ATuMm: Auto-tuning Memory Manager in Apache Spark

Danlin Jia*, Janki Bhimani*, Son Nam Nguyen†, Bo Sheng† and Ningfang Mi*
*Department of Electrical and Computer Engineering
Northeastern University, Boston, USA
Email: jia.da@husky.neu.edu, bhimani@ece.neu.edu, ningfang@ece.neu.edu
†Department of Computer Science
University of Massachusetts Boston, Boston, USA
Email: sonnam.nguyen001@umb.edu, shengbo@cs.umb.edu

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 found that the static memory manager (SMM) lacks flexibility, while the unified memory manager (UMM) puts heavy pressure on the garbage collection of JVM on which Spark resides. To address these issues, we design an auto-tuning memory manager (ATuMm) to support dynamic memory allocation with the consideration of both memory demands and latency introduced by garbage collection. We implement our new memory manager in Spark 2.2.0 and evaluate it by conducting experiments in a real Spark cluster. Our experimental results show that our auto-tuning memory manager can reduce the total garbage collection time and thus further improve the performance (i.e., reduced latency) of Spark applications, compared to the existing Spark memory management solutions.**

I. Introduction

Large-scale data processing has been one of the most concerned topics for engineers and scientists in recent years. With the proliferation of availability to the dataset and the need for scalable and containable mega-data processing frameworks, various analytics stacks are becoming prevalent in both industry and research fields. In the past years, processing a massive volume of data has entirely relied on the performance of computing facilities and 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 help users interact with 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], as 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 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 immutable. Based on these two features, Spark builds a lineage of an application to track each stage of computation 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.

[1]This work was partially supported by the National Science Foundation Career Award CNS-1452751.

In this way, Spark avoids the overhead of I/O operations and improves the overall performance. Therefore, in Spark, one of the most crucial factors 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 the 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: ***SMM*** (Static memory manager), and ***UMM*** (Unified memory manager). We 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 long Java garbage collection (GC). Based on the analysis of the defects of the existing memory managers, we design and implement a new memory manager, named ***ATuMm*** (Auto-tuning Memory Manager), to improve Spark performance.

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 2.2.0 and show that our new memory manager significantly improve the Spark performance.

978-1-7281-1025-7/19/$31.00 ©2019 IEEE

- **Analysis of memory usage and GC of Spark memory managers.** We investigate the execution memory usage and garbage collection of three memory managers (i.e., SMM, UMM, and ATuMm). We discover that our ATuMm decreases garbage collection time by preventing overloaded execution memory.

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

II. Motivation

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 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 divides 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, then some of the old RDDs will be evicted according to the LRU (Least Recently Used) algorithm. On the other hand, if execution memory is full, then all intermediate objects that are generated at runtime will be serialized and spilled into the disk to release memory space for subsequent task processing.

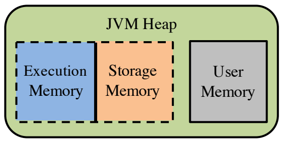

Fig. 1. **Memory Partition of Spark Memory Managers**

A. SMM Study: 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 full setup in Sec. IV-A), with PageRank [5] 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 the 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, it is incapable of caching RDDs that can be reused in following computations and thus cannot save the RDD processing time. If we assign too much space to storage memory, then the confined execution memory pool may trigger 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 achieve the optimal performance automatically.

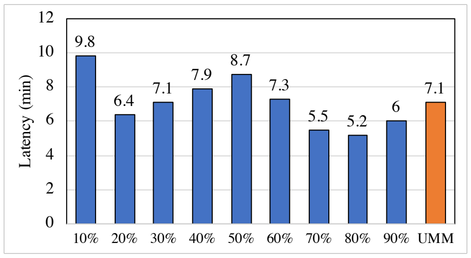

Fig. 2. **Latency of application under SMM and UMM.** SMM increases storage fraction from 10% to 90%.

B. Static VS. Dynamic: Latency Comparison

It is obvious that 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. That is, execution memory can force the storage memory pool to shrink if storage memory exceeds 50% of total accessible memory, even it is fully utilized. Based on this mechanism, UMM guarantees sufficient memory for executing on-time tasks, which avoids the content of execution memory from being spilled into the disk at the greatest extent.

We find that UMM still cannot always 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 storage fraction of 20% and 70%$\sim$90%, and thus not be able to achieve the 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 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 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, which is about 6 times the lowest GC time obtained by SMM with storage fraction of 90%. By combining the results in Fig. 3 and Fig. 2, we note that the GC time has considerable impacts on Spark performance, and UMM's performance degradation results from such long GC time.

We discover that long GCs occur under UMM because UMM expands the execution memory pool aggressively, which may result 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 take both GC time and execution time into consideration for dynamically adjusting memory partition.

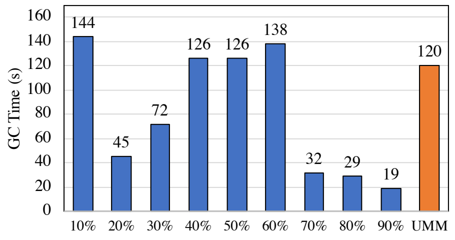

Fig. 3. **GC time comparison.** SMM increases storage fraction from 10% to 90%.

III. Auto Tuning Memory Manager Design

In this section, we present a new Spark memory manager, called ***Auto-Tuning Memory Manager*** (ATuMm), which aims to improve the overall latency for Spark applications by considering both *resource demands* and *GC impact* in dynamic memory resource allocation. Fig. 4 shows the overall block diagram illustrating the interaction of ATuMm with other Spark modules on an "Executor". A Spark cluster often consists of multiple "Executors". Each "Executor" hosts a set of running tasks, and the pools of storage and execution memory are managed independently. In addition, there are two managers in Spark that are responsible to 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.

As shown in Fig. 4, we develop two main components in ATuMm: (1) ***Auto Tuning Algorithm***, and (2) ***Memory Management Algorithm***. The former is called by the "Executor" module to adjust the storage fraction repeatedly and also set the limit (or the maximum allowed) of execution memory. The latter further responds to the memory requirements sent by the "Block Manager" and "Task Memory Manager" modules by considering both free storage/execution memory and the decision made by the "Auto Tuning Algorithm".

Upon the completion of each task, the "Auto Tuning Algorithm" receives the runtime logs of the completed task as well as the previously completed tasks from the "Executor" module. Based on these logs, the algorithm adjusts the storage

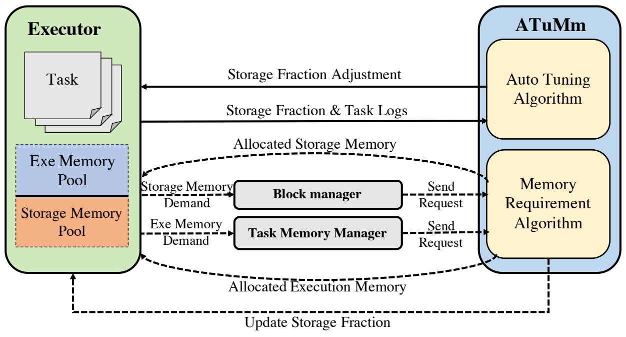

Fig. 4. **ATuMm Architecture**

fraction and then passes it to the "Executor" module for next task execution. The adjustments of storage fraction repeatedly occur until the last task in "Executor" is finished. The "Memory Requirement Algorithm" of ATuMm then reacts to the memory requirements from "Executor" and allocates memory space for RDD cache (i.e., storage memory) and task execution (i.e., execution memory). The storage fraction is also updated by this algorithm based on memory allocations.

A. *Auto Tuning Algorithm*

When a task on the "Executor" module is finished, 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 prior 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. By using these two variables, ATuMm can adjust memory partition with a limit on the maximum memory that can be allocated to execution memory.Alg. 1 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 to avoid 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.

Algorithm 1 Auto Tuning Algorithm.

```
Procedure barChange( GCTime, executionTime)
    curRatio=GCTime/executionTime
    if curRatio=preRatio then
        return None
    else if (curRatio<preRatio and preUpOrDown=true) or (curRatio>preRatio and
    preUpOrDown=false) then
        update preUpOrDown to ture, update preRatio
        return (setUp(step))
    else
        update preUpOrDown to false, update preRatio
        return (setDown(step))
    endif
Procedure setUp(step, preStorageFraction)
    if preStorageFraction+step<100% then
        curStorageFraction=preStorageFraction+step
        if usedStoragePoolSize/totalStoragePoolSize>80% then
            heapStorageMemory=heapStorageMemory+step*accessibleMemory
        endif
    endif
    update preStorageFraction
    return heapStorageMemory, curStorageFraction
Procedure setDown(step, preStorageFraction)
    if preStorageFraction−step>0 then
        curStorageFraction=preStorageFraction−step
        memoryEvict=memoryUsed−curStorageFraction
        if memoryEvict>0 then
            freeStorageMemory(memoryEvict)
        endif
        heapStorageMemory=heapStorageMemory−step*accessibleMemory
        if heapStorageMemory>=curStorageFraction*accessibleMemory then
            heapStorageMemory=curStorageFraction*accessibleMemory
        endif
        update preStorageFraction
    endif
    return heapStorageMemory, curStorageFraction
```

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"), then 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 that of the last task. Otherwise, we move the boundary in the 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 as well 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 Sec. 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 are allowed to 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 already fully utilized. A mechanism is required to determine which objects should be evicted. For storage memory, LRU (Least Recently Used), an existing RDD caching algorithm, is applied by the Spark block manager. 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 of the reasons why 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 details, *setUp()* increases the storage fraction by "step" (see lines 12 and 13 in Alg. 1) 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 Alg. 1). The difference between the storage memory pool size and "heapStorageMemory" will be the potential memory space that can be allocated to execution memory.

Procedure *setDown()* has the same inputs and outputs as *setUp()* to shrink the storage memory pool. In details, *setDown()* decreases the storage fraction by "step" (see line 20 in Alg. 1). However, it needs to consider RDD evictions to release the reduced storage memory additionally (see lines 21 and 22 in Alg. 1). For example, if the current storage memory pool is 5GB with 4.5GB used, and the potential storage memory becomes 4GB, then the memory space ('memoryEvict") that needs to be released is 0.5GB. *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 lines 25 in Alg. 1).

B. Memory Requirement Algorithm

The ***Memory Management Algorithm*** of ATuMm 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. According to the current memory partition and the variable "heapStorageMemory" determined by the "Auto Tuning Algorithm", this algorithm allocates available memory to the two manager modules (i.e., "Block Manager" and "Task Memory Manager") to meet their requirements. Alg. 2 describes the main procedures of this memory management mechanism.

– 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

Algorithm 2 Memory Requirement Algorithm.

```
Procedure acquireExecutionMemory(reqExe)
    extraNeed=reqExe-freeExecutionMemory
    if extraNeed>0 then
        memoryBorrow=min(extraNeeded,storageMemoryPoolSize-
        heapStorageMemory,freeStorageMemory)
        decreaseStoragePoolsize(memoryBorrow)
        increaseExecutionPoolsize(memoryBorrow)
        acquired = executionMemoryPool.acquire(freeExecution+memoryBorrow)
    else
        acquired=executionMemoryPool.acquire($reqExe$)
    endif
    return acquired
Procedure acquireStorageMemory(reqSto)
    memoryToFree=max(0, reqSto−freeStorageMemory)
    if memoryToFree>0 then
        freeStorageMemory(memoryToFree)
    endif
    acquired = storageMemoryPool.acquire(reqSto)
    if heapStorageMemory<usedStorageMemory then
        heapStorageMemory=usedStorageMemory
    endif
    return acquired
```

solid line represents the potential boundary between execution memory and storage memory, and a dashed line represents the value of "heapStorageMemory" that indicates 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, if the required execution memory exceeds the boundary of "heapStorageMemory", then 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 over-allocation 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. By this way, GC time can be reduced as discussed in Sec. 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 new RDDs. In both scenarios 2 and 3, we further update the variable "heapStorageMemory" to be equal to the actual storage memory pool size.

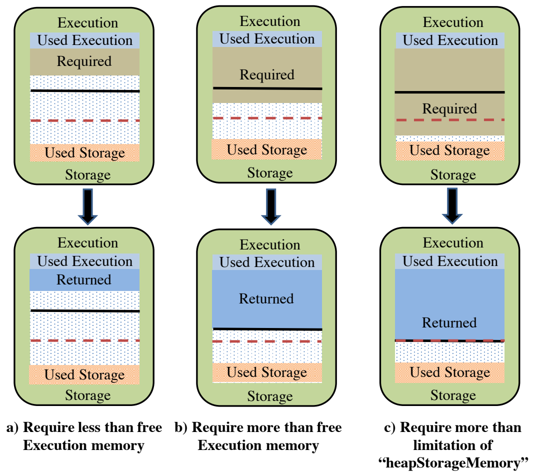
Execution
Used Execution
Required
Used Storage
Storage
Returned
a) Require less than free Execution memory
b) Require more than free Execution memory
c) Require more than limitation of "heapStorageMemory"

Fig. 5. **Execution Requirement Conditions**

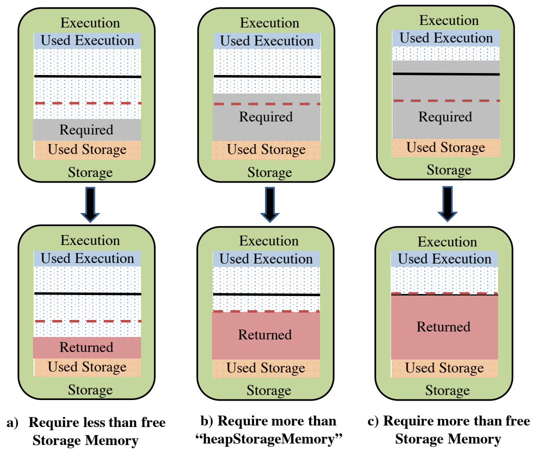
Execution
Used Execution
Required
Used Storage
Storage
Returned
a) Require less than free Storage Memory
b) Require more than "heapStorageMemory"
c) Require more than free Storage Memory

Fig. 6. **Storage Requirement Conditions**

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".

IV. Evaluation

A. Spark Implementation

In this section, we introduce the implementation of our ATuMm in a real Spark cluster and investigate Spark performance under ATuMm. We conduct our evaluation in a Spark cluster with 1 driver and 4 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, $1GB$ memory,

TABLE I
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

and $50GB$ disk space. Table I summarizes the details of our testbed configuration.

We implement our ATuMm as a new portable memory manager module, besides SMM and UMM, in Apache Spark 2.2.0, which contains functions interacting with other Spark modules. It's 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 (https://github.com/DanlinJia/spark_core_ATMM). Specifically, we develop functions *acquireStorageMemory()* and *acquireExecutionMemory()* to allocate storage and execution memory to "Block Manager" and "Task Memory Manager". We also integrate a profile collector in the "Executor" module to collect task logs. Then, function *barChange()* receives these task logs and calls functions *increaseStorageFraction()* or *decreaseStorageFraction()* to adjust memory partition. Furthermore, we integrate a memory usage analyzer in ATuMm to collect the runtime memory usage information. Users can replace the existing Spark memory manager to ATuMm by simply setting a configurable parameter before submitting a Spark application.

In our experiments, we set the accessible memory and the initial storage fraction of ATuMm as 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. And the window size representing the number of previous tasks is set as 20% of activated tasks by default. Users are able to preconfigure these parameters in ATuMm before launching any Spark applications.

B. Results

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 [5], [6]. Considering the duration of experiments, we report results for a workload of $1GB$ input data for applications.

Fig. 7 (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. 7-(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. 7-(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.

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 $1GB$, $2GB$, $3GB$ and $7GB$. As shown in Fig. 7-(c), ATuMm achieves the best performance when the input data sizes are $1GB$, $2GB$, and $3GB$. 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 $7GB$, 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 $7GB$ input data), which is better than that of SMM.

C. 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. 8, where PageRank is running with $3GB$ input data. Fig. 8-(a)$\sim$(c) present the storage memory usage across time under the three memory managers, while Fig. 8-(d)$\sim$(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. 8-(a)$\sim$(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. 8-(a), the static storage memory pool starts to evict RDDs when the utilization of the storage memory pool is full. However, in Fig. 8-(b), UMM drops the size of its storage memory pool to almost zero and then increase its

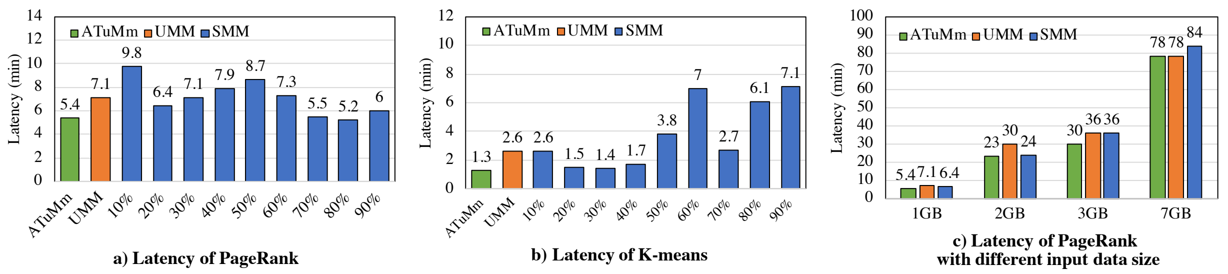

Fig. 7. **Execution Time of Applications Under SMM, UMM and ATuMm**

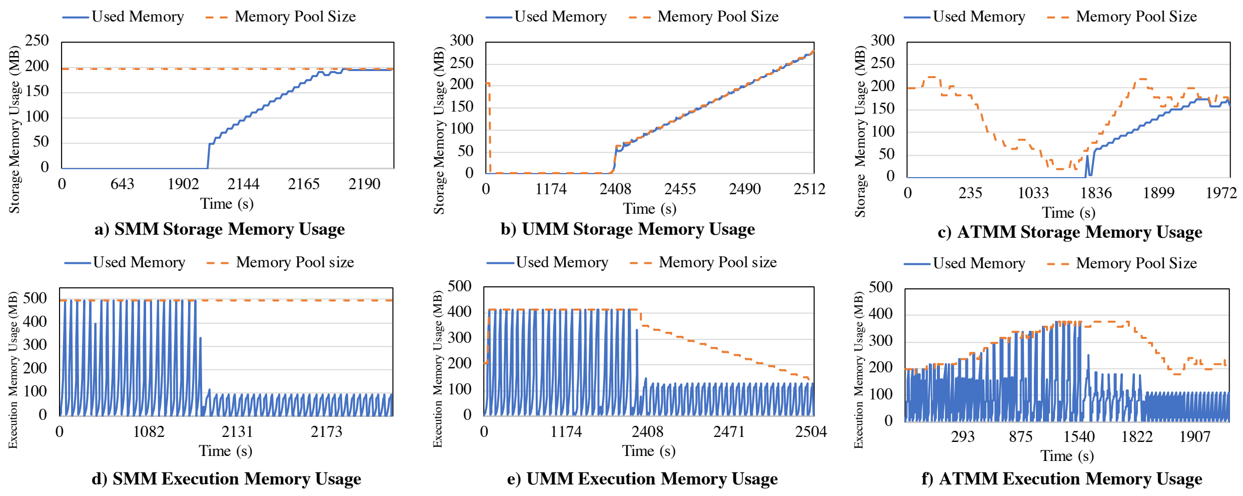

Fig. 8. **Memory Usage Analysis of SMM, UMM and ATuMm**

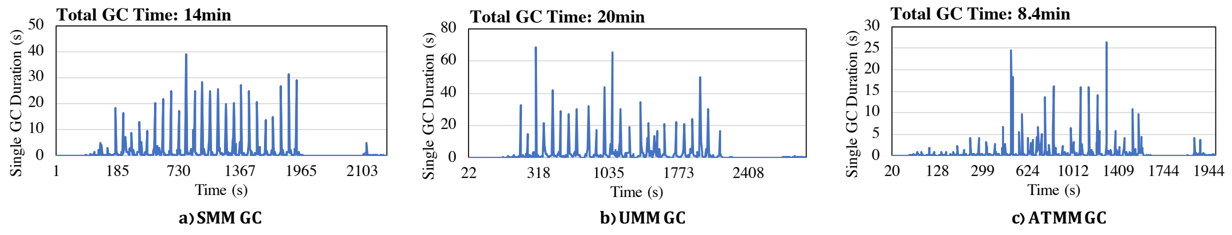

Fig. 9. **GC Analysis of SMM, UMM and ATuMm**

storage pool when RDDs are cached. The storage memory pool changes more dynamically under ATuMM, as shown in Fig. 8-(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. 8-(d)∼(f). SMM fixes the execution memory pool size regardless of workload diversity, while UMM and ATuMm alter the execution memory pool size based on demands. Fig. 8-(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. 8-(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 $200MB$, 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 $3GB$ input data as representative and compare GC time using three memory managers. Fig. 9 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 [7]. Fig. 9-(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. 9-(b). More importantly, we can observe that the full GCs under ATuMm are all below 30 seconds in Fig. 9-(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 $14min$, $20min$ and $8.4min$, respectively. 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).

V. Related Work

Deducting latency of Spark applications has been a concerned topic since Spark was proposed. Several works have been published, which focus on improving spark performance by optimizing cache algorithm [8], [9] and task scheduling [10] However, there does not exist any work on the new Spark memory manager design. However, these works help us obtain a deep understanding of how memory requirement affects Spark performance, which in turn provides the perspective to design our memory manager.

MEMTUNE [11] 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 requirement if tasks are storage-sensitive. Another work Sparkle [12] designs and implements new Spark modules to optimize Spark performance on a scale-up machine. It builds a shared memory pool for each executor, which saves the shuffle time between nodes of the Spark cluster and achieves a performance improvement. This work leverages the shared memory pool that allows each executor in the Spark cluster to be aware of the location of each RDD partition, which decreases the overhead of fetching data among each other.

Dynamic JVM heap management [13] enables multiple JVMs to run on the host simultaneously with reduced contention and analyzes the impact of JVM configuration and GC on Spark performance. We leverage the knowledge and experiments in these works to help design our algorithm. Also, the Spark open source community provides suggestions on Spark memory tuning and GC tuning for users. However, these tuning suggestions need users to explore the characteristics of applications and configure workloads based on their Spark running environment. We get perspectives from these suggestions and propose the new Spark memory manager to tune memory allocation automatically.

VI. 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 design a new Spark memory manager (ATuMm) that leverages the feedback of GC time and memory demands to partition the memory pool dynamically. We implement ATuMm in Spark 2.2.0 and construct experiments in a real Spark cluster. We find that our ATuMm obtains around 25% improvement of Spark performance, compared with existing memory managers. We also find that one of the primary sources of performance improvement lies in the reduction in GC time.

Currently, our auto tuning algorithm only tracks one of the local minimum storage fractions that can provide better performances than the other two existing memory managers. In the future, we plan to propose new algorithms to find optimal performance globally. Besides, we plan to scale up the Spark cluster and investigate ATuMm performance for more diverse applications.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

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

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," ser. HotCloud'10. USENIX Association, 2010, pp. 10–10.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," ser. NSDI'12, 2012, pp. 2–2.

[5] M. J. F. I. S. Reynold S. Xin, Joseph E. Gonzalez, "Graphx: A resilient distributed graph system on spark," AMPLab, EECS, UC Berkeley, Jun 23 2013.

[6] U. R. Raval and C. Jani, "Implementing improvisation of k-means clustering algorithm," 2016.

[7] R. Xin and J. Rosen, "Tuning java garbage collection for apache spark applications," https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html.

[8] T. B. G. Perez, X. Zhou, and D. Cheng, "Reference-distance eviction and prefetching for cache management in spark," ser. ICPP 2018. ACM, 2018, pp. 88:1–88:10.

[9] L.-Y. Ho, J.-J. Wu, P. Liu, C.-C. Shih, C.-C. Huang, and C.-W. Huang, "Efficient cache update for in-memory cluster computing with spark," ser. CCGrid '17. IEEE Press, 2017, pp. 21–30.

[10] V. S. Marco, B. Taylor, B. Porter, and Z. Wang, "Improving spark application throughput via memory aware task co-location: A mixture of experts approach," ser. Middleware '17. ACM, 2017, pp. 95–108.

[11] 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," May 2016, pp. 383–392.

[12] M. Kim, J. Li, H. Volos, M. Marwah, A. Ulanov, K. Keeton, J. Tucek, L. Cherkasova, L. Xu, and P. Fernando, "Sparkle: Optimizing spark for large memory machines and analytics," ser. SoCC '17.

[13] S. Sahin, W. Cao, Q. Zhang, and L. Liu, "Jvm configuration management and its performance impact for big data applications," *2016 IEEE International Congress on Big Data (BigData Congress)*, pp. 410–417, 2016.