

The *State-Action-Reward-State-Action (SARSA) algorithm* [26] is a classic Q-learning algorithm that learns through the transition from one state to another by performing a single action.

SARSA updates the Q-value that depends on the current state S_t, the selected action by the agent A_t, the reward value R_{t+1} received for choosing action A_t, the next state entered after performing this action S_{t+1}, and finally the next action, A_{t+1}, that the agent will select from state S_{t+1}. To be precise, Q-learning uses the following for updating the Q-values and stores these values in a Q-table for each (state, action) pair.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

where $Q(S_t, A_t)$ is the previous Q-value of (state, action), $Q(S_{t+1}, A_{t+1})$ is the next Q-value of (state, action) and $\alpha \in (0, 1)$ is a learning rate that determines the importance of the previous rewards. The closer this factor is to 1, the greater the emphasis on more recent information. The discount factor $\gamma \in (0, 1)$ determines the importance of future rewards. The smaller this factor, the more the agent is opportunistic by valuing immediate rewards.

One of the challenges in this approach is the trade-off between exploration and exploitation. The agent will choose an action based on its past experiments and what is already known to obtain current maximum rewards (exploitation) or through choosing new actions to discover new rewards (exploration). Therefore, exploration improves knowledge for long-term benefits, but exploitation maximizes short-time benefits. One way to address this is through an ε-greedy policy that balances exploitation and exploration. In this case, the agent will select the action with the largest Q-value with probability $1\text{-}\varepsilon$, and selects (uniformly) a random action with probability ε where $\varepsilon \in (0,1)$. Algorithm 1 summarizes the different steps of the SARSA algorithm [22, 25].

1) Initialize Q (S, A) arbitrarily
2) Repeat (for each episode):
3) Initialize S_t
4) choose an action A_t based on ε-greedy policy
5) Repeat for each step of episode:
6) Take action A_t and observe next state S_{t+1} and reward R_{t+1}
7) Choose action A_{t+1} based on ε-greedy policy
8) $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
9) $S_t \leftarrow S_{t+1}$
10) $A_t \leftarrow A_{t+1}$
11) Until S is terminal

Algorithm 1 SARSA algorithm

Design of CLQLMRS

In this section, we first design CLQLMRS based on an associated Markov Decision Process (MDP) model. Second, we introduce the proposed system architecture and describe its components. Finally, we explain the CLQLMRS algorithm with an example to illustrate its operation.

Modeling cache-locality scheduling as an MDP

We suggest that the *reinforcement learning (RL)* algorithm is an appropriate approach for solving our scheduling problem because RL does not require any prior knowledge about the environment and no training data is necessary. The goal of the proposed scheduler is to find a suitable node for tasks in order to increase the data locality and cache locality rates.

We specify our *MDP* model as follows:

- *Agent*: The job scheduler is an agent that should be trained on how to assign a task to the nodes to increase data and cache locality rates.
- *Environment*: The Hadoop environment contains K machines or virtual machines as worker nodes that are clustered into two or more racks and are denoted by $W=\{W_1, W_2...W_K\}$. The submitted workload consists of M Map tasks and R Reduce tasks such that the total number of tasks is $N=M+R$. The task set is defined by $T=\{T_1, T_2...T_N\}$.
- *Action:* Assigning a task to a node is the action of the agent, so formally $A: T \rightarrow W$.
- *State*: We consider five states based on different levels of data locality that may happen as the consequence of launching a task on a worker node: Cache Locality (CL), Data Locality (DL), Cache Rack Locality (CRL), Data Rack Locality (DRL), and Nothing (NO). When a task is allocated to a worker node that has cached the required input data, the state is Cache Locality (CL). The Data Locality (DL) state occurs when the job scheduler assigns a task to a worker node that contains its input data. If a task launches on other worker nodes that are placed in the same rack as the cached DataNode or DataNode with the required data the states are denoted as Cache Rack locality (CRL), and Data Rack locality (DRL), respectively. Otherwise, the state is Nothing (NO), therefore the state values set is defined as $S=\{CL, DL, CRL, DRL, NO\}$.
- *Reward:* The objective of learning is to maximize the number of local tasks which use data locality or cache locality. Therefore, we consider greater reward as the degree of locality becomes more desirable, with the greatest reward for cache locality and the least (negative) reward for no locality. We take into account five reward values related to states as given in Table 2. We consider different weights for each data locality level that indicate the importance of each level. Moreover, the reward function rewards greater locality rates in a nonlinear fashion by giving a reward that depends on a fixed weight multiplied by the number of tasks that have achieved the given locality.

System architecture

In this section, we propose a system architecture that stems from the *Yarn* [27] architecture. Its components are *Resource Manager* and *NameNode* as master nodes and the remaining nodes as worker nodes. The system architecture is presented in Fig. 1 and the role of each component is as follows:

- *Resource Manager*: Runs on a master node and has two main components: Job scheduler and Application Manager.
- *Application Manager*: Manages resource allocation among applications in the cluster. It provides a service for restarting the Application Master container on failure.
- *CLQLMRS job scheduler*: Assigns tasks to nodes with free slots using our proposed scheduler.
- *NameNode*: Runs on the master nodes and contains block metadata that determines the location of data on DataNodes and cache metadata that contains

Table 2 Reward values for different states

States	Rewards
Cache-locality	(+1 × number of tasks with cache locality) / (total number of tasks)
Data-locality	(+0.5 × number of tasks with data locality) / (total number of tasks)
Cache rack-locality	(-0.25 × number of tasks with cache rack locality) / (total number of tasks)
Data rack-locality	(-0.5 × number of tasks with data rack locality) / (total number of tasks)
Nothing	(-1 × number of tasks with no locality) / (total number of tasks)

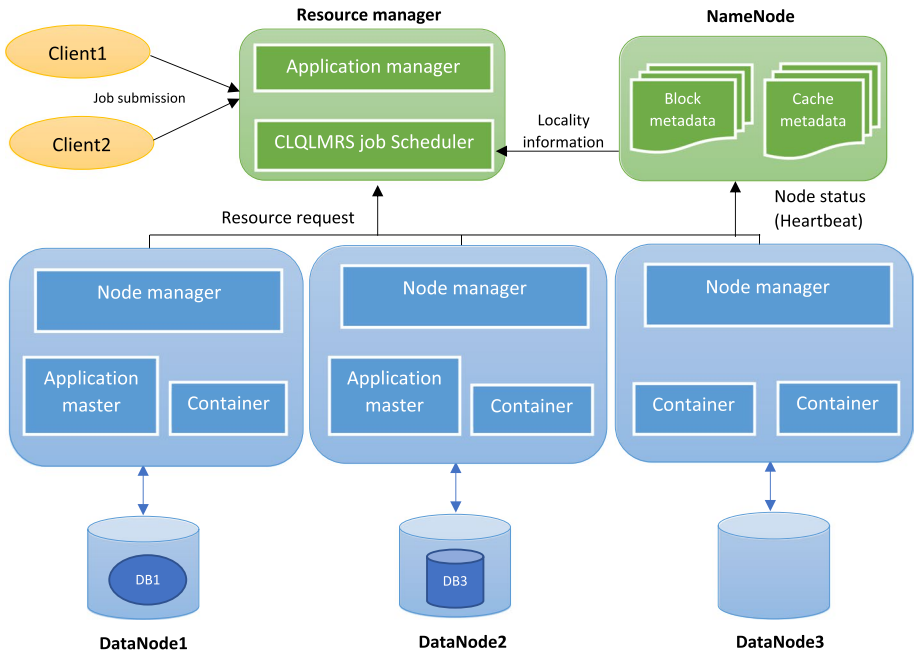

Fig. 1 The proposed system architecture

information about cached data on DataNodes. In addition, each DataNode sends a heartbeat message periodically to the NameNode for updating this metadata and providing free slot information. This information is used by the job scheduler to determine the cache locality and data locality in assigning tasks to the DataNodes.

- *Node Managers*: Run on the worker daemons and are responsible for the execution of a task on each DataNode. They register with the Resource Manager and send heartbeats with the health status of nodes. Their primary goal is to manage application containers assigned by the resource manager. They synchronize with the Resource Manager.
- *Application Master*: Manages the user job lifecycle and resource needs of individual applications. It works along with the Node Manager and monitors the execution of tasks. An application is a single job submitted to the framework. Each such application has a unique, framework-specific Application Master associated with it. It coordinates an application's execution in the cluster and also manages faults. Its task is to negotiate resources from the Resource Manager and work with the Node Manager to execute and monitor the component tasks. It is responsible for negotiating appropriate resource containers from the Resource Manager, tracking their status, and monitoring progress. Once started, it periodically sends heartbeat messages to the Resource Manager to affirm its health and to update its resource demands.
- Container: A collection of physical resources such as RAM, CPU cores, and disks on a single node. It grants rights to an application to use a specific number of resources (memory, CPU, etc.) on a specific host.
- *DataNodes*: These are worker nodes that contain data or cached data.

The CLQLMRS algorithm

In this section, we describe our proposed algorithm in detail. We first state our assumptions and then illustrate the steps of the CLQLMRS algorithm by an example.

1) Input T= {T_1, T_2,..T_N }, W={W_1, W_2,…..W_K}, Sl={Sl_{W_1} , Sl_{W_2} , ……Sl_{W_K} }
2) Initialize R_t= 0, S_t =NO, N_{CL}, N_{DL}, N_{CRL}, N_{DRL}=0 and Q-table to zero
3) Repeat in each episode while T is empty or there are no free slots
4) A_t: $T_i \rightarrow W_j$
5) $DB_x \leftarrow$Need-Data (T_i)
6) If (DB_x, W_j) is in the cache hash table then
7) $S_{t+1} \leftarrow$ CL
8) $R_{t+1} = (+1 \times N_{CL}) / N$
9) $N_{CL}=N_{CL}+1$
10) End if
11) If (DB_x, W_j) is in the data hash table then
12) $S_{t+1} \leftarrow$ DL
13) $R_{t+1} \leftarrow (+ 0.5 \times N_{DL}) / N$
14) $N_{DL}= N_{DL} +1$
15) End if
16) If Same-Rack (W_j, W_k) for each (DB_x, W_k) pair in the cache hash table then
17) $S_{t+1} \leftarrow$ CRL
18) $R_{t+1} \leftarrow (-0.25 \times N_{CRL}) / N$
19) $N_{CRL} = N_{CRL}+1$
20) End if
21) If Same-Rack (W_j, W_k) for each (DB_x, W_k) pair in the data hash table then
22) $S_{t+1} \leftarrow$ DRL
23) $R_{t+1} \leftarrow (-0.5 \times N_{DRL}) / N$
24) $N_{DRL} = N_{DRL}+1$
25) End if
26) Otherwise
27) $S_{t+1} \leftarrow$ NO
28) $R_{t+1} \leftarrow (-1 \times N_{NO}) / N$
29) $N_{NO} = N_{NO}+1$
30) End if
31) $Sl_{wj} \leftarrow Sl_{wj} -1$
32) {T}$\leftarrow$ {T}-T_i
33) If (rand () <ε) then
34) $A_{t+1} \leftarrow$ Random Task ()
35) Else
36) $A_{t+1} \leftarrow$ lookup Q-table for new state & choose action with max Q-value
37) End if
38) $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
39) $S_t \leftarrow S_{t+1}$
40) $A_t \leftarrow A_{t+1}$
41) Go to step 3

Algorithm 2 CLQLMRS Algorithm

We consider a set of virtual machines in different racks that are clustered and include one master node as a *NameNode* and multiple worker nodes denoted by $W=\{W_1, W_2,...W_K\}$. Each worker node contains a number of free slots for processing tasks denoted by $Sl=\{Sl_{W1}, Sl_{W2}, ...Sl_{WK}\}$. Input data are split into data blocks of the same size that are distributed among the worker nodes. We assume that there are two hash tables in the *NameNode* that contain cache state information (the cache hash table) and data block locations (the data hash table). For instance, the (DB_x, W_y) pair in the cache hash table indicates that data block DB_x has been cached into node W_y. A user submits a job composed of M Map tasks and R Reduce tasks, denoted by $T=\{T_1, T_2,...T_N\}$, where $N=M+R$. In this problem, the job scheduler as an agent should be trained to find the appropriate worker node that has a free CPU slot and contains or has cached the required input data, and its aim is to maximize data and cache locality.

Algorithm 2 provides the details of the CLQLMRS algorithm. In this algorithm, the initial value for the reward and the Q-table are set to zero and the initial state is NO. In the first step, the agent selects a random action that assigns task T_i to a worker node W_j $(A_t=T_i \rightarrow W_j)$ and task T_i requires a data block DB_x for processing. Next, it investigates the different statuses of W_j based on various data locality levels and calculates rewards associated with each state according to Table 2:

1. CL state: The cache locality state arises if data block DB_x is cached on node W_j or in other words, the pair (DB_x, W_j) exists in the cache hash table. The reward for this state is calculated by $R_{t+1}=(+1\times N_{CL}) / N$, where N_{CL} is the number of tasks with cache locality.
2. DL state: The data locality state will occur if W_j contains data block DB_x, meaning that the pair (DB_x, W_j) is in the data hash table. In this case, we use $R_{t+1}=(+0.5\times N_{DL}) / N$ for calculating the reward, where N_{DL} is the number of tasks with data locality.
3. CRL state: Cache rack locality will happen if W_j is located in the same rack with the nodes that cached DB_x. Therefore, the reward is calculated by $R_{t+1}=(-0.25\times N_{CRL}) / N$, where N_{CRL} is the number of tasks with rack locality.
4. DRL state: Data rack locality will happen if W_j is located in the same rack with the nodes that include data block DB_x. Therefore, the reward is calculated by $R_{t+1}=(-0.5\times N_{DRL}) / N$, where N_{DRL} is the number of tasks with rack locality.
5. NO state: If none of the above states happens then the reward is $R_{t+1}=(-1\times N_{NO})/N$, where N_{NO} is the number of tasks without any locality.

After determining the next state and calculating the reward, the number of free slots in the worker node W_j is reduced by one. Then the agent selects the next action based on the ε-greedy policy. As mentioned in the previous section, a random number is generated, if its value is less than ε then a random action will be selected, otherwise, the Q-table is used to choose an action with the best Q-value. Initially, if the ε value is large, then its value should decrease to allow for greater initial exploration, followed by increased exploitation. Next, the SARSA formula is used for estimating the corresponding Q-value and updating the Q-table for the (S_t, A_t) pair. The state and action are then updated. The algorithm repeats the above steps until the end of the episode when there are no tasks remaining for scheduling or there are no free slots on the worker nodes.

In order to clarify how the proposed algorithm calculates the next state and the next reward in each step of one episode, we provide an example. Figure 2

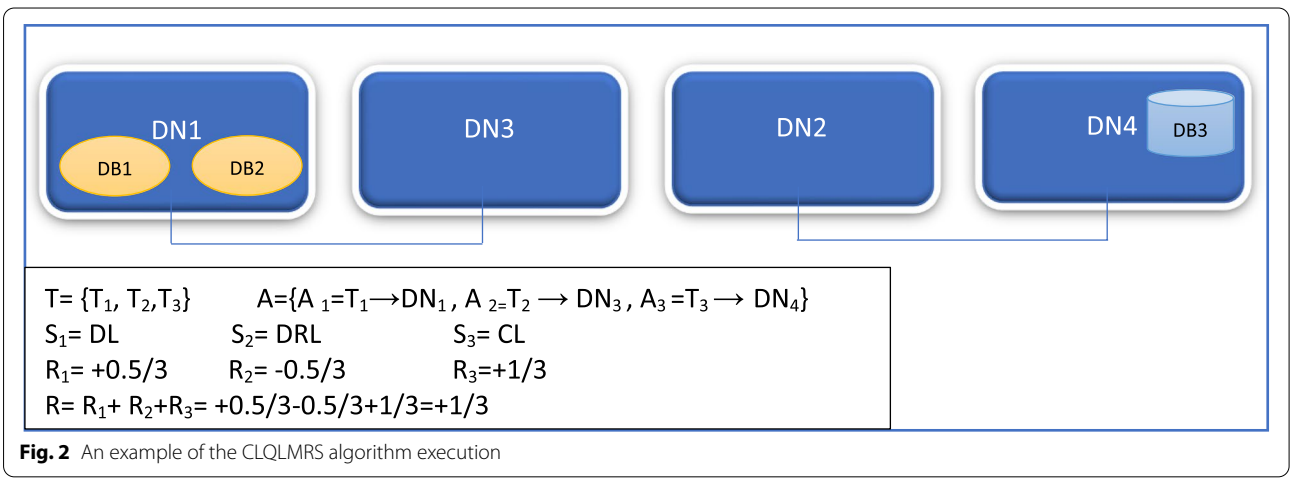

Fig. 2 An example of the CLQLMRS algorithm execution

illustrates a cluster of four nodes situated in two racks, and data blocks DB_1, and DB_2 are located in the DataNode DN_1, and data block DB_3 is cached on DataNode DN_4. We assume that $T=\{T_1, T_2, T_3\}$ is the set of tasks to be executed, and the required data for these tasks are $D=\{D_1, D_2, D_3\}$ respectively. We assume the randomly selected action A is the set of actions $A=\{A_1=T_1\rightarrow DN_1, A_2=T_2\rightarrow DN_3, A_3=T_3\rightarrow DN_4\}$. The notations S_1 and R_1 denote the next state and the next reward after performing action A_1. The algorithm determines the next state based on the selection action and locality information, therefore after performing action A_1 that assigns task T_1 to DN_1 containing the required data, it moves to the data locality state, and the reward is calculated based on Table 2 and is equal to $+0.5/3$. After allocating task T_2 to DataNode DN_3 by action A_2, the next state is Data Rack locality because DataNode DN_3 is located in the same rack as the DataNode that contains the required data and the reward is equal to *-0.5/3*. Also, action A_3 leads to the cache locality state for the next state as it assigns task T_3 to DataNode DN_4 that has the required data cached and the reward equals $+1/3$. Therefore, the cumulative reward R at the end of this episode is the sum of these three rewards after there are no tasks for scheduling or all the nodes' slots are busy.

CLQLMRS implementation

In this section, we describe the implementation of CLQLMRS. The implementation has two pieces: modeling the environment and implementing the proposed algorithm. For this purpose, we use the Gym library [28] in Python.

Environment implementation

Figure 3 demonstrates the structure of the Hadoop cluster that we are considering, consisting of eight nodes located in two racks such that odd-numbered nodes are in rack1 and even-numbered nodes are in rack2. These two racks are connected via an external network switch. For simplicity, we assume that each node has one free slot to process tasks. Also, data are cached and distributed among DataNodes in a random manner to determine the initial environment. In other words, we consider a static environment such that for each training episode the data distribution and cached data are fixed. We could test a dynamic environment for future work.

For environment implementation, we use three procedures: Init, Step, and Reset. The input parameters of the Init procedure are the number of tasks, maximum number of episodes, cache capacity, and memory capacity based on the number of data blocks. By using the Init procedure, the environment is initialized including determining the state space and action space, the initial value of the state, and reward which are equal to NO and zero, respectively. Next, the distribution of data and the location of cached data are declared by using a random distribution of data blocks among DataNodes which remain fixed during all training episodes. The second function, the Step function, simulates each step of training and checks whether the episode is finished or not by using the number of scheduled tasks or free slots of the worker node. The Step function determines the feedback from the environment after choosing any action in each state. In other words, this function takes an action as a parameter then it calculates the next state and step reward based on this action and locality information of demand data in both block and cache metadata. At the end of each episode, the Reset procedure first increments the number of episodes and checks whether the maximum number of episodes has been reached. This procedure is used to reset the environment which sets the step reward to zero, all tasks to unscheduled, and frees all slots.

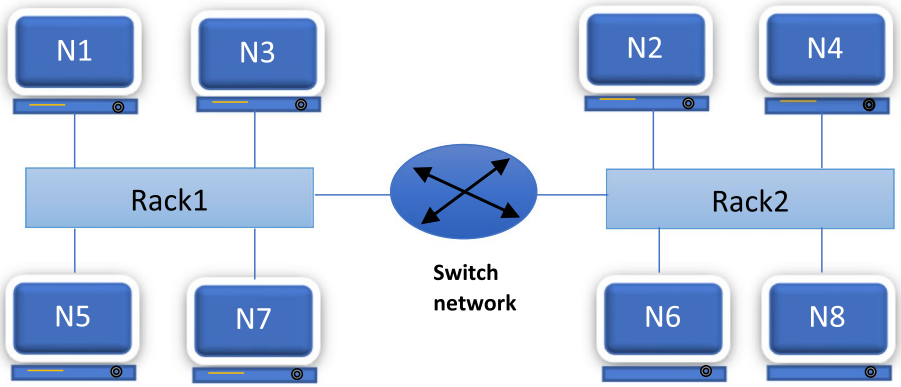

Fig. 3 The structure of the Hadoop cluster

Implementing the proposed algorithm

Before implementation, we consider the CLQLMRS workflow that is presented in Fig. 4 to determine the relationship between the agent and environment in the SARSA algorithm. The CLQLMRS job scheduler agent contains a task pool waiting for scheduling and a list of nodes with free slots. Assigning the tasks to a node with demand data is the responsibility of the agent which provides a mapping between these two lists. Next, the Hadoop environment receives feedback based on each action A_t and determines the next state based on the locality information in the block metadata and cache metadata. The reward for the step is calculated based on the next state and updated Q-value in the Q-table based on the SARSA formula. Finally, we use the ε-policy to select the next action.

In our algorithm implementation, we tune several parameters. The ε-value, learning rate, and discount factor are set to 1, 0.1, and 0.95 respectively. Next, we initialize the environment using the Init procedure and set the Q-table to be all zeroes. Each episode is terminated when one of the following conditions is met: either there is no task to be scheduled or all worker node slots are busy. The episodes are repeated until the maximum number of episodes is reached. In each episode, the following steps are performed:

- First, the environment is reset by using the Reset procedure.
- Repeat the following steps until the episode is finished:
- In each step, one action is selected that assigns an unscheduled task to a worker node with free slots by using the ε-greedy policy. As mentioned in the previous section, a random number is generated, if its value is less than the ε-value then a random action will be selected otherwise, the Q-table is used to choose an action with the best Q-value. Then, the next state and step reward are calculated by calling the Step function. Finally, the Q-table is updated by using the SARSA formula and based on the current state, the current action, the step reward, the next state, and the next action.
- At the end of each episode, the cumulative reward, data locality rate, and cache locality rate are determined.
- After every 10 episodes, the value of ε is decreased by a factor of 0.999 because we want to increase the use of exploitation as the number of episodes increases.

CLQLMRS evaluation

In this section, we explain the experimental environment including software and hardware configurations and set some Hadoop configuration parameters. Our evaluation is divided into two sections: investigating the impact of various factors on cache locality and data locality improvement and CLQLMRS performance evaluation and its effect on Hadoop performance. We first evaluate the impact of different factors like cache size, data block size, and the number of Map tasks on cache and data locality improvement. Finally, we perform experiments

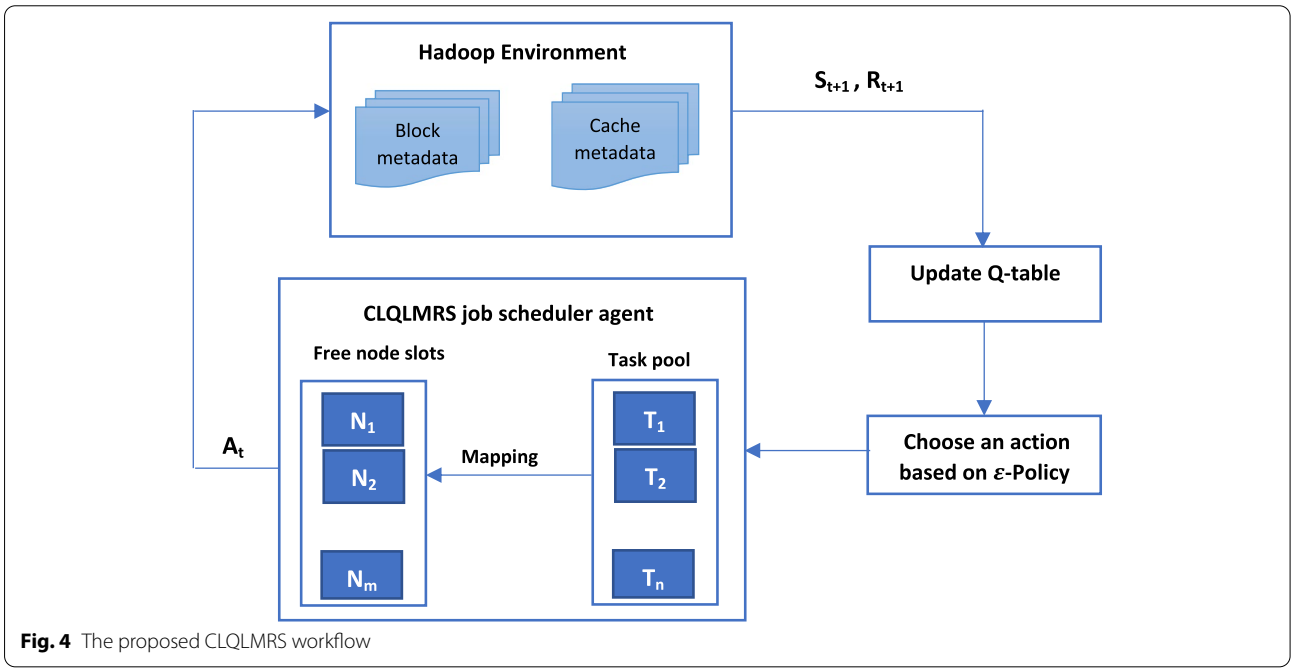

Fig. 4 The proposed CLQLMRS workflow

to study the impact of the novel job scheduler on overall Hadoop performance.

Experimental setup

For our experiments, we use a cluster consisting of a single NameNode and six DataNodes located in two racks such that odd nodes are in rack1 and even nodes are in rack2.

- *Hardware configuration*: The nodes are connected via a 10 Gigabit Ethernet switch. The experimental environment is a heterogeneous environment with different memory sizes at the nodes: NameNode has 16 GB RAM, DataNode 1 and DataNode 4 have 4 GB RAM, DataNode 2 and DataNode 5 have 6 GB RAM, DataNode 3 and DataNode 6 have 8 GB RAM. The CPU for all nodes is Intel core i5-4590, a 3.30 GHz CPU.
- *Software configuration*: We use the Ubuntu14.04 operating system and JDK 1.8, Hadoop version 2.7 (which employs in-memory caching), and Intel HiBench [29, 30] version 7.1.
- *Hadoop configuration parameters:* The block size of files in HDFS is chosen to be one of two values, 64 MB or 128 MB, the number of cache replicas is set to one, and data replication is set to 3. Table 3 presents the Hadoop configuration parameters with their values. The remaining Hadoop configuration parameters are set to default values.
- *MapReduce applications*: As we mentioned earlier, we use Intel HiBench as a Hadoop benchmark suite and carry out the experiments by using two groups of benchmarks: 1) Micro benchmarks that contain WordCount, Sort, and TeraSort applications. 2) Hive benchmarks as a big data query and analysis application, for instance, Join and Aggregation. WordCount is a CPU-intensive application that counts occurrences of each word in a text file. Sort is a typical I/O-bound application with moderate CPU usage and sorts input data. TeraSort is an I/O-oriented program such that it needs moderate disk I/O during the Map and Shuffle phases and heavy disk I/O in the Reduce phase. Both Aggregation and Join are supported by Hive and used for operation in the query. Join is a multiple-stage application where the results of each step are used as input for the next step.
- *Input data:* For carrying out experiments, we have used the default data sizes from the HiBench suite. Sort and WordCount have 60 GB and TeraSort has 1 TB as input data size. The input data WordCount, Sort, and TeraSort are generated by using the RandomTextWriter, RandomWriter, and TeraGen programs respectively, all contained in the Hadoop distribution.
- *Job schedulers:* We compare FIFO as the Hadoop native job scheduler, Delay as a job scheduler that considers only data locality, and adaptive cache local scheduling (ACL) as a scheduler that considers both levels of the locality (data locality and cache locality), along with our proposed job scheduler.

Table 3 Hadoop parameters

Hadoop property name	Hadoop property value
Dfs.replication	3
Dfs.blocksize	64 M or 128 M
MapReduce.map.memory.mb	1024
MapReduce.reduce.memory.mb	2048
mapreduce.map.output.compress	False
MapReduce.reduce.speculative	False
MapReduce.map.speculative	False
Mapred.map.tasks.speculative.execution	False
Mapred.reduce.tasks.speculative.execution	False

Metrics

In our experiments, we consider two key performance metrics:

- *Job execution time:* The fundamental standard for performance is to minimize the execution time of a job.
- *Local tasks rate:* Measures the ratio of the number of tasks run in the node where their required data are resident or cached to the total number of tasks.

Impact of various factors on data locality

In this section, we investigate the effect of different factors on cache and data locality improvement and compare our novel job scheduler with Hadoop's native job scheduler and Delay. These experiments examine:

- Cache locality rate as a function of cache size
- Local tasks rate as a function of data block size
- Local tasks rate as a function of the number of tasks

Impact of cache size on cache locality

In this experiment, we investigate the impact of different cache sizes on the cache locality rate. For this purpose, we first define the cache locality rate as the ratio of the number of tasks that utilize cached data to the total

number of tasks. Next, we consider different cache sizes (2, 4, 6, and 8 GB) and execute the WordCount application with the native Hadoop, adaptive cache local scheduling (ACL), and CLQLMRS job schedulers. Also, we selected two values, 2 and 5, for the C parameter in ACL to investigate its effect on cache locality.

Figure 5 presents these experimental results. It is obvious that the cache locality rate increases with the cache size in all cases. This is consistent with intuition, as we can cache more data by increasing the cache size, and more tasks have a chance to utilize cached data. Therefore, we can conclude that the increasing cache size has a positive effect on the cache locality rate. For the native Hadoop scheduler, there appear to be diminishing returns, for example, the cache locality rate does not increase significantly by increasing the cache size from 6 to 8 GB. Also, by doubling the cache size from 4 to 8 GB the cache locality rate rises by 3%, 17%, 25%, and 24% for the native Hadoop scheduler, ACL(C=2), ACL(C=5), and CLQLMRS, respectively. When we compare our proposed scheduler with ACL, we conclude that ACL has similar performance to CLQLMRS when the parameter C has a large value. Also, we observe that the cache locality rate in ACL (C=5) is more than in ACL (C=2). Therefore, the small C value leads to a low cache hit ratio even though we get low scheduling overhead in this case. We can expect a high cache hit ratio with a large C value however, it suffers from a long waiting time to schedule and high overhead. This observation suggests that the CLQLMRS scheduler is better able to take advantage of increased cache size.

Impact of data block size on local tasks

This experiment assesses the effect of data block size on local tasks rate. In this experiment, we change the data block size from 64 to 256 MB in the Hadoop settings, then run the WordCount application using each of the four job schedulers. For the ACL job scheduler, we set the C value to 3 in the remainder of the experiments, as this choice appears to have a reasonable trade-off between scheduling overhead and cache hit ratio.

Figure 6 demonstrates that we have the highest rate of local tasks with the largest data block size. This follows intuition, as including more input data means that more tasks can utilize local data. Therefore, the size of the data block has a direct impact on the local tasks rate. Moreover, the ACL and the proposed job scheduler take into account data locality and cache locality in assigning tasks to the nodes, as a result, they have higher local task rates than FIFO and Delay. We observe that by increasing data block size the performance of CLQLMRS and ACL are similar because the larger amount of data can be located in a data block that can be cached. Since the number of skips is set to be proportional to the percentage of cached input data for the job, therefore the number of tasks skipped is reduced to achieve local tasks. A key observation is that for CLQLMRS, the local tasks rate is already quite high at a small block size, and has comparable performance to Delay and ACL when the block size is half as large.

Impact of number of Map tasks on local tasks

This experiment is carried out to demonstrate the influence of the number of Map tasks on the number of local tasks. For this purpose, we use the WordCount application with the number of Map tasks ranging from 10 to 50, using the different job schedulers.

Figure 7 illustrates that the number of Map tasks has a positive impact on increasing the local tasks

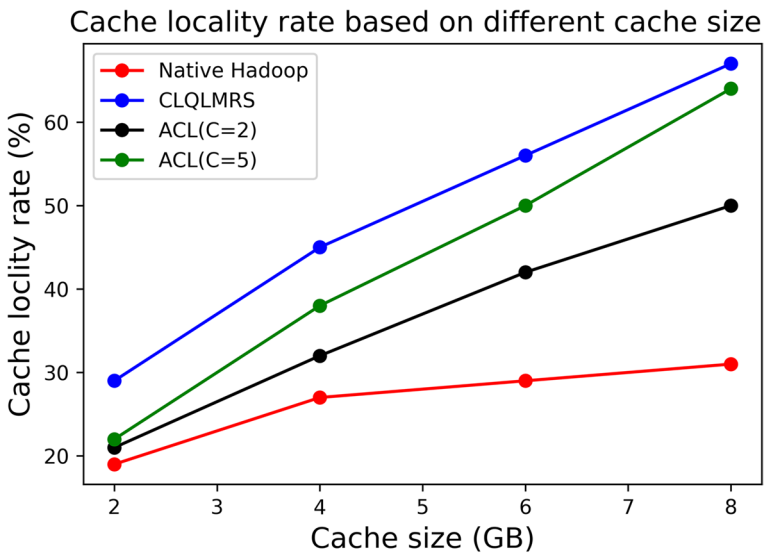

Fig. 5 Cache locality rate based on cache size

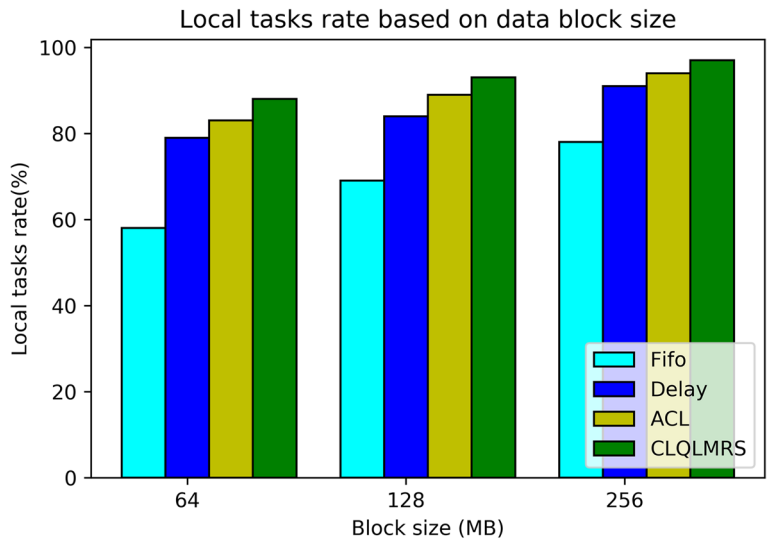

Fig. 6 Local tasks rate based on data block size

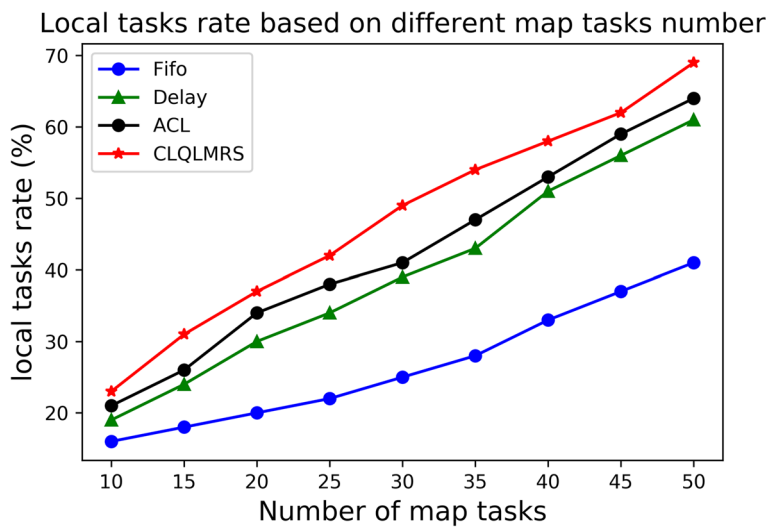

Fig. 7 Local tasks rate based on number of Map tasks

Table 4 Locality levels achieved by each scheduler based on different job size

Job size	FIFO			Delay			ACL			CLQLMRS		
	CL	DL	RL	CL	DL	RL	CL	DL	RL	CL	DL	RL
3 Maps	0%	2%	50%	12%	75%	96%	21%	81%	99%	28%	85%	99%
10 Maps	7%	37%	95%	19%	99%	100%	29%	99%	100%	36%	99%	100%
50 Maps	12%	61%	98%	24%	95%	99%	38%	98%	100%	43%	99%	100%
100 Maps	25%	78%	99%	35%	91%	99%	46%	94%	100%	55%	98%	100%

rate because by increasing the number of tasks the chance that a task is local also increases. The CLQLMRS and ACL schedulers present the greatest increase due to the fact that they consider both data locality and cache locality in their scheduling decisions.

While Fig. 7 presents aggregate locality results, Table 4 presents different locality levels (cache, data, and rack

locality) achieved by the four job schedulers for different job sizes based on the number of Map tasks. It can be observed that the maximum cache locality reaches 25%, 35%, 46%, and 55% in FIFO, Delay, ACL, and CLQLMRS, respectively when we have 100 Map tasks. Also, the Delay and ACL scheduler yield 99% data locality and 100% rack locality for the 10 Map task case. Then, as the number of Map tasks is increased to 100, the locality rate decreases. On the other hand, data and rack locality rates remain steady in CLQLMRS. We can conclude that the locality rate is more stable for large job sizes in our proposed job scheduler.

Performance evaluation

In this section, we evaluate the performance of CLQLMRS and its effect on Hadoop based on two metrics: local tasks rate and job execution time. For this purpose, we run a combination of the Micro benchmark and the Hive benchmark in Intel HiBench that is composed of I/O-bound jobs, CPU-bound jobs, and data-oriented applications to compare CLQLMRS with ACL, Delay, and FIFO.

Performance evaluation based on local tasks rate

Figure 8 demonstrates the proportion of local tasks in different applications by using the four job schedulers. Join and Aggregation applications have the least and the most local tasks rate among all applications. Due to fact that the Join application has independent multiple stages and cannot reuse data it has a low cache hit rate. However, Aggregation is a high cache affinity application which means that it can more utilize the benefit of cached data. The proposed scheduler improves the locality rate by 23%, 22%, 22%, and 20% in Aggregation, Sort, Join, and Terasort applications as compared to FIFO. Therefore, the novel job scheduler has the best improvement in locality rate for I/O-Bound jobs and data-oriented applications which results in reduced data transmission through the network and improved performance.

When we compare job schedulers, it can be observed that FIFO has the worst tasks locality rate. ACL and CLQLMRS have the highest local tasks rate for all applications because both of them consider cache locality and data locality in their scheduling strategy. Also, Delay, ACL, and CLQLMRS improve the local tasks rate to 13.8%, 17.6%, and 20.8% on average respectively when compared with FIFO.

Performance evaluation based on job execution time

The average job execution time of various applications is presented in Fig. 9. When we compare the four job schedulers, we see that job execution time decreases in both ACL and CLQLMRS (as compared to Delay and FIFO). The high rate of local tasks has a positive impact on job execution time by decreasing the amount of data transmitted through the network. It can be observed that ACL and CLQLMRS improve average job execution time by 19.4%, and 26.54% against FIFO. Also, CLQLMRS presents an 8.98% improvement in job execution time related to ACL because ACL spends some waiting time to launch local tasks while our proposed strategy eliminates any waiting time and trains the scheduler to find the best locality level.

Also, Table 5 shows the improvement in these applications' execution time related to the Hadoop native scheduler. The proposed method finishes the Sort and TeraSort applications 20% and 18.45% faster than FIFO,

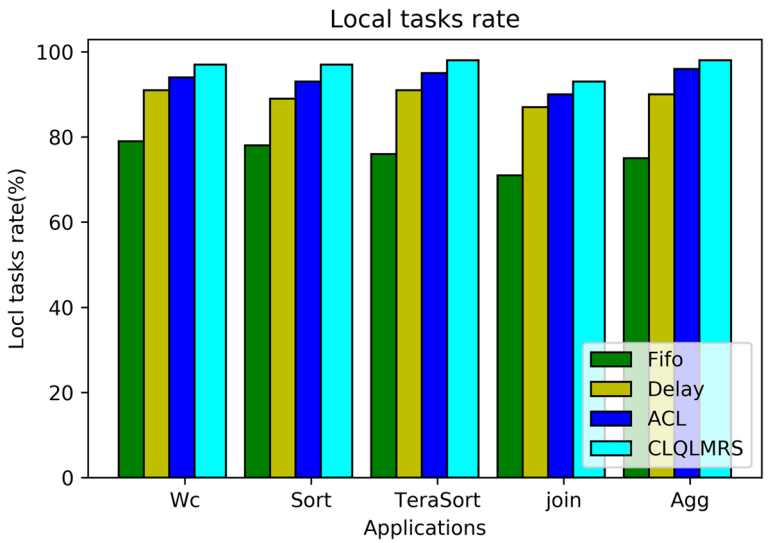

Fig. 8 Local tasks rate for different applications

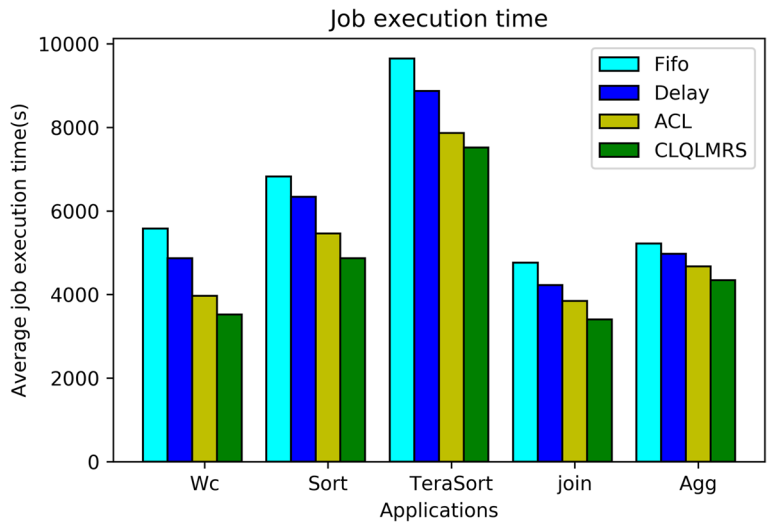

Fig. 9 Average job execution time for different applications

Table 5 The percentage of improvement in different applications' execution time

Application	Delay	ACL	CLQLMRS
WordCount	12.72%	28.82%	36.88%
Sort	7%	20%	28.65%
TeraSort	8%	18.45%	22%
Join	11.29%	19.26%	28.48%
Aggregation	4.69%	10.48%	16.72%

respectively, and it reduces the execution time for these applications by 10.8% and 4.4%, respectively, as compared to ACL. Therefore, CLQLMRS has the greatest impact in decreasing execution time for I/O-Bound jobs and data-oriented applications like Sort, TeraSort, and Aggregation. We can conclude that it is best suited for these types of applications.

Conclusion and future work

In this paper, we proposed a state-of-the-art job scheduler (CLQLMRS) that employs reinforcement learning to train a scheduling policy that considers both cache locality and data locality. The advantage of this method is that it does not need any prior knowledge because it can learn by interacting with the Hadoop environment. Experimental results show that some factors like cache size, data block size, and the number of Map tasks are directly related to the rate of local tasks. Our proposed algorithm improves Hadoop performance by 26.54%, 19.63%, and 8.98% against FIFO, Delay, and ACL, respectively. Because CLQLMRS improves the locality rate in both cache and data it leads to decreased job execution times and improved Hadoop performance. Moreover, we conclude that CLQLMRS is best suited for I/O-Bound jobs and data-oriented applications. However, this mechanism needs to train the scheduling policy, which is one of this method's limitations. As a consequence, if the environmental changes are quick enough, retraining in a timely manner may not be possible.

In future work, we plan to study the scalability of the proposed algorithm with respect to both the size of the cluster and the size of jobs. As mentioned earlier, the size of the Q-table is determined by state and action spaces. In a large cluster, we have more worker nodes with more slots, and the number of tasks grows for large jobs. In this case, the dimension of the Q-table is increased, and updating it requires more time. We suggest using multiple tables with different hash functions to tackle this issue. The Q-values for unknown (state, action) pairs would be approximated by Q-values of several nearby (state, actions) pairs. The values of nearby points can be hashed into the same bucket as values of distant points that have little effect on a given target pair.

Authors' contributions

Mrs. Rana Ghazali contributed to the main idea and wrote the main manuscript text. Dr. Ali Movaghar introduced the core idea. Dr. Sahar Adabi supervised the research and reviewed the paper. Dr. Douglas Down reviewed the paper and supervised the evaluation process. Dr. Ali Rezaee conceived and designed the analysis. All authors read and approved the final manuscript.

Funding

Not applicable. No funding in any form is received for this manuscript.

Availability of data and materials
The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Consent for publication
Not applicable. This manuscript does not have any individual person data.

Competing interests
There are no financial or non-financial competing interests.

Author details
[1]Department of Computer Engineering, North Tehran Branch, Islamic Azad University, Tehran, Iran. [2]Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran. [3]Department of Computing and Software, McMaster University, 1280 Main St W, Hamilton, ON, Canada. [4]Department of Computer Engineering, Sharif University of Technology, Azadi Ave, Tehran, Iran.

Received: 13 May 2022 Accepted: 31 August 2022
Published online: 19 September 2022

References

1. "Apache Hadoop" http://hadoop.apache.org/
2. "Centralized Cache Management" https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html
3. Usama M, Liu M, Chen M (2017) Job schedulers for big data processing in Hadoop environment: testing real-life schedulers using benchmark programs. Digit Commun Netw 3:260–273
4. Abdallat AA, Alahmad AI, Amimi DAA, AlWidian JA (2019) Hadoop MapReduce job scheduling algorithms survey and use cases. Mod Appl Sci 13:38
5. Kalia K, Gupta N (2021) Analysis of Hadoop MapReduce scheduling in a heterogeneous environment. Ain Shams Eng J 12:1101–1110
6. Kang Y, Pan L, Liu S (2022) Job scheduling for big data analytical applications in clouds: A taxonomy study. Futur Gener Comput Syst 135:129–145
7. Bawankule KL, Dewang RK, Singh AK (2022) A classification framework for straggler mitigation and management in a heterogeneous Hadoop cluster: A state-of-art survey. J King Saud Univ Comput Inf Sci. https://doi.org/10.1016/j.jksuci.2022.02.021
8. Ghazali R, Adabi S, Down DG, Movaghar A (2021) A classification of Hadoop job schedulers based on performance optimization approaches. Clust Comput 24:3381–3403
9. Gandomi A, Reshadi M, Movaghar A, Khademzadeh A (2019) HybSMRP: a hybrid scheduling algorithm in Hadoop MapReduce framework. J Big Data. https://doi.org/10.1186/s40537-019-0253-9
10. Zhang P, Li C, Zhao Y (2016) An improved task scheduling algorithm based on cache locality and data locality in Hadoop. Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings 0. p 244–249
11. Pai VS, Aron M, Banga G, Svendsen M, Druschel P, Zwaenepoel W, Nahum E (1998) Locality-aware request distribution in cluster-based network servers. ACM Sigplan Notice 33:205–216. ACM
12. Floratou A et al (2016) Adaptive caching in Big SQL using the HDFS cache. Proceedings of the 7th ACM Symposium on Cloud Computing, SoCC. p 321–333. https://doi.org/10.1145/2987550.2987553
13. Zaharia M et al (2010) Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European conference on Computer systems. p 265. https://doi.org/10.1145/1755913.1755940
14. Lim B, Kim JW, Chung YD (2017) CATS: cache-aware task scheduling for Hadoop-based systems. Clust Comput 20:3691–3705
15. Hwang E, Kim H, Nam B, Choi YR (2018) CAVA: Exploring memory locality for big data analytics in virtualized clusters. Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID. p 21–30. https://doi.org/10.1109/CCGRID.2018.00017
16. Sastry VN, Negi A, Naik NS (2018) Improving straggler task performance in a heterogeneous MapReduce framework using reinforcement learning. Int J Big Data Intell 5:201
17. Rashmi S, Basu A (2017) Q learning-based workflow scheduling in Hadoop. Int J Appl Eng Res 12:3311–3317
18. Orhean AI, Pop F, Raicu I (2018) New scheduling approach using reinforcement learning for heterogeneous distributed systems. J Parallel Distrib Comput 117:292–302
19. Ding D et al (2020) Q-learning based dynamic task scheduling for energy-efficient cloud computing. Futur Gener Comput Syst 108:361–371
20. Asghari A, Sohrabi MK, Yaghmaee F (2021) Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel SARSA reinforcement learning agents and genetic algorithm. J Supercomput 77:2800–2828
21. Sutton RS, Barto AG (1998) Reinforcement learning: an introduction. MIT Press, Cambridge
22. Naeem M, Rizvi STH, Coronato A (2020) A Gentle Introduction to Reinforcement Learning and its Application in Different Fields. IEEE Access 8:209320–209344
23. Puterman ML (2014) Markov decision processes: discrete stochastic dynamic programming. Wiley, New York
24. Watkins CJCH (1989) Learning from delayed rewards. Ph.D. Diss. King's College, Cambridge
25. Jang B, Kim M, Harerimana G, Kim JW (2019) Q-Learning Algorithms: A Comprehensive Classification and Applications. IEEE Access 7:133653–133667
26. Corazza M, Sangalli (2015) Q-learning, and SARSA: a comparison between two intelligent stochastic control approaches for financial trading. University Ca'Foscari of Venice, Dept of Economics Research Paper Series No 15
27. "Yarn" https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html
28. "Gym" https://gym.openai.com
29. Huang S, Huang J, Dai J, Xie T, Huang B (2014) The HiBench Benchmark Suite : Characterization of the MapReduce-Based Data Analysis. https://doi.org/10.1109/ICDEW.2010.5452747
30. "Hibench" https://github.com/Intel-bigdata/HiBench

Publisher's Note