Learning-Based Dynamic Memory Allocation Schemes for Apache Spark Data Processing

Danlin Jia, Li Wang, Natalia Valencia, Janki Bhimani, Bo Sheng, and Ningfang Mi

***Abstract*—Apache Spark is an in-memory analytic framework that has been adopted in the industry and research fields. Two memory managers, Static and Unified, are available in Spark to allocate memory for caching Resilient Distributed Datasets (RDDs) and executing tasks. However, we find that the static memory manager (SMM) lacks flexibility, while the unified memory manager (UMM) puts heavy pressure on the garbage collection of the JVM on which Spark resides. To address these issues, we design a learning-based bidirectional usage-bounded memory allocation scheme to support dynamic memory allocation with the consideration of both memory demands and latency introduced by garbage collection. We first develop an auto-tuning memory manager (ATuMm) that adopts an intuitive feedback-based learning solution. However, ATuMm is a slow learner that can only alter the states of Java Virtual Memory (JVM) Heap in a limited range. That is, ATuMm decides to increase or decrease the boundary between the execution and storage memory pools by a fixed portion of JVM Heap size. To overcome this shortcoming, we further develop a new reinforcement learning-based memory manager (Q-ATuMm) that uses a Q-learning intelligent agent to dynamically learn and tune the partition of JVM Heap. We implement our new memory managers in Spark 2.4.0 and evaluate them by conducting experiments in a real Spark cluster. Our experimental results show that our memory manager can reduce the total garbage collection time and thus further improve Spark applications' performance (i.e., reduced latency) compared to the existing Spark memory management solutions. By integrating our machine learning-driven memory manager into Spark, we can further obtain around 1.3x times reduction in the latency.**

***Index Terms*—JVM memory management, distributed data processing, machine learning, Apache Spark, Q-learning.**

I. Introduction

THE unprecedented proliferation of data has triggered a significant development of scalable analytics stacks in recent years. Developers and researchers strive to boost data-processing speed in hardware and software. However, processing a massive volume of data has entirely relied on the performance of computing facilities and the efforts of users and can only achieve a suboptimal performance [1]. Thus, distributed frameworks (e.g., Hadoop [2]) that share computational resources on a cluster have been proposed to handle the overwhelming data. However, it has been noticed that in Apache Hadoop, many I/O requests are generated for accessing the intermediate data, To address this issue, in-memory analytic frameworks (e.g., Apache Spark [3]) have been developed to improve data processing performance.

Apache Spark [3], one of the most successful in-memory analytic frameworks, has been going through a boom in the past few years. Specifically, Apache Spark implements an abstraction of a data structure called Resilient Distributed Datasets (RDD) [4], which can be manipulated in parallel on different executors. Each RDD is created from an input dataset or another RDD and is immutable. Based on these two features, Spark builds a lineage of an application to track each computation stage and recover from faults in a tolerant way. Furthermore, Spark stores intermediate data (i.e., RDDs) in RAM, which reduces communication overhead between Spark executors, especially for some iterative and interactive machine learning applications. In this way, Spark avoids the overhead of I/O operations and improves overall performance. Therefore, one of the most crucial factors in Spark is the management of memory resources. An effective memory management scheme can shrink an application's latency (i.e., the total execution length) and improve performance dramatically. Unfortunately, Apache Spark hides the default scheme in memory management from users, who have few opportunities to monitor and configure the memory space.

In this work, we first investigate two existing Spark memory managers: Static memory manager (***SMM***) and Unified memory manager (***UMM***). Specifically, SMM applies predefined configurations to allocate fixed memory partitions for Spark applications, which heavily relies on the user's efforts and knowledge of the application's characteristics for memory optimization. On the other hand, UMM can dynamically allocate memory based on the run-time memory demands. However, UMM introduces heavy Garbage Collection (GC) as it tends to overprovision memory for runtime objects. We further run representative data processing benchmarks to collect the latency of applications under these two memory managers. We find that the Spark performance is significantly affected by the memory partition, which may lead to either long Java garbage collection (GC) or long delay in intermediate data access. Based on the analysis of the defects of the existing memory managers, we design a

Manuscript received 10 April 2023; revised 31 August 2023; accepted 28 September 2023. This work was supported by the National Science Foundation (NSF) under Grants CNS-2008324, CNS-2323100, CNS-1452751, and CNS-2008072. Recommended for acceptance Dr. by J. Zhai. *(Corresponding author: Danlin Jia.)*

Danlin Jia, Li Wang, and Ningfang Mi are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA (e-mail: jia.da@northeastern.edu; wang.li4@northeastern.edu; ningfang@ece.neu.edu).

Natalia Valencia and Janki Bhimani are with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA (e-mail: nvale010@fiu.edu; jbhimani@fiu.edu).

Bo Sheng is with the Department of Computer Science, University of Massachusetts Boston, Boston, MA 02125 USA (e-mail: bo.sheng@umb.edu).

Digital Object Identifier 10.1109/TCC.2023.3329129

learning-based bidirectional usage-bounded memory management scheme that monitors the run-time execution performance and dynamically re-allocates memory space to Spark execution and RDD storage. We first propose a basic version of our new autotuning memory manager, named ***ATuMm***, which leverages an intuitive feedback-control solution to improve Spark performance by dynamically adjusting memory pools with a fixed adjustment step.

To obtain an optimal learning speed, the users of ATuMm need to tune the adjustment step manually. However, it is not trivia to configure this adjustment step. Significantly when the memory demands of an application vary frequently, an inappropriate adjustment step might limit the benefit of ATuMm. To address this issue, we further propose a Q-learning-based Spark memory manager, called *Q-ATuMm*, which aims to develop an intelligent agent to help make decisions of the adjustment step automatically. The goal of Q-ATuMm is to utilize a machine learning algorithm (e.g., Q-learning [5]) to adjust memory partitions in Spark dynamically and efficiently. We remark that Q-learning offers several advantages compared to other machine learning algorithms, especially in scenarios involving sequential decision-making and dynamic environments.

The main contributions of this work are as follows.

- *Understanding of two existing memory managers in Spark:* We study the infrastructure of two Apache Spark memory managers to understand how these two managers allocate memory space to the storage and execution pools. We further conduct real experiments to analyze the performance of these two managers.
- *Design and implementation of an auto-tuning memory manager:* We propose a new Spark memory manager, named ATuMm, that dynamically tunes the size of storage and execution memory pools based on the performance of current and previous tasks. We implement and evaluate ATuMm in Spark 2.4.0 and show that our new memory manager significantly improves the Spark performance.
- *Optimization of memory management by developing an intelligent agent:* We develop an intelligent agent by using the Q-Learning algorithm and integrate the agent in Spark as a new memory manager, named Q-ATuMm. We show that Q-ATuMm can further improve the performance via our new machine learning agent for both iterative data processing applications and ad-hoc database queries.
- *Analysis of memory usage and GC of Spark memory managers:* We investigate the execution memory usage and garbage collection of all four Spark memory managers (i.e., SMM, UMM, ATuMm, and Q-ATuMm). We discover that both ATuMm and Q-ATuMm decrease garbage collection time by preventing overloaded execution memory. Also, we observe that Q-ATuMm has lower latency than ATuMm.

In the remainder of this paper, we will discuss the issues of two existing memory managers and related work which motivates our design of a new memory management scheme in Section II. In Section III and Section IV, we present the detailed algorithm and the evaluation of our two new memory managers. Conclusion is presented in Section V.

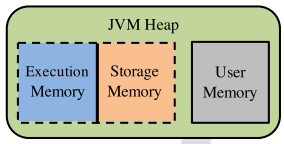

Fig. 1. Memory partition of spark memory managers.

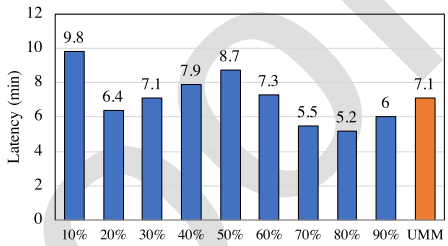

Fig. 2. Latency of application under SMM and UMM. SMM increases storage fraction from 10% to 90%.

II. Motivation and Related Work

In this section, we study the performance of Spark applications managed by two existing Spark memory managers (i.e., SMM and UMM). In both memory managers, as shown in Fig. 1, a portion of Java heap (i.e., memory in the dashed rectangle) is dedicated for processing Spark applications (called *Accessible Memory*), while the rest of memory is reserved for Java class references and metadata usage (called *User Memory*). Accessible memory is further divided into two partitions, *Storage Memory* and *Execution Memory*. The boundary between the storage memory and execution memory is fixed (i.e., static) in SMM, but flexible in UMM. Storage memory is used for caching RDDs, while execution memory is used for runtime task processing. If storage memory is already fully utilized when a new RDD needs to be cached, some old RDDs will be evicted according to the LRU (Least Recently Used) algorithm. On the other hand, if execution memory is full, all intermediate objects generated at runtime will be serialized and spilled into the disk to release memory space for subsequent task processing.

A. SMM: Static Memory Partition Analysis

To understand how memory partition can affect Spark performance, we conduct a set of experiments in a Spark cluster consisting of four homogeneous workers (see the setup in Section IV-B), with PageRank [6] as a representative benchmark. We set the boundary, which we also refer to as *storage fraction* (i.e., the ratio of storage memory to accessible memory), from 10% to 90% of accessible memory space under SMM. Since the total accessible memory dedicated to Spark applications remains constant, execution memory is decreased when storage memory is increased.

Fig. 2 first illustrates the experiment results for SMM with different storage fractions. We can observe that the Spark performance varies with different memory partitions. Intuitively, if the storage memory is too small to cache RDDs that will be reused in the following computations, the RDD processing time cannot be

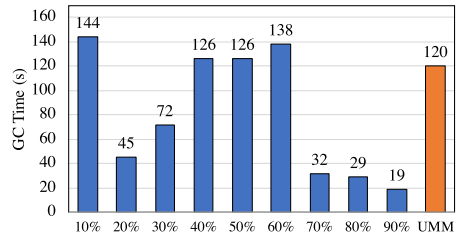

Fig. 3. GC time comparison. SMM increases storage fraction from 10% to 90%.

saved. On the other hand, if we assign too much space to storage memory, then the confined execution memory pool may trigger a high overhead of I/O communications. However, neither one of these two effects dominates the other, and the resulting joint performance depends on the characteristics of the workload. As shown in Fig. 2, the latency is not a monotonic function of the storage memory size. Therefore, we conclude that SSM yields varying performance with different storage fractions and cannot automatically achieve optimal performance.

B. Static VS. Dynamic: Latency Comparison

SMM cannot fit all kinds of workloads well because of its lack of flexibility. Compared with SMM, UMM allocates memory resources dynamically according to resource demands. Furthermore, UMM gives a higher priority to execution memory than to storage memory. Execution memory can force the storage memory pool to shrink if storage memory exceeds 50% of total accessible memory, even if it is fully utilized. Based on this mechanism, UMM guarantees sufficient memory for executing run-time tasks, which avoids the content of execution memory from being spilled into the disk to the greatest extent.

We find that UMM still cannot consistently achieve the best performance, although it strives to adjust the storage fraction based on resource demands dynamically. For example, the last bar in Fig. 2 further shows the latency of UMM. We can see that UMM does help improve the performance by obtaining lower latency than SMM with some storage fractions (e.g., 10% and 50%). Whereas UMM cannot beat SSM with a storage fraction of 20% and 70%∼90%, and thus cannot achieve optimal performance.

C. UMM Limitation: GC Impact

To explore the cause of UMM's ineffectiveness, we conduct a set of experiments to investigate the impact of garbage collection (GC) on Spark application latency. We plot the GC times of SMM with different storage fractions and that of UMM in Fig. 3. We observe that SMM has a much lower GC time when storage fraction is set to 20%, 30%, and $\geq 70\%$. In contrast, the GC time under UMM is as high as 120 seconds, about six times the lowest GC time obtained by SMM with a storage fraction of 90%. By combining the results in Figs. 3 and 2, we note that the GC time has considerable impacts on Spark performance and UMM's performance degradation results from such a long GC time.

We discover that long GCs occur under UMM because UMM expands the execution memory pool aggressively, resulting in a large amount of intermediate data in execution memory. The Java garbage collector then needs to maintain these in-memory intermediate data and thus increases the overall GC time. Such high GC time finally introduces extra latency to a Spark application's execution. Besides, there exist no explicit methods to eliminate these long GCs by configuring UMM by users. This observation motivates us to consider both GC time and execution time for dynamically adjusting memory partition. The impact of GC on Spark's performance is also investigated in existing works, which will be discussed in Section II-E.

D. Need for Learning-Based Solutions

The basic version of our new memory manager (ATuMm) is designed based on an intuitive feedback-control solution, which uses the current task's execution as the feedback to decide the increase or decrease in the boundary between the execution and storage memory pools with a fixed adjustment step. To obtain an optimal learning speed, the user must manually configure the adjustment step, which requires pre-knowledge about the workload and the system characteristics. Even with an optimal adjustment step, our ATuMm may not consistently achieve the best performance. One reason is the fixed adjustment step that cannot work well for applications with varying memory demands. Another reason is that ATuMm makes the tuning decisions heavily depending on the execution status of the current task. Motivated by the above limitations, we need to design a more comprehensive learning solution that can have an intelligent agent to "smartly" calculate rewards for dynamically tuning the adjustment step and thus optimizing the learning speed. We select Q-learning algorithm as our intelligent memory management agent for the following reasons. First, Q-learning is model-free, meaning it doesn't require a complete understanding of the underlying system dynamics. This makes it suitable for situations where the environment is complex, uncertain, or difficult to model accurately. Second, Q-learning employs temporal difference learning, allowing it to learn from each individual interaction with the environment. This characteristic makes it well-suited for online learning and environments where data arrives sequentially. Third, compared to other powerful but complicated ML/DL models, i.e., convolutional neural networks and transformers, Q-learning is light to integrate with existing systems and offers low learning overhead.

E. Gap in the Existing Works

We summarize existing works in Table I. MEMTUNE presents an algorithm that adjusts memory allocation based on the characterizations of tasks (i.e., storage-sensitive or execution-sensitive). This work considers the impact of JVM on Spark performance to decide how to balance memory allocation for obtaining a good performance. But, this work only focuses on analyzing the sensitivity of tasks and takes different actions, such as reserving more memory for storage requirements if tasks are storage-sensitive. Another work DSMM, dynamically sets the storage fraction by simply comparing the size of the data set

TABLE I
COMPARISON OF EXISTING SPARK MEMORY OPTIMIZATION WORKS

	Optimization Level	Workload Characterizing	Machine Learning	Garbage Collection
MEMETUNE [7]	Memory	Sensitivity Analysis	N/A	N/A
DSMM [8]	Memory	Data Size Analysis	N/A	N/A
SMBSP [9]	Framework	N/A	Artificial Neural Network	N/A
MLAT [10]	Framework	N/A	Regression & Clustering	N/A
PokéMem [11]	Memory	Data Size Analysis	N/A	Considered
MCS [12]	Memory	N/A	N/A	Considered
Q-ATuMm	Memory	Learning-based Analysis	Q-Learning	Considered

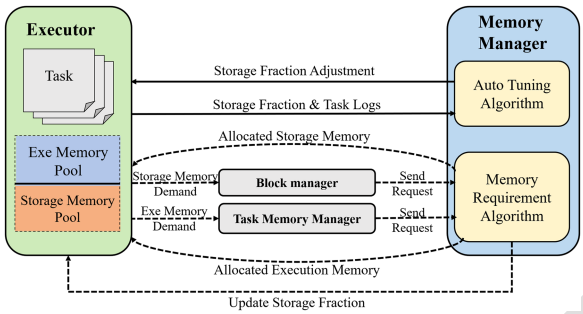

Fig. 4. New memory allocation scheme architecture.

with its memory usage. Compared to our work, these two works fail to track the memory requirement diversity at run-time, which still relies on preknowledge of the application's characteristics.

SMBSP applies Artificial Neural Network (ANN) to configure Spark's parameters automatically, including computation, cache, and storage configurations. MLAT is another work that utilizes machine learning to auto-config Spark's parameters. This work learns proper configurations for different Spark clusters as well. However, these two works optimize Spark's performance at a coarser level and lack consideration of runtime workload characteristic adjustment compared to our work. We also note that our work contributes to optimizing Spark's caching logic and can be adapted easily to [9] and [10].

PokéMem and MCS consider the impact of GC on Spark's performance and strive to optimize memory management via controlling GC. PokéMem focuses on reducing memory pressure by estimating the data size of objects created by third-party libraries. However, the estimation model is data structure- and library-dependent. MCS is close to our work which defines constraints to limit the priority of execution memory. However, it lacks dynamic adjustment of these constraints.

III. NEW LEARNING-BASED MEMORY MANAGER DESIGN

In this section, we present our new learning-based memory allocation scheme, which aims to improve the overall latency for Spark applications by considering both *resource demands* and *garbage collection impact* in dynamic memory resource allocation. Fig. 4 shows the overview of our design and illustrates the overall block diagram of Spark modules on an "Executor". A Spark cluster often consists of multiple "Executors". Each "Executor" hosts a set of running tasks and manages their storage and execution memory pools independently. In addition, there are two managers in Spark that are responsible for the memory requests sent from the "Executor" module. Specifically, the "Block Manager" manages the storage memory requirements, and the "Task Memory Manager" manages the execution memory requirements.

In our memory allocation scheme, we develop two new main modules, called *Auto Tuning Algorithm* (i.e., ATuMm or Q-ATuMm), and *Memory Management Algorithm*, and integrate them with the existing Spark modules, as shown in Fig. 4. The "Executor" periodically calls the "Auto Tuning Algorithm" to adjust the storage fraction and set the limit (or the maximum allowed) of execution memory. The "Memory Management Algorithm" further responds to the memory requirements sent by the "Block Manager" and "Task Memory Manager" modules by considering both free storage/execution memory space and the decision made by the "Auto Tuning Algorithm". Upon completing each task, the "Auto Tuning Algorithm" receives the runtime logs of the completed task and the previously completed tasks from the "Executor" module. Based on these logs, the algorithm adjusts (1) the boundary between the storage and execution memory pools and (2) the maximum allowed memory space to the execution pool. The adjustment decisions are then passed to the "Executor" for the next task execution. The above adjusting process repeatedly occurs until the last task at the "Executor" completes. Meanwhile, the "Memory Requirement Algorithm" bases on the memory requirements from the "Executor" to allocate the memory space for the RDD cache (i.e., storage memory) and task execution (i.e., execution memory). The storage fraction is then accordingly updated by this algorithm based on runtime memory demands.

A. Memory Requirement Algorithm

The *Memory Management Algorithm* is designed to allocate memory space for RDD caching and task execution. In particular, this algorithm receives the online memory requirements from the "Block Manager" and the "Task Memory Manager" modules. Specifically, our scheme maintains two parameters: "StorageFraction" and "heapStorageMemory". While the former decides the maximum available memory of the storage memory pool, the latter limits the maximum available memory of the execution memory pool. According to the current storage partition and "heapStorageMemor", this algorithm allocates available memory to the two manager modules (i.e., "Block Manager" and "Task Memory Manager") to meet their requirements.

Algorithm 1 describes the main procedures of this memory management mechanism.

Algorithm 1: Memory Requirement Algorithm.

```
1  Procedure acquireExecutionMemory(reqExe)
2      extraNeed=reqExe-freeExecutionMemory
3      if extraNeed>0 then
4          memoryBorrow=min(extraNeeded,storageMemoryPoolSize-
             heapStorageMemory,freeStorageMemory)
5          decreaseStoragePoolsize(memoryBorrow)
6          increaseExecutionPoolsize(memoryBorrow)
7          acquired = executionMemory-
             Pool.acquire(freeExecution+memoryBorrow)
8      else
9          acquired=executionMemoryPool.acquire(reqExe)
10     return acquired
11 Procedure acquireStorageMemory(reqSto)
12     memoryToFree=max(0, reqSto−freeStorageMemory)
13     if memoryToFree>0 then
14         freeStorageMemory(memoryToFree)
15     acquired = storageMemoryPool.acquire(reqSto)
16     if heapStorageMemory<usedStorageMemory then
17         heapStorageMemory=usedStorageMemory
18     return acquired
```

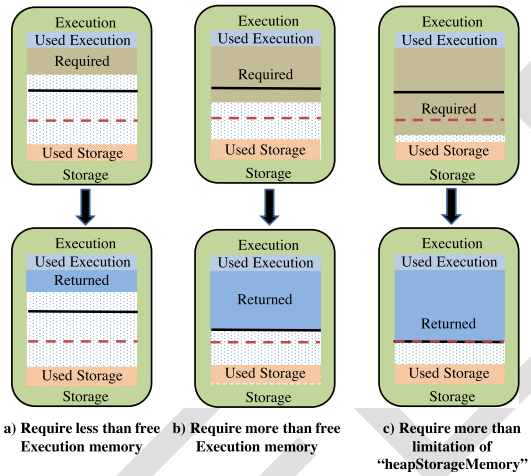

Fig. 5. Execution requirement conditions.

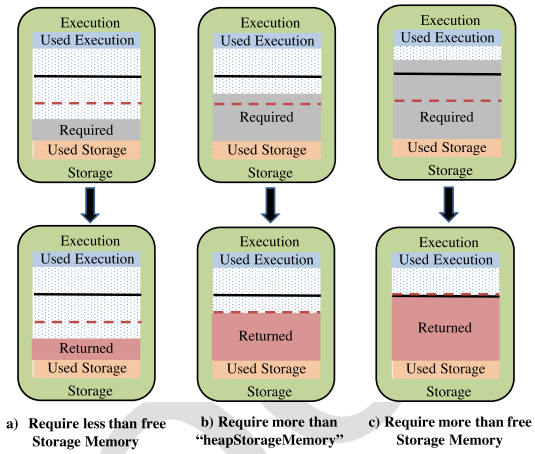

Fig. 6. Storage requirement conditions.

– *Procedure requireExecutionMemory()* takes "reqExe" as the input, which is the execution memory size required by "Task Memory Manager", and returns the actual allocated execution memory. Specifically, execution memory requirements can be one of the three scenarios shown in Fig. 5. In the figure, we plot the Spark memory pool on an "Executor", where a solid line represents the potential boundary between execution memory and storage memory. A dashed line represents the value of "heapStorageMemory", indicating the least reserved space for storage memory. Besides, we also mark the used execution and storage memory space. In the first scenario, the required execution memory is less than the free execution memory, see Fig. 5(a). Then, the procedure allocates all needed memory to "Task Memory Manager".

The second scenario is shown in Fig. 5(b), where the required execution memory exceeds the free execution memory but not beyond the limit of "heapStorageMemory". Procedure *requireExecutionMemory()* still allocates all needed memory to "Task Memory Manager" and meanwhile expands the execution memory pool by moving down the boundary bar (see the solid line in the bottom plot of Fig. 5(b)). Finally, suppose the required execution memory exceeds the boundary of "heapStorageMemory". In that case, the procedure only allocates the memory up to "heapStorageMemory" (see the dashed line in the bottom plot of Fig. 5(c)) and also moves down the boundary bar to "heapStorageMemory". Our algorithm prevents memory overallocation for task execution by limiting the memory that can be allocated to execution memory. For example, in both scenarios (b) and (c), the execution memory pool occupies part of storage memory after allocating memory to the execution memory pool. However, in scenario (c), we use "heapStroageMemory" to avoid the execution memory pool invading the storage memory pool. In this way, GC time can be reduced as discussed in Section II.

– *Procedure requireStorageMemory()* receives the required storage memory size ("reqSto") from the "Block Manager" module for allocating actual memory to cache RRDs. Similarly, we have three possible conditions of storage memory requirements, depicted in Fig. 6. If the required storage memory is less than free storage memory as shown in Fig. 6(a) and (b), then all required memory will be allocated to "Block Manager" (no matter beyond "heapStorageMemory" or not). In contrast, if the required storage memory is more than the free storage memory (see Fig. 6(c)), then only the memory space up to the boundary bar will be allocated to "Block Manager," and meanwhile, RDD eviction will be triggered to release some memory for caching new RDDs. In both scenarios 2 and 3, we further update the variable "heapStorageMemory" to be equal to the actual storage memory pool size.

It is noticeable that "Memory Management Algorithm" does change the storage fraction under some scenarios, such as the ones shown in Fig. 5(b) and (c). Thus, the storage fraction is jointly determined by both "Memory Management Algorithm" and "Auto Tuning Algorithm".

B. Auto Tuning Algorithm

Here, we first present the basic version of our auto-tuning algorithm, named ATuMm, which uses a feedback-control way to dynamically adjust the boundary of two memory pools with

Algorithm 2: ATuMm.

```
1  Procedure barChange( GCTime, executionTime)
2      curRatio=GCTime/executionTime
3      if curRatio=preRatio then
4          return None
5      else if (curRatio<preRatio and preUpOrDown=true) or
         (curRatio>preRatio and preUpOrDown=false) then
6          update preUpOrDown to ture, update preRatio
7          return (setUp(step))
8      else
9          update preUpOrDown to false, update preRatio
10         return (setDown(step))
11 Procedure setUp(step, preStorageFraction)
12     if preStorageFraction+step<100% then
13         curStorageFraction=preStorageFraction+step
14         if usedStoragePoolSize/totalStoragePoolSize>80% then
15             heapStorageMemory=heapStorageMemory+
16             step*accessibleMemory
17     update preStorageFraction
18     return heapStorageMemory, curStorageFraction
19 Procedure setDown(step, preStorageFraction)
20     if preStorageFraction−step>0 then
21         curStorageFraction=preStorageFraction−step
22         memoryEvict=memoryUsed−curStorageFraction
23         if memoryEvict>0 then
24             freeStorageMemory(memoryEvict)
25         heapStorageMemory=heapStorageMemory−
26         step*accessibleMemory
27         if heapStorageMemory>=curStorageFraction*accessibleMemory
             then
28             heapStorageMemory=curStorageFraction*accessibleMemory
29         update preStorageFraction
30     return heapStorageMemory, curStorageFraction
```

a fixed adjustment step. Then, we propose a Q-learning-based algorithm, named Q-ATuMm, which uses an intelligent agent to optimize the learning speed by automatically tuning the adjustment step.

1) Basic Version. ATuMm: When a task on the "Executor" completes, the "Auto Tuning Algorithm" takes the GC time, the execution time of the completed task, and the current storage fraction as inputs and then compares the performance of the completed task (in terms of the ratio of GC time to execution time) with that of the previous tasks to make the adjustment decision. In particular, the "Auto Tuning Algorithm" returns two variables: (1) a new storage fraction ("curStorageFraction") for the potential memory partition, and (2) a new "heapStorageMemory" variable to indicate the least memory reserved for storage memory. Using these two variables, ATuMm can adjust the memory partition with a limit on the maximum memory that can be allocated to execution memory. Algorithm 2 shows the pseudo-code of the "Auto Tuning Algorithm".

Both *setUp()* and *setDown()* repartition the accessible memory to the storage and execution pools based on the decision made by *barChange()*. We also remark that the variable "heapStorageMemory" is new in our design, which plays a critical role in avoiding long GC time resulting from over-allocated execution memory. Later, we present how this variable is used in the "Memory Requirement Algorithm" to control the actual memory space for RRD caching and task execution.

– Procedure *barChange()* receives GC time and execution time of the current task from the "Executor" module. We consider the ratio of GC time to execution time as a measurement of Spark performance. A low ratio indicates a "good performance", vise verse. Then, *barChange()* makes an adjustment decision from one of three possible actions (i.e., keep still, increase storage fraction, and decrease storage fraction). In particular, we use two variables, "preRatio" and "preUpOrDown" to record the ratio of GC time to the execution time of previous tasks and the last adjustment decision, respectively. We compare "curRatio" with "preRatio" to calculate the reward of the last adjustment. If the current task yields a better performance (i.e., "curRatio" is lower than "preRatio"), the boundary-moving decision that we previously made (i.e., "preUpOrDown") gets a reward. Thus, we decide to keep moving the boundary further in the same direction as the last task. Otherwise, we move the boundary in a direction that is opposite to that of the last adjustment. Besides these two actions, if the Spark performance converges (i.e., the current ratio is equal to the previous ratio), the boundary keeps still. After taking the new action, the storage fraction changes, and two variables (i.e., "preRatio" "preUpOrDown") are updated for the next decision.

– Procedures *setUp()* and *setDown()* control how to expand or shrink the storage and execution memory pools base on the decision made in *barChange()*. As mentioned in Section II, Spark memory is divided into two pools, i.e., storage memory and execution memory. We thus consider there exists a partition "bar" between storage and execution memory in Spark. Setting the bar up means enlarging the storage memory pool and shrinking the execution memory pool, while setting the bar down means decreasing the storage memory pool and expanding the execution memory pool. In ATuMm, users can configure the percentage of accessible memory (indicated as "step") that will be increased or decreased in each adjustment.

It is challenging to move the partition bar if both storage and execution memory pools are fully utilized. A mechanism is required to determine which objects should be evicted. LRU (Least Recently Used), an existing RDD caching algorithm, is applied by the Spark block manager for storage memory. We adopt this caching algorithm to manage the RDD evictions from storage memory. For execution memory, *barChange()* is called only when a task has finished its computation and released all its occupied memory resources. Thus, there is no need to evict objects from the execution memory pool. This is also one reason we choose to adjust the memory boundary after each task's completion.

Procedure *setUp()* takes "preStorageFraction" and the predefined parameter "step" (e.g., 5%) as inputs to determine a new storage fraction ("curStorageFraction") to repartition the memory and a bound ("heapStorageMemory") to reserve the least storage memory space. In detail, *setUp()* increases the storage fraction by "step" (see lines 12 and 13 in Algorithm 2) if the new storage memory pool size is less than the overall available memory space. Meanwhile, *setUp()* updates "heapStorageMemory" only if 80% of the storage memory is used (see lines 14, and 15 in Algorithm 2). The difference between the storage memory pool size and "heapStorageMemory" will be the potential memory space allocated to execution memory.

Procedure *setDown()* has the same inputs and outputs as *setUp()* to shrink the storage memory pool. In details,

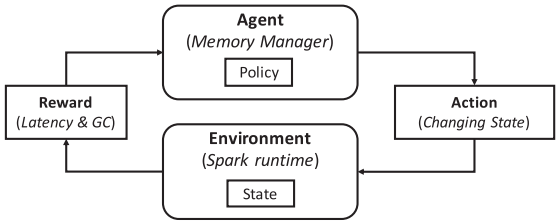

Fig. 7. Reinforcement Learning (Q-Learning) Algorithm in Q-ATuMm. We define 1) agent represents the memory manager, 2) environment is Spark runtime, 3) state represents "StorageFraction" and "heapStorageMemory" that limit the allocation of storage and execution memory, 4) action is changing state, and 5) reward is calculated from latency and GC time.

Algorithm 3: Q-ATuMm

```
1  Procedure initializeAgent()
2      Initialize stateSpace
3      Initialize actionSpace
4      Initialize QTable
5      Initialize α, ϵ, γ
6      Initialize stateIndex, actionIndex
7  Procedure QLearningAgent(GCTime, executionTime, stateIndex,
     actionIndex)
8      reward=taskTime/(GCTime+δ)
9      QTable(stateIndex, actionIndex) = updateQTable(reward,
         stateIndex, actionIndex)
10     rnd = random(0, 1.0)
11     if rnd< ϵ then
12         actionIndex = random(0, actionSpace.length)
13     else
14         actionIndex = GetIndex(QTable(stateIndex).max)
15     action = actionSpace(actionIndex)
16     state = stateSpace(stateIndex)
17     return action
18 Procedure updateQTable(reward, stateIndex, actionIndex)
19     QValue = QTable(stateIndex, actionIndex)
20     stateValue = γ*(QTable(stateIndex).max - QValue)
21     QValue = QValue + α*(reward+stateValue)
22     return QValue
```

setDown() decreases the storage fraction by "step" (see line 20 in Algorithm 2). However, it needs to consider RDD evictions to release the reduced storage memory additionally (see lines 21 and 22 in Algorithm 2). For example, if the current storage memory pool is 5 GB with 4.5 GB used, and the potential storage memory becomes 4 GB, then the memory space ('memoryEvict") that needs to be released is 0.5 GB. *setDown()* then needs to trigger the caching algorithm to evict cached RDDs to shrink the storage memory pool. Finally, *setDown()* updates (or decreases) "heapStorageMemory"by "step" of accessible memory. If "heapStorageMemory" is more than the new storage memory, then *setDown()* sets 'heapStorageMemory" to be equal to the new storage memory (see line 25 in Algorithm 2).

2) Q-Learning Based Version: Q-ATuMm: As discussed in Section II-D, ATuMm suffers from the inflexibility of the adjustment step. In order to optimize the adjustment speed, we further refine our auto tuning algorithm by using reinforcement learning techniques to automatically set the adjustment step for changing the memory boundaries. On the other hand, Spark applications process data in batches, possessing consistent memory and computation characteristics, which can be learned by reinforcement learning efficiently. Q-learning is a specific algorithm within the broader field of reinforcement learning, which receives feedback from the objective and makes decisions to optimize the rewards. As shown in Fig. 7, an *agent* interacts with an *environment* by taking actions, then the environment returns a reward of the action to the agent and updates the state of the environment. By exploiting different actions across all possible states, the agent can produce an optimal policy to manipulate the states of the environment.

Q-learning maintains a Q-table, where the columns and rows represent states and actions. The values (i.e., value function) in the Q-table represent the expectation of benefits of applying an action, given a state. The agent updates the value function based on an equation (particularly Bellman equation [13]). Specifically, Q-learning maintains an exploration-exploitation balance, ensuring that the agent explores new actions and state-action pairs while exploiting learned information to make optimal decisions. Theoretically, an epsilon-greedy exploration strategy, as used in the Bellman equation, guarantees that all state-action pairs are visited infinitely often, which is crucial for convergence. Another important factor in Q-learning is the convergence rate. The convergence rate of Q-learning depends on factors such as the learning rate schedule and the characteristics of the environment. In practice, while Q-learning converges asymptotically, convergence speed can vary, and certain modifications, like learning rate annealing, can influence the convergence rate. We evaluate the impact of learning rate and other hyper-parameters in Section IV-C4.

In Q-ATuMm, when the "Executor" finishes a task, the agent (i.e., memory manager) calculates the reward of the last action based on the execution time and GC time of the current task. Then Q-ATuMm updates the policy and makes a decision about which is the next state. Specifically, *InitializeAgent()* initializes all parameters before running applications. *QLearningAgent()* uses the garbage collection time and execution time of the completed task to calculate the reward of the current action and calls *UpdateQTable()* to update values of the current state and action in Q-table. *QLearningAgent()* then decides the action to execute the following task by either exploring a new action or exploiting a known action. We note that Q-ATuMm creates a two-dimension discrete action space, where each element in the action space represents a pair of "StorageFraction" and "heapStorage-Memory", as introduced in Section III-A. We define "StorageFraction" and "heapStorage-Memory" as ratios of the overall heap size, ranging from 1% to 99%. The status space is the same as the action space. Algorithm 3 describes the details of Q-ATuMm. Q-ATuMm trains the model on-the-fly.

– Procedure *initializeAgent()* initializes the state space, the action space and the Q-table. We denote α as the learning rate, representing the length of the step to update the value function. ϵ is the exploration ratio, which indicates how much the agent prefers to explore unknown actions. We denote γ as a discount factor reflecting how much the future rewards contribute to the current update.

– Procedure *QLearningAgent()* receives the garbage collection and execution time of the task, with the state of current "stateIndex" and "stateAction", which locate the value function in the Q-table to update. Because our goal is to minimize garbage

TABLE II
TESTBED CONFIGURATION

Component	Specs
Host Server	Dell PowerEdge T310
Host Processor Speed	2.93GHz
Host Memory Capacity	16GB DIMM DDR3
Host Memory Data Rate	1333 MHz
Host Storage Device	Western Digital WD20EURS
Host Disk Bandwidth	SATA 3.0Gbps
Host Hypervisor	VMware Workstation 12.5.0
Processor Core Per Node	1 Core
Memory Size Per Node	1 GB
Disk Size Per Node	50 GB

collection and reduce the overall latency, *QLearningAgent()* defines the reward as the ratio of the execution time (GC time plus others) to the GC time plus a constant number (i.e., $\delta = 0.01$) to avoid zero denominators (see line 8 in Algorithm 3). *UpdateQTable()* is then called to update the value function in the Q-table. *QLearningAgent()* uses a parameter ϵ to decide to explore a random action or to exploit the action with the largest benefit (see lines 11-14 in Algorithm 3). A larger ϵ means the agent prefers to explore unknown actions. Finally, *QLearningAgent()* returns the action to the "Executor" to execute the following tasks.

– Procedure *updateQTable()* takes the reward as an input to calculate the new value in Q-table based on the Bellman equation [13]. First, *UpdateQTable()* locates the value in Q-table and then computes the "stateValue" to estimate the reward of the next state. It is worth pointing out that the parameter γ is used to decide how important future decisions are. A larger γ indicates the agent relies more on the future reward than the current one. Finally, *UpdateQTable()* updates the "Q value" with the current reward and the estimated future reward. The parameter α is used as the learning rate to control how fast the agent learns from the rewards. There is a trade-off between learning speed and accuracy. A larger learning rate can allow the agent to learn and move faster to the optimal solution, but meanwhile, has a higher possibility of causing the agent to be trapped in a locally optimal point.

IV. EVALUATION

In this section, we discuss the implementation and the evaluation of ATuMm and Q-ATuMm in a real Spark cluster. We aim to investigate the performance in terms of latency, memory usage, and garbage collection at run-time. We use default UMM and SMM mode as our baseline, which is discussed in Section II.

A. Testbed

We conduct our experiments in a Spark cluster with one driver and four workers that are homogeneous to each other. The cluster is deployed on the Dell PowerEdge T310 and hypervised by VMware Workstation 12.5.0. Each node in the Spark cluster is assigned 1 CPU, 1 GB memory, and 50 GB disk space. Table II summarizes the details of our testbed configuration.

We implement ATuMm and Q-ATuMm as new portable memory manager modules, besides SMM and UMM, in Apache Spark 2.4.0, which contain functions interacting with other Spark modules. It is noticeable that our new memory manager can also be integrated into Spark from the version of 1.6.0 to 2.4.0. The source code is available on GitHub.[1] The LOC is 2,428 in total. Specifically, we develop functions *acquireStorageMemory()* and *acquireExecutionMemory()* to allocate storage and execution memory to "Block Manager" and "Task Memory Manager", respectively. We also integrate a profile collector in the "Executor" module to collect task logs. Specifically, ATuMm applies function *barChange()* to receive these task logs and calls functions *increaseStorageFraction()* or *decreaseStorageFraction()* to adjust memory partition. Meanwhile, Q-ATuMm uses function *updateQTable()* to maintain the Q-Table for the agent to perform reinforcement learning. Furthermore, we integrate a memory usage analyzer in ATuMm and Q-ATuMm to collect the run-time memory usage information. Users can replace the existing Spark memory manager to ATuMm or Q-ATuMm by simply setting a configurable parameter before submitting a Spark application.

B. ATuMm Evaluation

We set the accessible memory and the initial storage fraction of ATuMm as the same as those of UMM (i.e., accessible memory is 60% of JVM heap, and storage memory is initialized as 50% of accessible memory). The step to increase or decrease storage fraction in each adjustment is configured as 5% of accessible memory by default. Furthermore, the window size representing the number of previous tasks is set as 20% of activated tasks by default. Users can pre-configure these parameters in ATuMm before launching any Spark applications.

1) Latency Analysis: We evaluate and compare the performance of Spark applications under three memory managers (SMM, UMM, and ATuMm) by conducting experiments with different applications. We choose PageRank and K-means as benchmarks because these two applications are two ubiquitous techniques, which are widely applied in machine learning and data mining applications [6], [14]. Considering the duration of experiments, we report results for a workload of 1 GB input data for applications.

Fig. 8(a) and (b) illustrate the latency of PageRank and K-means under different memory managers. We set various storage fraction under SMM manually, and compare the latency of SMM with that of UMM and ATuMm. In Fig. 8(a), we observe that the performance of UMM beats SMM with some storage fractions (e.g., 40% to 60%). However, when SMM sets the storage fraction to 80%, it reaches the best performance, which achieves 27% shorter latency compared to UMM. More importantly, the latency of our ATuMm is close to the lowest among all, and our ATuMm beats UMM as well. Moreover, as shown in Fig. 8(b), our ATuMm can achieve the best performance (i.e., the lowest latency), compared with both UMM and SMM. We conclude that ATuMm outperforms the other two existing memory managers with the same computation resources allocated.

[1] https://github.com/DanlinJia/spark _ core_ATMM

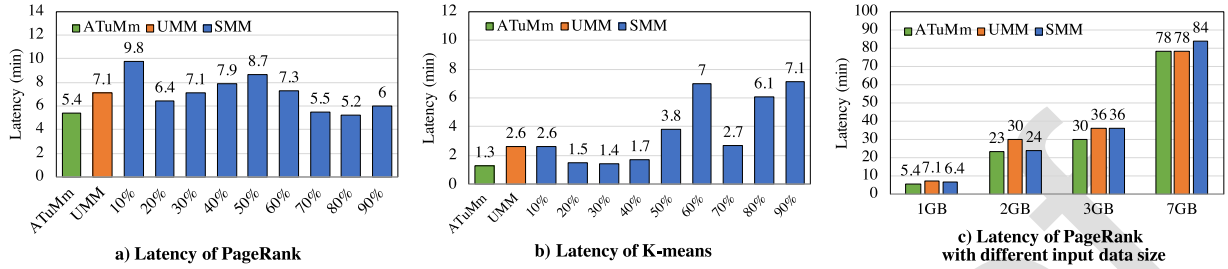

Fig. 8. Execution time of applications under SMM, UMM and ATuMm.

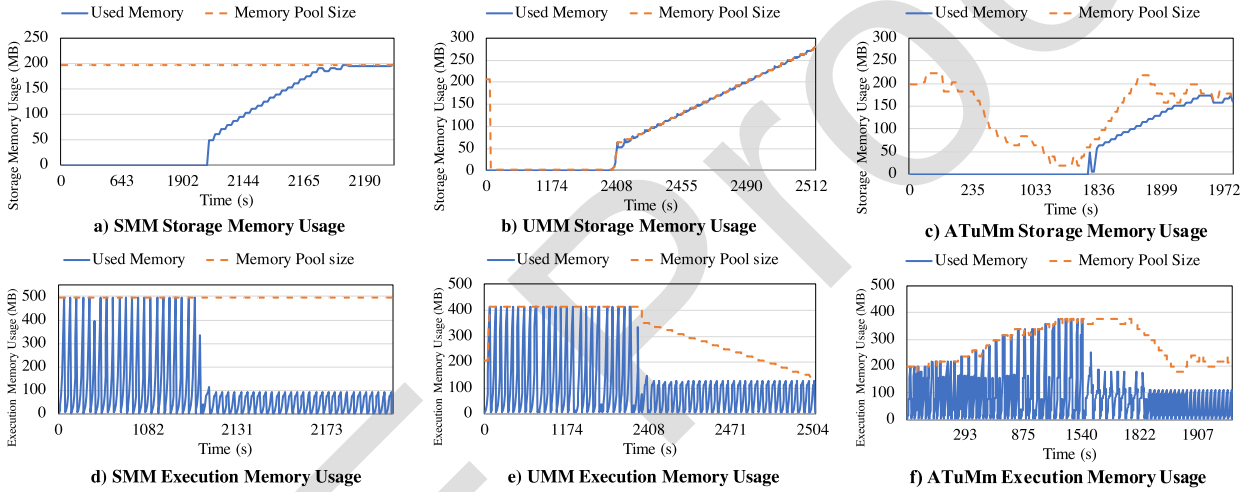

Fig. 9. Memory usage analysis of SMM, UMM and ATuMm.

2) Sensitivity Analysis: We also conduct a set of experiments to investigate the sensitivity of input data size, where we compare the performance of PageRank under three memory managers in the default mode with different input data sizes, such as 1 GB, 2 GB, 3 GB and 7 GB. As shown in Fig. 8(c), ATuMm achieves the best performance when the input data sizes are 1 GB, 2 GB, and 3 GB. Compared to UMM, ATuMm improves the latency by 25%. We interpret this improvement by observing that ATuMm leverages the GC time to repeatedly adjust the boundary between storage and execution memory, which prevents the Spark applications from a long GC duration as UMM introduced. When input data grows up to 7 GB, the overwhelming workload takes full usage of execution memory to process input data. Both UMM and ATuMm expand the execution memory pool aggressively to satisfy the massive execution memory requirements. As a result, UMM and ATuMm obtain similar performance (e.g., 78 minutes for 7 GB input data), which is better than that of SMM.

3) Memory Usage and Garbage Collection Analysis: We further look closely at the execution details of three Spark memory managers by plotting their memory usages in Fig. 9, where PageRank is running with 3 GB input data. Fig. 9(a)∼(c) present the storage memory usage across time under the three memory managers, while Fig. 9(d)∼(f) depict the corresponding execution memory usage. In each plot, the dashed line is the maximum memory size accessible for the corresponding memory (such as storage or execution), and the solid line is the actual usage of the memory pool.

From Fig. 9(a)∼(c), we observe that the storage memory utilization is similar for all three memory managers, which increases up to the maximum allowed storage pool size as time goes by. This is because RDDs are cached periodically in PageRank. Whereas, the storage memory pool sizes are different under three memory managers at different times. That is, both UMM and ATuMm dynamically change the storage memory pool sizes instead of the fixed one as SMM does. As shown in Fig. 9(a), the static storage memory pool starts to evict RDDs when the utilization of the storage memory pool is full. However, in Fig. 9(b), UMM drops the size of its storage memory pool to almost zero and then increase its storage pool when RDDs are cached. The storage memory pool changes more dynamically under ATuMm, as shown in Fig. 9(c). ATuMm first drops the storage fraction gradually as the execution memory pool expands, and then increases it as RDDs are cached. It is noticeable that ATuMm not only increases the storage memory pool based on storage memory requirements to cache RDDs, but also adjusts the pool size more rapidly than UMM to limit the execution memory pool size.

We further show our analysis of the execution memory usage under three memory managers in Fig. 9(d)∼(f). SMM fixes the execution memory pool size regardless of workload diversity,

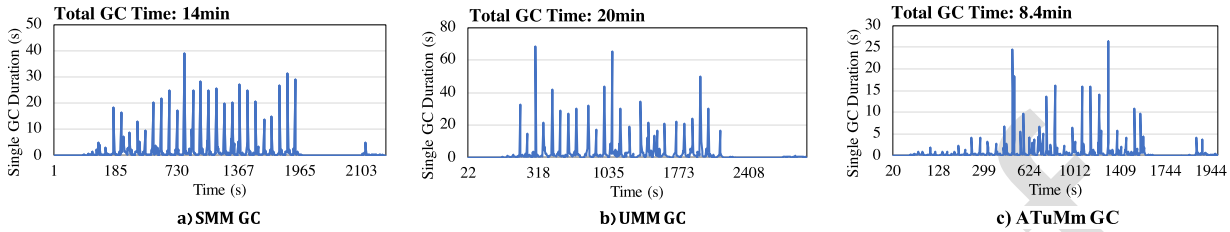

Fig. 10. GC analysis of SMM, UMM and ATuMm.

while UMM and ATuMm alter the execution memory pool size based on demands. Fig. 9(e) shows that the execution memory pool of UMM expands aggressively and occupies almost all accessible memory when the first execution requirement comes. Contrarily, in Fig. 9(f), ATuMm increases gradually across time until it satisfies all execution requirements. This is because UMM expands the execution memory pool only based on execution memory requirements, while ATuMm further considers the impact of GC on Spark performance to control the expansion of the execution memory pool. In addition, as the execution memory usage drops, UMM still gives the execution memory pool as much memory space as possible (i.e., all memory except that for caching RDDs). Conversely, ATuMm decreases the execution memory pool size more rapidly to limit the memory allocated to the execution memory pool. By this way, ATuMm can effectively prevent Spark applications from long GC durations introduced by overloaded execution memory. We can observe that the execution memory pool size converges to around 200 MB, which guarantees enough memory for task execution and further offers a relatively low GC time.

We next present our observation regarding GC time. To show our observations, we use the PageRank application with 3 GB input data as representative and compare GC time using three memory managers. Fig. 10 shows the duration of garbage collection during the runtime of the application, where each spike represents an occurrence of a full GC (i.e., JVM stops all tasks and scans the whole heap to remove unreferred objects) that majorly contributes to GC time [15]. Fig. 10(a) shows that the maximum full GC time of SMM is around 40 seconds. While, under UMM, a full GC can take more than 70 seconds, see Fig. 10(b). More importantly, we can observe that the full GCs under ATuMm are all below 30 seconds in Fig. 10(c), which is smaller than both SMM and UMM. Besides, We observe that fewer spikes occurred under ATuMm than under UMM and SMM, which means that the frequency of full GCs under ATuMm is also lower than SMM and UMM. We also record the total GC time of SMM, UMM, and ATuMm, which is $14\tilde{}min$, $20\tilde{}min$ and $8.4\tilde{}min$, respectively. Since we use 4 executors in the experiment, the GC time of each executor should be divided by 4, which is considered as the contribution of GC to the overall execution time. Thus, we can conclude that ATuMm is able to significantly reduce the maximum and the total time of GCs when compared to SMM and UMM and thus accelerates the execution of Spark applications with minimum makespan (i.e., total execution length).

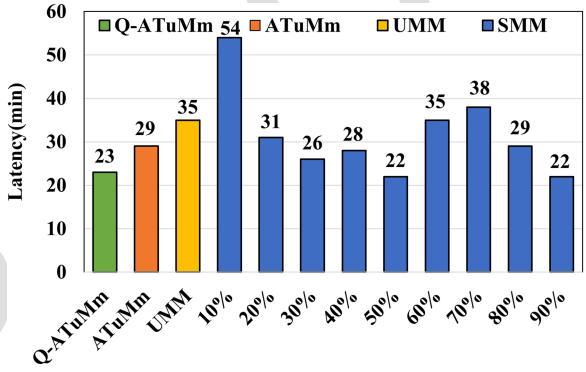

Fig. 11. Latency of PageRank under SMM, UMM, ATuMm, and Q-ATuMm.

C. Q-ATuMm Evaluation

We further implement and evaluate our Q-learning based version Q-ATuMm. We construct experiments on different categories of workloads (i.e., data-intensive applications and business queries) to evaluate the performance of Q-ATuMm, compared with that of SMM, UMM, and ATuMm. We tune the three hyper-parameters (i.e., learning rate, exploration ratio, and discount factor as shown in Section III-B2) in Q-ATuMm to achieve the best performance. The discussion on these hyper-parameters will be shown later in this section.

1) PageRank Analysis: We first construct the same experiments with PageRank on Q-ATuMm as shown in Section IV-B1. In order to trigger intensive data loading and processing, we increase the input data size to 5 GB. We observed that the application has fewer iterations to execute when the input size is small. Therefore, the Q-learning algorithm has fewer samples to learn. The performance of Q-ATuMm is worse with small data size. We also fix the number of iterations in PageRank as 20 in all experiments.

Fig. 11 illustrates the latency of PageRank under the four different memory managers. We manually set SMM storage fractions from 0.1 to 0.9 to observe the optimal latency experimentally. We observe that the best performance under SMM is achieved when the storage fraction is 50% and 90%, while UMM cannot reach that, which is consistent with our observations in Section IV-B1. On the other hand, we observe that both ATuMm and Q-ATuMm outperform UMM. More importantly, Q-ATuMm further reduces the latency by 28% compared to ATuMm.

TABLE III
QUERY CLASSIFICATION

No.	Resource Intensity	Queries
1	I/O Intensive	Q1, Q3, Q4, Q10, Q21
2	CPU Intensive	Q1, Q3, Q6, Q12, Q13, Q21

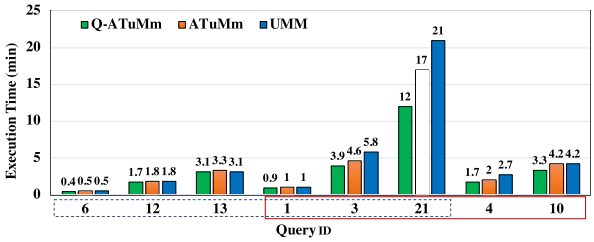

Fig. 12. Latency of TPC-H queries. Queries within the blue dashed box are CPU intensive. Queries within the red solid box are I/O intensive.

TABLE IV
EXECUTION TIME AND GC TIME COMPARISON

Manager	Execution Time (min)	GC Time (min)
Q-ATuMm	23	21.48
ATuMm	29	26
UMM	35	31.72
SMM 0.9	22	20.24

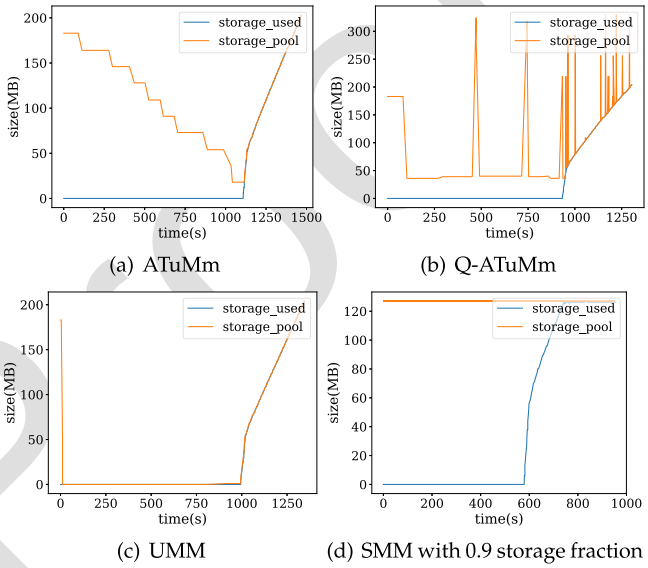

Fig. 13. Storage memory usage among all four memory managers.

2) Workload Intensity Analysis: Q-ATuMm is further evaluated on a decision support benchmark named TPC-H [16] in the context of Apache Spark. TPC-H consists of twenty-two business-oriented queries and concurrent data modifications. TPC-H evaluates the performance of decision support systems by executing ad-hoc queries on a generated synthetic data set. In our experiment, we select representative queries running on a 10 GB data set. Work [17] investigates characteristics of TPC-H queries and classifies them based on resource intensity. We select two types of queries in TPC-H to evaluate Q-ATuMm, as shown in Table III. CPU Intensive quires contain operations like *order* and *select*, while I/O intensive quires either need to load large data set into memory or perform operations on multiple data sets, e.g., *join*. It is worth noticing that some quires can be both CPU and I/O intensive (e.g., Q1, Q3, and Q21).

We compare the performance of selected queries under Q-ATuMm with that under ATuMm and UMM. The first six queries in Fig. 12 illustrates the latency of CPU intensive queries with different memory manages. We observe that the latency of Q1, Q6, Q12, and Q13 does not have a visible variance among three memory managers, while Q-ATuMm outperforms the other two in Q3 and Q21. Our experimental results indicate that CPU-intensive queries hardly benefit from both ATuMm and Q-ATuMm, as their performance heavily relies on CPU resources. The last five queries in Fig. 12 are I/O intensive queries that need to load data into memory and trigger more RDD caching, which can significantly benefit from our new design. Thus, we observe a decent latency reduction above 20% in Q-ATuMm, compared with that in UMM. For Q1, we find that although Q1 needs to join two tables, each table is small. Therefore, even though Q1 is also classified as an I/O extensive query, its execution time is not reduced significantly by Q-ATuMm.

3) Memory Usage and Garbage Collection Analysis: To closely analyze the performance improvement under Q-ATuMm, we further collect the aggregated GC time of all executors under ATuMm, Q-ATuMm, UMM, and SMM with 0.9 storage fraction and show both total execution time (i.e., latency) and GC time for PageRank in Table IV. We first notice that GC time plays a dominant role in the total execution time. By gradually reducing the storage fraction when the execution memory pool expands, our memory managers (i.e., ATuMm and Q-ATuMm) can significantly reduce the GC time by 17% and 32%, compared to UMM. Q-ATuMm further reduces the GC time (close to the optimal one as shown in the row of SMM 0.9 in Table IV) by using the Q-learning reinforcement technique to set the adjustment step for changing the memory boundaries automatically.

We further show storage memory usage among all four memory managers in Fig. 13. First, SMM has a fixed storage pool size (e.g., 0.9 storage fraction), and its storage memory usage increases up to the maximum allowed storage pool size as time goes by, which is caused by caching RDDs in each iteration. On the other hand, UMM, ATuMm, and Q-ATuMm dynamically change the storage memory pool size as time progresses based on the run-time memory resource demands. For example, as shown in Fig. 13, all of them start to increase the storage pool size at around 1000 seconds when RDDs start being cached.

However, we can observe that UMM immediately decreases the storage memory pool size to around zero to give more space to the execution memory pool, which unfortunately can cause a long GC time, as we discussed in Section II-C. To address this issue, ATuMm decreases the storage memory pool size gradually until it converges with the storage memory used size. It is visible that ATuMm gradually adjusts storage memory size based on the caching of RDDs, but it is less aggressive than UMM. For Q-ATuMm, we observe that the randomness that comes from exploration causes the spikes as the storage

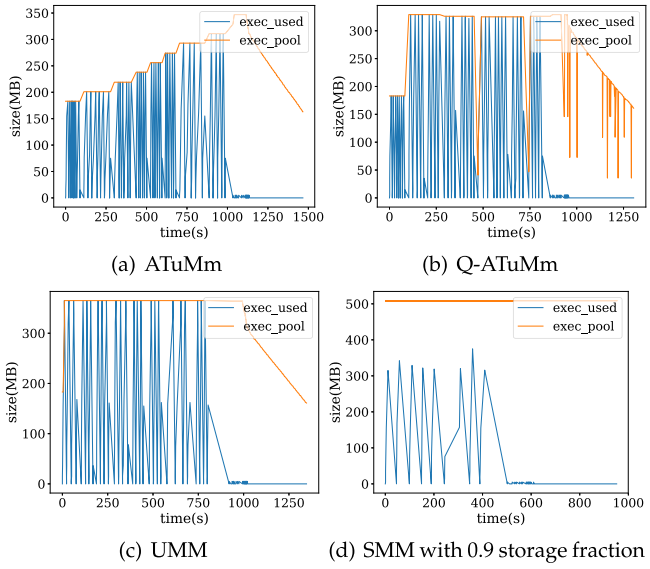

Fig. 14. Execution memory usage among all four memory managers.

memory pool size is dynamically adjusted. We also notice that the memory storage pool size decreases to below 50 almost from the starting point and stays there for about 900 seconds before the demand for storage memory increases because of RDD caching. In conclusion, we see that Q-ATuMm converges faster than ATuMm but less aggressive than UMM.

We also show execution memory usage among all four memory managers in Fig. 14. SMM's execution memory pool size remains fixed even though the actual execution memory usage is always lower than the allocated one, which indicates that SMM cannot fully utilize the execution memory, and meanwhile, it avoids triggering larger GC time. Based on the workload demands, UMM, ATuMm, and Q-ATuMm dynamically alter the execution memory pool size, which again proves to be more beneficial for execution memory utilization. ATuMm gradually increases execution storage as time passes, which helps reduce the long GC time. Q-ATuMm's execution memory pool size, on the other hand, is adjusted considerably to execution memory usage and converges at around 150 seconds, which is faster than ATuMm. The observation shows that our design of Q-ATuMm can converge fast to the run-time execution memory demands, but not as aggressive as that in UMM, which shortens GC time and saves execution time.

4) Hyper-Parameter Tuning: We finally discuss the impacts of three hyper-parameters, i.e., learning rate (α), exploration ratio (ϵ), and discount factor (γ), on Q-ATuMm's performance. We conduct a set of sensitivity analysis tests by setting different values of these hyper-parameters to run PageRank applications. Instead of extensively exploring all possible combinations, we selectively fix any two hyper-parameters and change the third one. Table V summarizes the top 5 combinations that obtain the best latency.

We find that three out of five appropriate values for the learning rate α are 0.3. Although a higher learning rate may guarantee Q-ATuMm converges quickly, it is possible to be trapped in a locally optimal solution. A small learning rate ensures that Q-ATuMm can achieve the optimal global solution, even with a slower speed. We also set the exploration ratio ϵ to 0.1 because a lower exploration ratio can allow more exploitation than exploring different states and identify the best values for achieving the optimal performance. As Q-ATuMm has a relatively simple state space, we expect Q-ATuMm to learn on the known states instead of exploring around randomly. Finally, considering that the discount factor determines the importance of future rewards, and PageRank is an iterative application with periodic patterns across time, we find that a significant discount factor (i.e., 0.9) can speed up the convergency.

TABLE V
LATENCY OF TOP 5 HYPER-PARAMETER COMBINATIONS

Learning Rate	0.3	0.3	0.2	0.3	0.7
Exploration Ratio	0.1	0.5	0.2	0.9	0.1
Discount Factor	0.9	0.9	0.9	0.9	0.9
Latency (min)	23	24	24	24	24

We also tune the three hyper-parameters of Q-ATuMm to investigate their impacts on the performance of TPC-H applications. Similarly, we extensively change the values from 0.1 to 0.9 for each hyper-parameter and receive the following observations. First, we find that the discount factor is not sensitive for both CPU intensive and I/O intensive queries because most of the queries are completed within a short period before the discount factor takes effect. Second, the exploration ratio is less sensitive for CPU-intensive queries than for I/O intensive queries because CPU-intensive queries hardly benefit from Q-ATuMm. Finally, more than one combination of the three hyper-parameters can lead to the same best performance, which indicates that TPC-H quires are not sensitive to hyper-parameters of Q-ATuMm as they are not iterative applications.

V. CONCLUSION

Apache Spark speeds up large-scale data processing by leveraging in-memory computation. However, the existing Spark memory manager (UMM) incurs long garbage collections, which degrades Spark performance significantly. In this work, we first present a new Spark memory manager (ATuMm) that leverages the feedback of GC time and memory demands to partition the memory pool dynamically. We further adopt a reinforcement learning algorithm to develop an intelligent agent (Q-ATuMm) to manage memory partition for complicated workloads. We implement ATuMm and Q-ATuMm in Spark 2.4.0 and construct experiments in a real Spark cluster. We find that ATuMm obtains around 25% improvement of Spark performance, compared with existing memory managers in the best case. By applying learning-based memory management, Q-ATuMm can further improve Spark's performance to 34%. We contribute the latency improvement to successfully reducing the GC time for both ATuMm and Q-ATuMm. In the future, we plan to evaluate our design on a larger volume of applications with different types of resource intensity. By constructing experiments extensively, we are able to find a hyper-parameter combination that provides optimal performance for general data-processing

applications. We also plan to integrate other ML algorithms, e,g., LSTM, to compare cost and performance with Q-learning.

REFERENCES

[1] M. Armbrust et al., "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] T. White, *Hadoop: The Definitive Guide*. Sunnyvale, CA, USA: Yahoo Press, May 2012.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.

[4] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.

[5] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133653–133667, 2019.

[6] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, AMPLab, EECS, UC Berkeley, 2013, Art. no. 2.

[7] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE: Dynamic memory management for in-memory data analytic platforms," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2016, pp. 383–392.

[8] S.-J. Chae and T.-S. Chung, "DSMM: A dynamic setting for memory management in Apache Spark," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 143–144.

[9] M. A. Rahman, J. Hossen, and C. Venkataseshaiah, "SMBSP: A self-tuning approach using machine learning to improve performance of spark in Big Data processing," in *Proc. IEEE 7th Int. Conf. Comput. Commun. Eng.*, 2018, pp. 274–279.

[10] D. Nikitopoulou, D. Masouros, S. Xydis, and D. Soudris, "Performance analysis and auto-tuning for spark in-memory analytics," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2021, pp. 76–81.

[11] M. Kweun, G. Kim, B. Oh, S. Jung, T. Um, and W.-Y. Lee, "PokéMem: Taming wild memory consumers in Apache Spark," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 59–69.

[12] Z. Zhu, Q. Shen, Y. Yang, and Z. Wu, "MCS: Memory constraint strategy for unified memory manager in Spark," in *Proc. IEEE 23 rd Int. Conf. Parallel Distrib. Syst.*, 2017, pp. 437–444.

[13] C. Sammut and G. I. Webb, Eds., *Bellman Equation*. Boston, MA, USA: Springer, 2010, pp. 97–97. [Online]. Available: https://doi.org/10.1007/978--0-387-30164-8_71

[14] U. R. Raval and C. Jani, "Implementing & improvisation of K-means clustering algorithm," 2016.

[15] R. Xin and J. Rosen, "Tuning Java garbage collection for Apache Spark applications," [Online]. Available: https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

[16] L. Yue-Peng, "TPC-H analysis and test tool design," *Comput. Eng. Appl.*, 2007.

[17] M. Bayati, J. Bhimani, R. Lee, and N. Mi, "Exploring benefits of NVMe SSDs for BigData processing in enterprise data centers," in *Proc. 5th Int. Conf. Big Data Comput. Commun.*, 2019, pp. 98–106.

Danlin Jia is a senior storage architecture engineer with Memory Solutions Lab, Samsung Semiconductor Inc.

Li Wang is currently working toward the PhD degree with Northeastern University in Boston, Massachusetts.

Natalia Valencia received the master's degree in cybersecurity from Florida International University.

Janki Bhimani is an assistant professor with Florida International University, Miami.

Bo Sheng is an associate professor with Computer Science Department, University of Massachusetts Boston.

Ningfang Mi is an assistant professor with the Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts.