Computers, Materials & Continua
DOI:10.32604/cmc.2020.012513
Article

SMConf: One-Size-Fit-Bunch, Automated Memory Capacity Configuration for In-Memory Data Analytic Platform

Yi Liang[1,*], Shaokang Zeng[1], Xiaoxian Xu[2], Shilu Chang[1] and Xing Su[1]

[1]Faculty of Information Technology, Beijing University of Technology, Beijing, 100124, China
[2]Department of Informatics, University of Zurich, Zurich, CH-8050, Switzerland
[*]Corresponding Author: Yi Liang. Email: yliang@bjut.edu.cn
Received: 02 July 2020; Accepted: 31 July 2020

Abstract: Spark is the most popular in-memory processing framework for big data analytics. Memory is the crucial resource for workloads to achieve performance acceleration on Spark. The extant memory capacity configuration approach in Spark is to statically configure the memory capacity for workloads based on user's specifications. However, without the deep knowledge of the workload's system-level characteristics, users in practice often conservatively overestimate the memory utilizations of their workloads and require resource manager to grant more memory share than that they actually need, which leads to the severe waste of memory resources. To address the above issue, SMConf, an automated memory capacity configuration solution for in-memory computing workloads in Spark is proposed. SMConf is designed based on the observation that, though there is not one-size-fit-all proper configuration, the one-size-fit-bunch configuration can be found for in-memory computing workloads. SMConf classifies typical Spark workloads into categories based on metrics across layers of Spark system stack. For each workload category, an individual memory requirement model is learned from the workload's input data size and the strong-correlated configuration parameters. For an ad-hoc workload, SMConf matches its memory requirement signature to one of the workload categories with small-sized input data and determines its proper memory capacity configuration with the corresponding memory requirement model. Experimental results demonstrate that, compared to the conservative default configuration, SMConf can reduce the memory resource provision to Spark workloads by up to 69% with the slight performance degradation, and reduce the average turnaround time of Spark workloads by up to 55% in the multi-tenant environments.

Keywords: Spark; memory capacity; automated configuration

1 Introduction

In-memory distributed processing frameworks are widely adopted as the new engines for big data analytics [1,2]. The success of these frameworks is due to their ability to keep the reused data in the memory during the data processing, which avoids the large amount of disk I/O and achieves the performance acceleration. Spark is the most popular in-memory processing framework and has been adopted as the enabling infrastructure of a comprehensive ecosystem of various data analytics, including machine learning, graph computing and SQL query. It has been proved that iterative data analytics can achieve more than ten times performance enhancement on Spark, compared to the disk-based Hadoop framework [3,4].

Memory is the key resource to enable such performance enhancement. The memory capacity configuration is non-trivial for workloads in in-memory processing frameworks [5]. Insufficient memory capacity configuration leads to the extra disk I/O and re-computation, and hence, significant performance degradation. The current approach adopted in Spark is to statically configure the memory capacity based on user's specifications [6,7]. However, this requires users to have profound understanding of their workloads and the system stack. Even for expert users, determining the proper configuration is hard when they employ a prepackaged analytics application and have no enough knowledge of their system-level characteristics. To guarantee the performance enhancement, users in practice often adopt the conservative configuration recommended by Apache Spark Community, which is the upper-bound memory requirement of state-of-practice workloads, and require the resource manager to grant more memory share than that they actually need [8]. This results in the situation that, according to the resource manager's bookkeeping, all memory resources have been allocated and no more workloads can be admitted whereas a large amount of memory resources remain unused in the platform. For example, the real trace from Google's product platform reveals that most of cluster memory resources was allocated to data analytic workloads while around 50% of them are unused [9].

Automated configuration is an idea way to relieve users from the burden of memory capacity configuration, which guarantees both the workload performance and the memory utilization improvement. Though promising, this is a challenging task due to two reasons. First, as the diversity of workloads, there is not one-size-fit-all proper configuration. In our paper, the ***proper memory capacity configuration*** is determined by the minimal amount of memory that required by Spark workload to execute without any errors, data processing exceptions and data re-computations. We have observed that, among the typical in-memory computing workloads with the same input data size, such proper configurations are significantly different and vary up to 5.2 times on Spark. On the other hand, the case-by-case configuration solution will lead to high TCO (Total Cost of Ownership), especially for platforms that always face to ad-hoc workloads [10]. Second, it is difficult to decide the configuration parameters that have strong impacts on the memory capacity configuration. In-memory processing frameworks have a large amount of configuration parameters, e.g., Spark has more than 180 parameters, which have different impacts on the memory capacity configuration [11]. Meanwhile, memory configurations of various workloads may have different sensitivities to the setting of a specific parameter. Hence, such strong-impacted parameters vary across workloads.

In this paper, we focus on Spark and propose SMConf, an automated memory capacity configuration solution for in-memory computing workloads. SMConf is proposed based on our observation that, though there is not one-size-fit-all proper configuration, the one-size-fit-bunch configuration can be found for in-memory computing workloads on the given hardware and system software infrastructures. SMConf first chooses metrics from Spark system stack, that can represent the memory requirement characteristics of in-memory computing workloads, as features and classifies the typical Spark workloads into categories with these features, and then selects the strong-correlated configuration parameters and learns the memory requirement model for each workload category. For an ad-hoc workload, SMConf matches its memory requirement features to one of the workload categories and determines its proper memory capacity configuration with the corresponding memory requirement model. Our main contributions can be summarized as follows.

1. Through empirical study, we have two findings. One is that, though Spark workloads are diverse, they can be classified into categories and workloads in each category have similar memory requirement characteristics. The other is that the variations of the Spark parameter setting have different correlations to the memory requirement across Spark workloads.
2. We design a Spark workload classification method in SMConf from the system stack perspective. In this method, the classification features are selected from the metrics across layers of Spark system stack, including hardware layer, operating system layer and runtime layer. The K-medoid clustering technique is adopted to conduct the Spark workload classification and the distance measure is determined based on value variations of the selected features among workloads.
3. We design the memory requirement modelling method in SMConf. In this method, for each Spark workload category, the strong-correlated configuration parameters are first selected by the stepwise-regression technique, and then an individual memory requirement model is learned with the input data size and the selected configuration parameters as features using the support vector machine (SVM) regression technique.
4. We evaluate SMConf on Spark with representative workloads from SparkBench suite. The experimental results demonstrate that, for ad-hoc workloads, the memory capacity configurations determined by SMConf are close to the proper configurations. Compared to the conservative default configuration, SMConf can reduce the memory resource provision to Spark workloads by up to 69% with slight performance degradation. In the multi-tenant environment, with SMConf, the average turnaround time of Spark workloads can be reduced by up to 55%.

The rest of the paper is organized as follows. In Section 2, we introduce the background and motivation. Section 3 describes the overview of SMConf. Section 4 gives the details of the Spark workload classification method in SMConf. Section 5 presents the memory requirement modelling in SMConf. Section 6 presents the detailed performance evaluation of SMConf. Section 7 describes the related works. We draw the conclusion in Section 8.

2 Background and Motivations

In this section, we first describe the memory management in Spark. Then, we discuss the motivation of our proposed memory capacity configuration solution through empirical study on Spark workloads.

2.1 Memory Management in Spark

In Spark, data is managed as the memory abstraction called resilient distributed datasets (RDDs) [3]. An RDD is composed of several data blocks (partitions) which are distributed across computing nodes in Spark cluster. Data analytic workload in Spark is executed in form of batch job, which conducts a serial of RDD transformations and actions, and can be organized as a DAG (Direct Acyclic Graph) of RDDs. Once an RDD block is lost, it can be recomputed based on the dependencies among RDDs. Fig. 1 demonstrates the overview of Spark.

As shown in Fig. 1(a), the Spark job is executed in form of parallel tasks on several executors with a driver program controlling them on a master node. An executor is launched as a JAVA process on a worker node, which is the basic executing container in Spark. Each executor allocates its own heap memory space as Spark memory space to hold the cached RDDs, shuffle data and temporary data generated during the Spark workload execution. ***Spark memory space*** of an executor is organized as Fig. 1(b), which is composed of three regions, Reserved memory, Spark memory and User memory. Among these regions, Spark memory is the main region of Spark memory space and managed by Apache Spark. Spark memory can be further split into two regions, Storage memory and Execution memory. Storage memory is mainly used for both RDD caching and for temporary space serialized data “unroll”.

Execution memory is used for storing the objects required during the execution of Spark tasks, particularly the shuffle intermediate buffering. In the latest Spark memory management model, the boundary between Storage memory and Execution memory can be moved in case of memory pressure. In addition, Reserved memory is reserved by the system with fixed size, and User memory is used to store temporary data needed for RDD transformations.

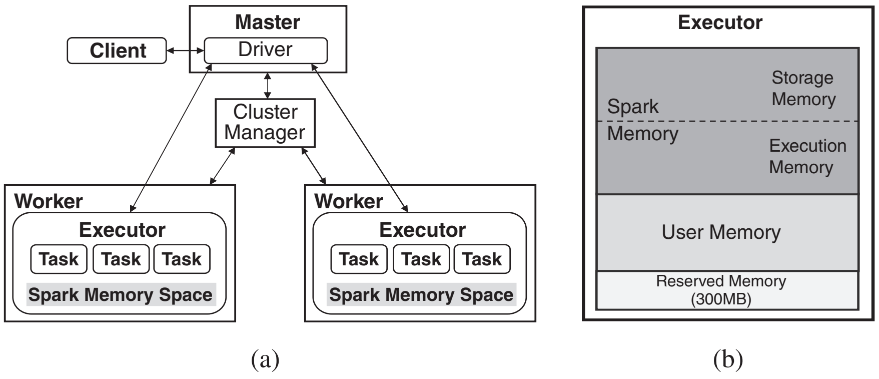


Figure 1: The overview of Spark. (a) Architecture. (b) Spark memory space

In Spark, the memory capacity configuration refers to the size setting of the Spark memory space in one executor. Executors involved in one Spark workload execution are configured with the same memory space size. The space size can be configured with the parameter of *spark.executor.memory*.

2.2 *Empirical Study*

Our proposed solution is based on the assumption that, though there is not one-size-fit-all proper configuration, the one-size-fit-bunch configurations can be found for in-memory computing workloads in Spark. As a result, only one single performance model needs to be learned for workloads with the similar memory requirement patterns. To verify this assumption, we conduct an empirical study by using SparkBench, a comprehensive benchmark suite for Spark [12]. We use Spark version 2.3.4, with Hadoop version 2.7 as the storage layer. All experiments are done on a Spark cluster with 3 nodes. Each node is equipped with one 4-core 3.3 GHz Intel Core i5-6600 processors and 16 GB memory. All nodes are interconnected with a 1 Gbps Ethernet. One of the nodes is configured as the master node and others as the worker nodes. On each worker node, one executor can be launched. We repeat each experiment for five times and take the average results.

2.2.1 Memory Requirement Variations among Workloads

In this part, we choose eight representative SparkBench workloads: LogisticRegression (LR), K-Means (KM), TeraSort (TS), ShortestPath (SP), PageRank (PR), SVM, PCA and SVD++, to demonstrate the memory requirement variations among Spark workloads. We run these workloads on the established Spark cluster in turn. For each running, the workload is allocated with all CPU cores on worker nodes. Each workload is executed with input data size as 600 MB, 1 GB, 1.4 GB, 1.8 GB, 2.2 GB, and 2.6 GB, respectively. Due to the complexity of Spark memory management model, the memory requirement of Spark workload is captured by fine-grained configuration tuning. First, the memory capacity configuration for each executor of a Spark workload is set to 12GB, which is the conservative default configuration recommended by Apache Spark Community [7]. Then, the memory capacity configuration is decreased gradually with the step of 100MB to find the minimum configuration that satisfies no exceptions, errors and data re-computations occur during Spark workload execution. We take such minimal configuration as ***the memory requirement of the Spark workload.***

Fig. 2 demonstrates that, with the same input data size, the memory requirements are quite various among Spark workloads. The requirements gap is 5.2 times by maximum. For example, when the input data size is 600 MB, the memory requirement of SVD++ workload is 3.42 GB, while that of K-Means is only 675 MB. Note that the Spark memory space mainly holds the cached RDD, the intermediate shuffle data and other temporary computation data. One reason for the memory requirements variations is that the different operation characteristics and data access patterns leads to the peak size of shuffle data and temporary data varied among workloads. For example, when the input data size is 600 MB□the intermediate shuffle data size ranges from 50 MB to 650 MB. The other reason is that the RDD caching demands are different. When the input data size is 600 MB, the cached RDD size of TeraSort workload is 50 MB, while that of PageRank workload is over 500 MB.

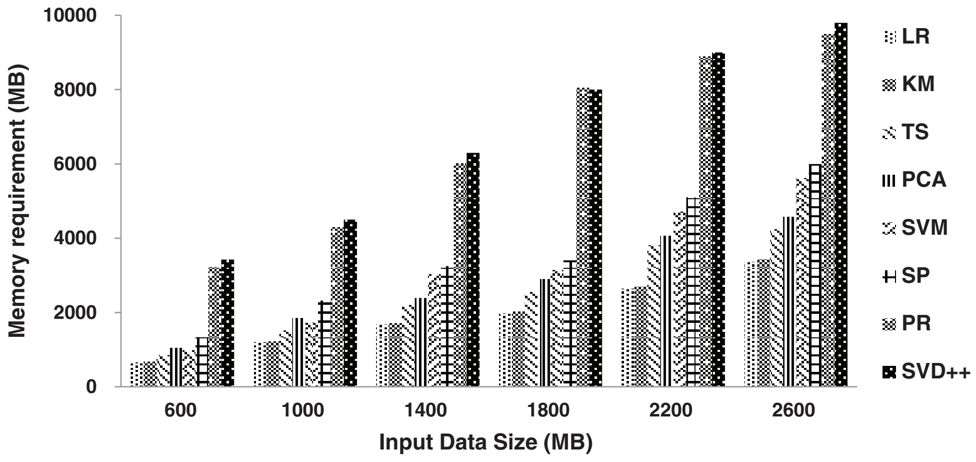


Figure 2: Memory requirements under different input data sizes

2.2.2 Workload Classification on Memory Requirements

Restudying the results in Fig. 2, it can be found that these Spark workloads can be classified into several groups. The memory requirements in each group are similar, while those of different groups are quite different. For example, when the input data size is 600 MB, the workloads can fall into five groups on the memory requirement: K-Means and LogisticRegression in [660 MB, 675 MB], Terasort in [850 MB, 850 MB], SVM and PCA in [982 MB, 1.05 GB], ShortestPath in [1.33 GB, 1.33 GB], PageRank and SVD++ in [3.22 GB, 3.42 GB].

We further investigate the workload classification under different input data size. For each input data size case, we set the Relative Average Deviation (RAD) of the workload's memory requirement less than 0.04 in a group, and greater than 0.15 among groups. As Tab. 1 shown, the classifications of Spark workloads keep relatively stable. When the input data size is greater than 1.4 GB, the workload classifications results are same. Even in 600 MB and 1 GB cases, K-Means, LogisticRegression, PageRank and SVD++ have the same classification results as those in the larger input data size cases. To unify the classifications in all cases, each of Terasort, PCA, SVM and ShortestPath workloads can be classified into one group independently. In conclusion, we have the assumption that the Spark workloads can be classified on the memory requirement and features that determine the classification should be explored.

Table 1: Example of Spark workload classification

	600 MB		1 GB		1.4 GB		1.8 GB		2.2 GB		2.6 GB	
Category	Workloads	RAD	Workloads	RAD	Workloads	RAD	Workloads	RAD	Workloads	RAD	Workloads	RAD
#1	LR, KM	0.01	LR, KM	0.01	LR, KM	0.04	LR, KM	0.01	LR, KM	0.01	LR, KM	0.04
#2	TS	0.00	TS	0.00	TS, PCA	0.04	TS, PCA	0.04	TS, PCA	0.03	TS, PCA	0.04
#3	PCA, SVM	0.03	PCA, SVM	0.03	SVM, SP	0.03	SVM, SP	0.04	SVM, SP	0.04	SVM, SP	0.03
#4	SP	0.00	SP	0.00	PR,SVD++	0.02	PR,SVD++	0.00	PR,SVD++	0.01	PR,SVD++	0.02
#5	PR,SVD++	0.03	PR,SVD++	0.02								

2.2.3 *Impact of Parameter Settings on Memory Requirements*

There are over 180 parameters in Spark that can be configured. Besides the input data size, the different parameter settings will impact the memory requirement of workloads on a given hardware and software infrastructure. To quantify it, we select one workload from each of the classified categories in Tab. 1, where the input data size is 600 MB. We conduct the experiment by varying the setting of the number of cores allocated to each executor (*spark.executor.cores*). For each workload, we take the memory requirement when *spark.executor.cores* is set to 1 as the baseline and demonstrate the normalized memory requirement in all test cases. The experimental results are shown in Fig. 3.

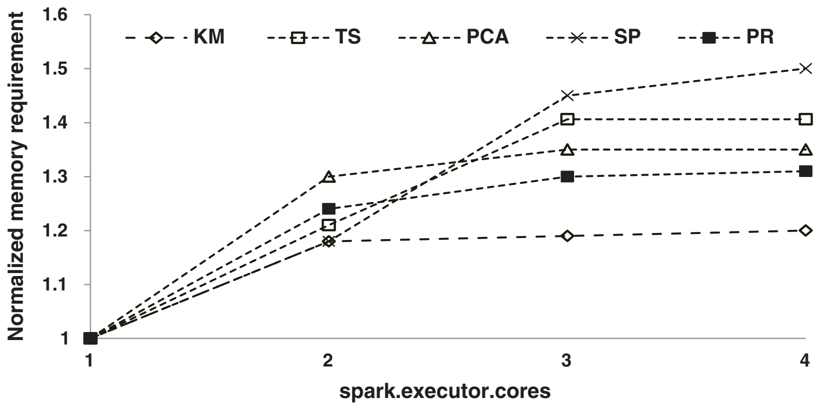


Figure 3: Memory requirement under different parameter settings

We have two observations from the experimental results. First, different workloads have different sensitivities to the setting of a specific parameter. For example, with the setting of *spark.executor.cores* increasing from 2 to 4, the memory requirement of ShortestPath increases monotonically, but the memory requirement of K-Means almost keeps stable. Second, the variation trends of the memory requirement are various among workloads. These observations inspire us that, for each workload category, an individual memory requirement model should be learned from features of different strong-correlated configuration parameters.

3 Overview of SMConf

SMConf is a one-size-fit-bunch, automated solution that can help to achieve the proper memory capacity configuration for diverse Spark workloads. Fig. 4 demonstrates the framework of SMConf.

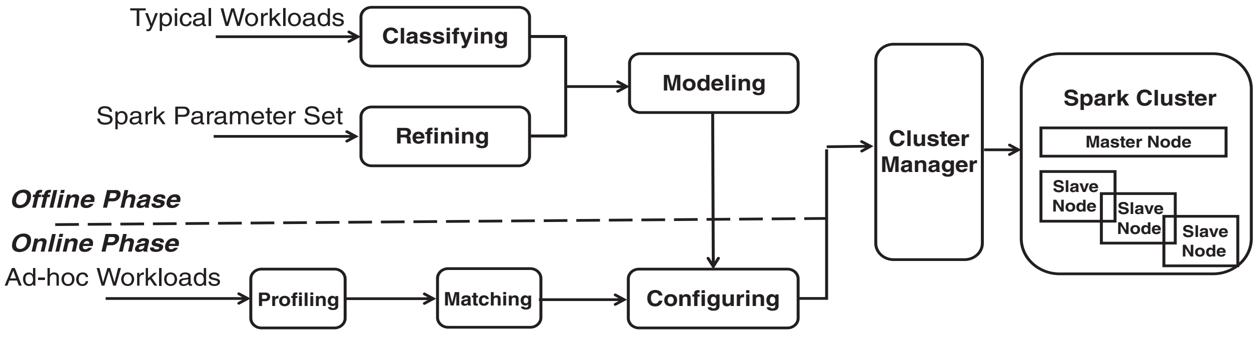


Figure 4: Framework of SMConf

The solution works in two phases: Offline phase and online phase. In the offline phase, first, typical workloads are chosen from Spark benchmarks and features that represent Spark workload's memory requirement are extracted from the system stack's perspective; With the extracted features, the typical workloads are classified into categories by using the clustering technique; second, the parameters that have strong correlations with memory requirements of Spark workloads are refined; finally, for each workload category, an individual memory requirement model is learned from features of strong-correlated Spark parameters and the input data sizes. In the online phase, the incoming ad-hoc workload can be profiled with small-sized input data and matched to one of the typical workload categories; and then its proper memory capacity configuration is automated determined by the corresponding memory requirement model.

4 Workload Classification in SMConf

To classify Spark workloads in SMConf, we first select the key metrics from all layers of Spark system stack as the classification features, and then these features are refined and used to classify typical Spark workloads by the K-medoid clustering method.

4.1 Key Features Selection

To classify Spark workloads accurately, the classification features should be representative of Spark workload's memory requirements. Specifically, such features should be stable to a specific Spark workload under various input data size and parameter settings, and significantly different among workloads with diverse memory utilization characteristics. ***Working set*** refers to the amount of memory that a workload requires in a given time interval, which is the intrinsic data access characteristics of Spark workloads and leads to the difference of memory utilizations among workloads. However, to measure the working set of a workload is highly time-consuming [13]. On the other hand, Spark's system stack is mainly composed of the runtime layer, the operating system layer and the hardware layer [14]. Each layer has its individual mechanism whose performance metrics can reflect the working set of a Spark workload, e.g., cache mechanism in hardware layer, page management mechanism in operating system and Java Virtual Machine (JVM) Garbage Collection mechanism in runtime layer. Hence, we try to find such metrics across Spark system stack as the classification features.

The candidate selected metrics are shown in Tab. 2. The candidate metrics covers all layers of Spark system stack and represent the working set of Spark workload to some extent.

In the hardware layer, we choose metrics of the workload's cache access and TLB access. This is because that both the hit/miss of the Cache and TLB can reflect the working set of the workload directly [15]. The metrics of L2_HIT, L2_MISS, L3_MISS are chosen for the cache access and the metrics of

DT_MISS is chosen for TLB access. In addition, we choose the metrics of IPC and LSR. Both metrics can represent the workload's data access frequency.

Table 2: Candidate metrics for Spark workload classification

Metric Name	Layer	Description
GCSD	Runtime Layer	Standard deviation of Garbage Collection frequency
GCCV	Runtime Layer	Coefficient of variation of Garbage Collection frequency
PFS	OS Layer	Page faults per second
CSS	OS Layer	Context switches per second
L2_HIT	Hardware Layer	L2 Cache hit ratio
L2_MISS	Hardware Layer	L2 Cache misses per 1000 instructions
L3_HIT	Hardware Layer	L3 Cache hit ratio
L3_MISS	Hardware Layer	L3 Cache misses per 1000 instructions
DT_MISS	Hardware Layer	Data TLB misses per 1000 instructions
LSR	Hardware Layer	Load and store ratio
IPC	Hardware Layer	Instructions per cycle

In the operating system layer, we choose two metrics. The first metric is PFS, which refers to the page faults per second (PFS). As the page management is the basic operating system memory management mechanism, a page fault occurs when a process attempts to access a block of data that does not reside in the physical memory. Hence, the amount of page faults occurring during the workload execution can reflect the working set size of the workload. The second metric is CSS, which refers to the context switches per second. Generally, the context switch may be caused by a workload's I/O wait, which possibly implies that the workload has a large working set and should read data from disk.

In the runtime layer, Spark launches JVM-based executor to host the workload executions. Garbage Collection is the important memory management mechanism of JVM [3]. Garbage collection frequency can usually reflect the JVM memory utilization of a workload. The larger working set may lead to the more frequent JVM garbage collection. On the other hand, Spark workloads are executed in multiple stages and a workload's memory utilization pattern may vary among those stages. Hence, we adopt two metrics in the runtime layer, GCSD and GCCV, to represent the standard deviation and coefficient of variation of garbage collection frequency among stages of a Spark workload execution, respectively.

To evaluate the representativeness of the candidate metrics to Spark workload's memory requirement, we adopt the Pearson Correlation Coefficient (PCC). PCC is a measure of the strength of the association between two variables on the same interval or ratio scale. For two variables ***X*** and ***Y***, their PCC can be represented as follows:

$$r_p = \frac{N\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{N\sum x_i - \left(\sum x_i\right)^2}\sqrt{N\sum x_i - \left(\sum y_i\right)^2}} \tag{1}$$

where, N is the amount of sample data, and x_i and y_i are the i^{th} sample data of ***X*** and ***Y***. The range of r_p is between 1 and –1, the large absolute value of r_p implies the strong correlation of the two variables [15].

For each typical Spark workload, we conduct experiments by varying the input data sizes and configuration parameters. We collect and normalize data on the workload's memory requirement and the candidate metrics in all experiments to generate the sample data set. For each candidate metric, we calculate its PCC to the workload's memory requirement with the generated sample data set. Candidate metrics, whose PCCs are greater than 0.4 and the corresponding p-values are less than 0.05, are selected as the classification features, to guarantee them to have significant correlations with Spark workloads' memory requirement [16]. As an example, we apply the feature selection method to the eight typical Spark workloads from SparkBench described in Section 2. The final selection results are significantly refined to six metrics (that is, GCSD, PFS, L2_MISS, L3_MISS, LSR and IPC) and cover all layers on Spark system stack.

4.2 Spark Workload Clustering

Based on the selected features, we apply the clustering technique to the Spark workload classification. In our solution, the centroids generated in the Spark workload clustering would be used to determine the workload category matching of ad-hoc workloads. To guarantee the accurate workload category matching, such centroids should be most representative of the actual values of classification features in their clustering groups. Hence, we adopt K-medoids clustering technique. K-medoids clustering is a robust alternative of K-means clustering. It uses the actual data point to represent the centroid of the cluster, which is the most centrally located object of the cluster and has minimum sum of distances to other points, and hence, is less sensitive to noises and outliers [17].

We conduct experiments by varying the input data sizes and configuration parameters on each typical Spark workloads. We collect and normalize data on the selected features in each experiment as the sample data set. The sample data are then grouped into k clusters with the K-medoids algorithm. Each cluster represents an individual Spark workload category. To accurately classify Spark workloads, the distance measure in K-medoids clustering needs to be clarified. The distance measure should be sensitive to the difference in sample data among workloads with different memory utilizations and satisfies that sample data of a specific workload should fall into one cluster with high possibility. In our solution, the distance measure is chosen from the three most common used distance metrics: Manhattan Distance, Chebyshev Distance and Euclidean Distance [18,19]. We choose the distance metric based on the characteristics of feature value variations among workloads. We choose three workloads from SparkBench: LogisticRegression (LR), K-Means (KM) and ShortestPath (SP), and the features from the example in Section 4.1. We calculate the mean value and coefficient of variation of features in the sample data from the three workloads. It is shown that, for each feature, the difference of mean values is relatively small, which is less than 0.02, among three workloads. However, as described in Section 2, LogisticRegression and K-Means have similar memory requirements and ShortestPath has the different one. We further investigate the coefficient of variation on each feature in the sample data from every two workloads. We found that all coefficient of variations are less than 0.05. The results indicate that the distance measure should amplify the outliers to differentiate the sample data from workloads with various memory utilizations. Hence, we choose the Euclidean distance as our distance measure. Results in Fig. 5 verify our selection.

With the Manhattan distance and Chebyshev distance, the average distance between K-Means and ShortestPath are almost the same as that between LogisticRegression and K-Means. With the Euclidean distance, those distances are significantly different, which is in line with the workload classification on memory requirement.

5 Memory Requirement Modelling in SMConf

Observations in Section 2 give the hint that the memory requirement of Spark workloads is impacted by both the input data size and the settings of configuration parameters on the given hardware and system software infrastructures. The framework of the memory requirement modelling method in SMConf can be described as follows. Among the large-amounted Spark configuration parameters, the candidate parameters, that have impacts on Spark workload's memory requirement, are first selected. Then, for each Spark workload category, different strong-correlated parameters are refined from the candidate parameter set. Finally, an individual model is learned for each workload category, to estimate the memory requirement under different input data sizes and those strong-correlated parameter settings.

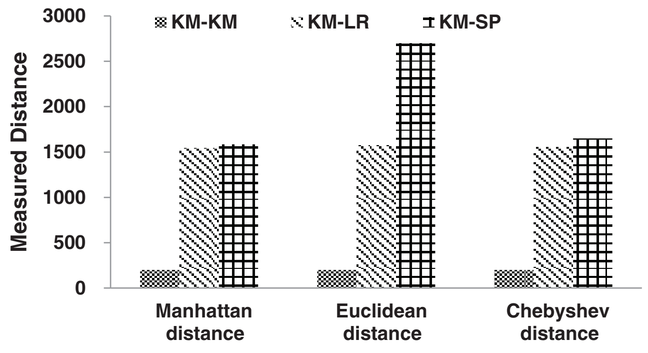


Figure 5: Comparison of different candidate distance metrics

5.1 Configuration Parameter Selection

To reduce the complexity of memory requirement modelling on the over 180 Spark configuration parameters, selecting the strong-correlated parameters is necessary. We first select the candidate parameters based on the data access operations during the Spark job execution, shown in Fig. 6.

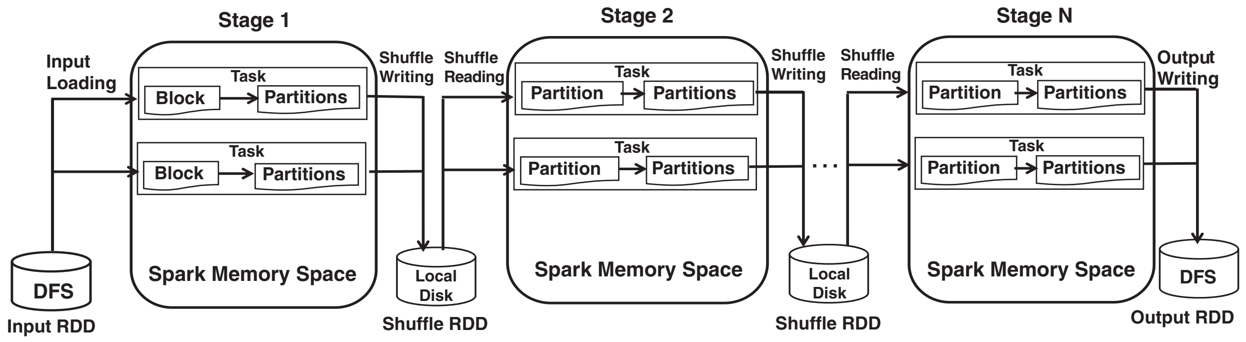


Figure 6: Data access operations in Spark job execution

As described in Section 2.1, the Spark job can be expressed as a Directed Acyclic Graph (DAG) and launched in several JVM-based executors. The memory space of a Spark job is allocated from the heap memory spaces of its residing executors. The job DAG is composed of multiple stages, which are executed in sequence. Each job stage can be mapped into a set of tasks running in the residing executors in parallel. Besides RDD caching, there are mainly three data access operations that have impacts on the Spark workload's memory requirement: Input Loading, Shuffle Writing and Shuffle Reading. The Input Loading operations are conducted in the first stage when each task reads a data block of the input RDD from the distributed file system into the Spark memory space. In Spark, the output data of a stage are called shuffle RDD. The Shuffle Writing operations are conducted in each stage, when the generated shuffle RDD is written into the buffer in Spark memory space and flushed to the local disk on the RDD size exceeding the threshold. The Shuffle Reading operations are conducted in all follow-up stages when any task reads a partition of the shuffle RDD generated by its ascending stage into Spark memory space to process.

Tab. 3 lists the main configuration parameters that concern the three data access operations in Spark. The parameters of *Spark.executor.cores*, *dfs.block.size* and *spark.default.parallelism* determine the total size of the loading/reading data of concurrent tasks in an executor in the first stage or follow-up stages. As the input and shuffle RDDs can be stored in Spark memory space in the compressed mode, we choose two parameters of RDD compressions, *Spark.shuffle.compress* and *Spark.rdd.compress*. The parameters of *Spark.maxRemoteBlockSizeFetchToMem* and *Spark.shuffle.file.buffer* have impact on the memory requirement in the Shuffle Reading and Shuffle Writing operation, respectively. Finally, the parameter of *Spark.memory.fraction* and *spark.storage.memoryFraction* determine the maximum available Spark memory space for execution and storage in an executor with a given heap memory size and the space for cached RDD that is immune to eviction, respectively.

Table 3: Candidate parameters for memory requirement modelling

Parameters	Descriptions
spark.executor.cores	CPU cores assigned to a Spark executor
dfs.block.size	The block size of the input file
spark.default.parallelism	Maximum number of partitions in Shuffle RDD
spark.shuffle.compress	Type of Shuffle RDD compression
spark.rdd.compress	Type of serialized RDD partitions compression
spark.maxRemoteBlockSizeFetchToMem	Threshold of the block size that remote blocks will be fetched to disk
spark.shuffle.file.buffer	The size of the in-memory buffer for each shuffle file output stream
spark.memory.fraction	Fraction of Spark memory space available for execution and storage
spark.storage.memoryFraction	Amount of Storage memory immune to eviction (expressed as the fraction of the size of available memory space available for execution and storage)

To avoid the overfitting, during the memory requirement modelling, we apply a stepwise regression technique to each Spark workload category to select its own strong-correlated parameters for the memory requirements from the candidate parameters [20]. The stepwise regression is a step-by-step iterative construction of a regression model by adding or removing predictor items based on their statistical significance in the model. The stepwise regression method can be applied to the situation that there are many predictor items with limited sample data. To conduct the stepwise regression, we collect the sample data for each workload category, by measuring the memory requirements of workloads with different input data sizes and different settings of the candidate parameters on the given infrastructure. In each regression step, the p-value of a t-statistic is computed to test models with and without a potential parameter. We select the parameters whose p-values are smaller than or equal to the significance level of 0.05 as the strong-correlated parameters to the memory requirement modelling. Examples of the difference of selection results among workload categories will be shown in Section 6.1.

5.2 Memory Requirement Modelling

As shown in Section 2, the impacts of input data size and parameter settings on the memory requirement of Spark workload are diverse and complex. We choose the machine learning technique to model the memory requirement of Spark workloads in SMConf. To reduce the cost of sample data set generation, the selected machine learning method should be accurate with limited sample data. Hence, we adopt support vector machine (SVM) regression technique [21]. SVM regression is a kind of supervised machine learning method and works on the principle of support vector machine. It is effective to learn the continuous-valued functions in the situation of small-sized training data on many features and robust from outliers. In SMConf, for each Spark workload category, an individual SVM regression model is constructed to estimate their memory requirements under different input data sizes and configuration parameter settings.

The principle of SVM regression is to compute a linear regression function in a high dimensional feature space by mapping the input data via a nonlinear function. The linear regression model in an M feature space is then expressed as follows.

$$f(x) = \sum_{i=1}^{M} \alpha_i k(x_i, x) + b \tag{2}$$

where α_i is Lagrange multiplier, $k(x_i, x)$ denotes the kernel function that performs a linear dot product of nonlinear transformations and b is the bias term.

In SMConf, we choose the workload's input data size and the strong-correlated configuration parameters based on stepwise regression method as features and the workload's memory requirement as output of the SVM regression model. For sample data collection, we go through all selected Spark workloads in each workload category and capture their memory requirements with different input data sizes in the range of 200 MB to 35% of total memory size in a given Spark cluster, using different strong-correlated parameter settings. We adopt the most popular epsilon-intensive SVM ($\epsilon -$ SVM) regression method which finds the function f that deviates from the actual output by a value no greater than ϵ for all training data points. The LIBSVM library is applied and the Radial Basis Function is chosen as the kernel function [22,23].

SVM regression has a good generalization and prediction capacity due to that, it tries to minimize the generalized error bound, and hence, achieves the generalized performance. To assess the accuracy of the established model in practice, the k-fold cross-validation is performed. In k-fold cross-validation, the generated sample data is randomly partitioned into k equal-sized subsamples. Among k subsamples, a single subsample is retained as the validation data to assess the model, and the remaining k – 1 subsamples are used as training data. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the validation data. The k results from the folds can then be averaged to produce a single estimation. The advantage of this method is that all observations are used for both training and validation. For our assessment, the sensible choice is k = 10, as the estimation of prediction error is almost unbiased in 10-fold cross-validation.

5.3 *Model Matching for Ad-hoc Workloads*

The difficulties of applying the established memory requirement models to the online memory capacity configurations of Spark workloads is how to choose the proper model for diverse ad-hoc workloads with unpredictable memory requirement characteristics. SMConf solves the problem by profiling the ad-hoc workload with a small-sized input data, capturing the key features of the workload's memory requirement, and forming the feature vector. For each Spark workload category, the Euclidean distance of the feature vector to that of the centroid of the category is then calculated. The ad-hoc workload is finally matched to the category with the minimum distance. Then, the corresponding model is used to determine its memory capacity configuration.

6 Performance Evaluation

We have implemented SMConf in Spark version 2.3.4 by adding or modifying over 12 classes. Mainly, we add *JobProfiler, FeatureMatcher, MemoryPredictor* classes to enable the online ad-hoc Spark workload memory utilization predictions and modify *DAGScheduler* class to optimize the memory capacity configuration. We use *Perf, Sar* and *JProfiler* tool to capture the performance metrics on hardware layer, operating system layer and runtime layer of Spark system stack, respectively. The Spark workload classifications and the memory requirement models generated in the offline phase is stored as HDFS files and the file paths can be set with the new adding parameters *spark.ml.classification* and *spark.ml. predictionmodel*.

We evaluate the efficiency of SMConf for the memory requirement modelling of Spark workloads, the adaptiveness to ad-hoc workloads and the workload performance improvements in multi-tenant environments. We conduct experiments on a Spark cluster with five nodes. The node configuration is the same as that descried in Section 2. We use SparkBench to conduct the performance evaluation. SparkBench is a comprehensive benchmark suite for Spark ecosystem, which is mainly composed of three categories of workloads, the batch data analytics, the streaming processing and the interactive queries [12]. In this paper, we focus on the batch data analytics, and hence, choose all the twelve workloads from this category, including ShortestPaths (SP), Terasort (TS), PCA, K-Means (KM), LogisticRegression (LR), SVM, PageRank (PR), SVD++, Matrix Factorization (MF), TriangleCount (TC), ComponentConnected (CC) and DecisionTree (DT). The selected workloads cover various domains, such as machine learning, graph computation and etc., and have diverse memory consumption characteristics. Among them, we choose the former eight workloads as ***typical workloads*** to learn the memory requirement models and the latter four workloads as the ***ad-hoc workloads***.

In our experiments, we tune the parameter of *spark.executor.memory* from 1 GB to 12 GB, with the step size of 100 MB. We take the minimal configuration, that satisfies a Spark workload executing without any errors, exceptions and data re-computations, as its memory requirement and take it as the *proper memory capacity configuration* of the Spark workload. *The default memory capacity configuration* is set as 3 GB per CPU core, which is the recommendation from Apache Spark Community [7].

6.1 Accuracy of Memory Requirement Modelling

In our first experiment, we evaluate the accuracy of memory requirement modelling in SMConf. For this purpose, we use the eight typical workloads from SparkBench suite and predict their memory requirements with SMConf under different input data sizes and configuration parameter settings. Based on SMConf, these eight typical workloads are classified into four categories and the strong-correlated parameters are listed in Tab. 4.

For each workload, we vary the input data sizes and configuration parameter settings and generate 72 setting combinations. The input data size varies from 1 GB to 8 GB. The parameters are set as follows, *spark.executor.cores* as 1, 2 and 4, *dfs.block.size* as 64 MB and 128 MB, *spark.default. parallelism* as 2, 4, and 8, *spark.rdd.compress* and *spark.shuffle.compress* as 0 and 1.

We adopt the metric of Mean Relative Error (MRE) to measure the accuracy of memory requirement modelling for each workload category.

$$MRE = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|y_i - y_i^{'}\right|}{y_i^{'}} \tag{3}$$

where, y_i and $y_i^{'}$ are the predicted and actual values of a spark workload's memory requirement with a specific setting combination, respectively. n is the total number of observed values in each workload.

Table 4: Spark workload categories and their strong-correlated parameters

Category	Workloads	Strong-correlated parameters
#1	SVD++, PR	spark.executor.cores, dfs.block.size, spark.default.parallelism, spark.rdd.compress, spark.shuffle.compress
#2	SP, SVM	spark.executor.cores, dfs.block.size, spark.default.parallelism, spark.rdd.compress, spark.shuffle.compress
#3	TS, PCA	spark.executor.cores, dfs.block.size
#4	KM, LR	spark.executor.cores

Fig. 7 demonstrates the MRE of memory requirement prediction of each typical workload as the input data sizes and parameter settings varied.

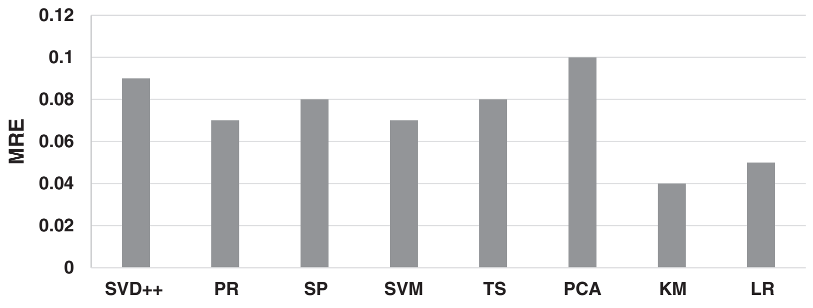


Figure 7: MRE of memory requirements of typical Spark workloads

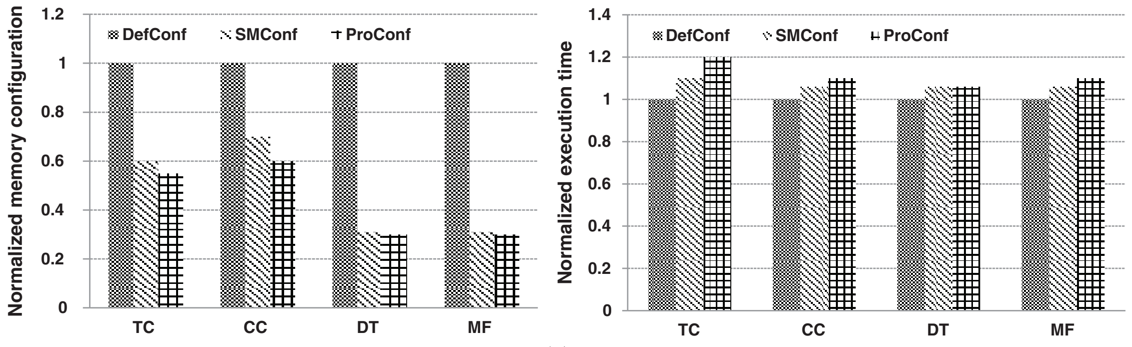
Generally, the MRE of memory requirement prediction is less than 10% in all test cases. Workloads belonging to category #4 (that is, KM and LR) have the best prediction accuracy, where the MRE is less than 5.1%. This is because that with the least strong-correlated parameter features, their memory requirement model learning has lower risk of overfitting with the same size of sample data set.

6.2 Adaptiveness to Ad-hoc Workloads

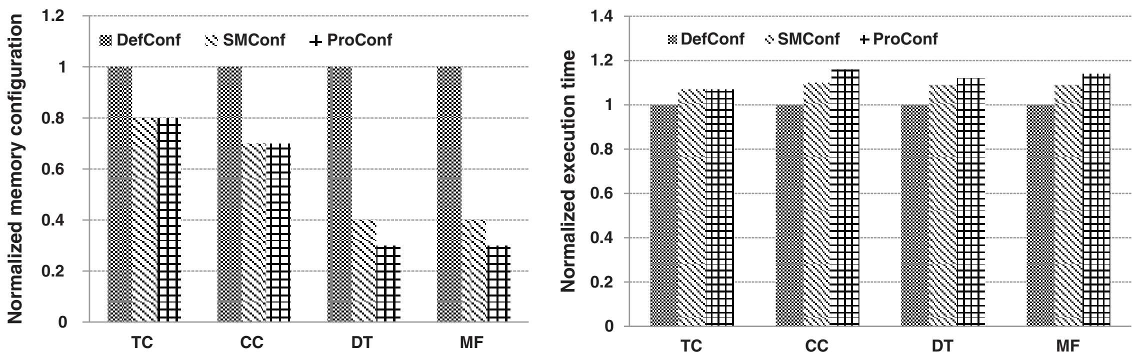
In this section, we evaluate SMConf's ability to determine the memory capacity configuration of ad-hoc workloads from SparkBench suite (that is, Matrix Factorization (MF), TriangleCount (TC), ComponentConnected (CC) and DecisionTree (DT)) and evaluate its performance impact on these ad-hoc workloads.

We adopt the typical workload classification and memory requirement modelling results in Section 6.1. Each ad-hoc workload is first matched to a workload category. Then, its memory capacity configuration is determined by the corresponding memory requirement model. We compare the memory capacity configuration with SMConf to the default configuration (DefConf) and the proper configuration (ProConf). As for the performance impact, we compare the execution time of ad-hoc workloads under the three configurations. In all test cases, we take the performance under the default configuration as the baseline and demonstrate the normalized results. We first evaluate SMConf's capacity with different input data size. We fix the Spark configuration parameters settings as default and vary the input data size of ad-hoc workloads as 2 GB, 4 GB and 8 GB. The results are shown in Fig. 8.

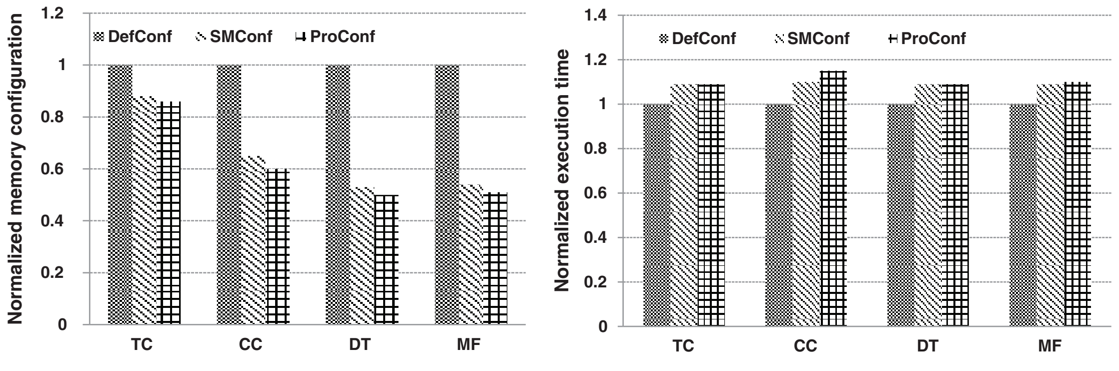
Fig. 8 demonstrates the normalized comparison results of memory capacity configuration under the input data size variation. Compared to the default configuration, the memory capacity configuration is reduced by 43% by average and 69% by maximum with SMConf. Meanwhile, particularly, when the input data size decreases from 8 GB to 2 GB, the average reduction ratio increases from 35% to 52%. Hence, SMConf can adapt to the input data size variation and correspondingly save more memory provision under small and medium input data size. On the other hand, the configuration with SMConf is close to the proper configuration, with the relative error no more than 0.16 by maximum in all cases. Hence, by classifying Spark workloads and use more effective prediction model, SMConf can determine the memory capacity configuration of Spark workloads accurately.



(a)



(b)



(c)

Figure 8: Impact of SMConf on memory configuration and workload performance for various input data sizes. (a) 2 GB, (b) 4 GB, (c) 8 GB

As for the workload performance, ad-hoc workloads can achieve the similar execution time with SMConf and the proper configuration. The relative error of SMConf with respect to the proper configuration is no more than 0.08 in all cases. On the other hand, compared to the conservative default configuration, the execution time increases no more than 10% with SMConf configuration. It is reasonable because that, with the black-box prediction model, SMConf may not decide as sufficient memory provision as the default configuration and leads events, such as page fault and etc., occur, which slow down the workload execution. Hence, we can draw the conclusion that, with SMConf, users can significantly reduce the excess memory provision with a slightly performance overhead.

We then evaluate SMConf's capacity with different configuration parameter settings. In this test, the input data size is fixed as 4 GB. For each ad-hoc workload, we measure its memory capacity configuration and the executions time with twenty various parameter setting combinations. The results are shown in Fig. 9.

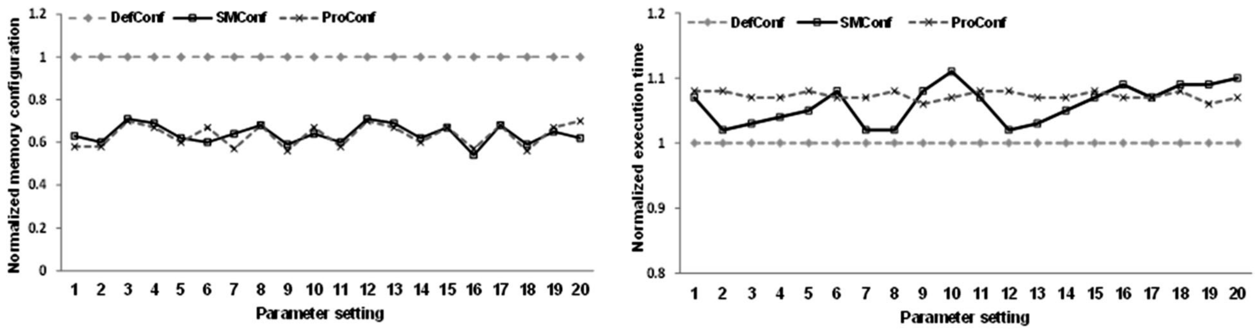


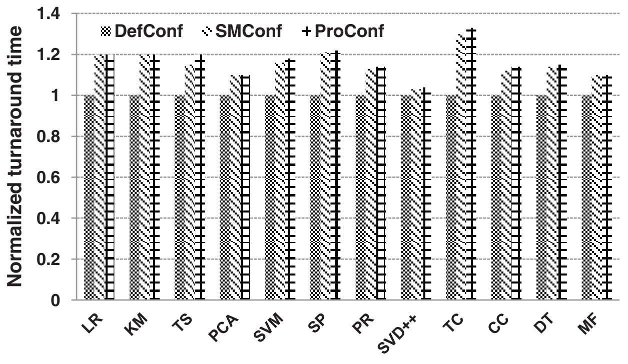
Figure 9: Impact of SMConf on memory configuration and workload performance for various parameter settings

Due to the space limit, we only demonstrate the normalized result of ComponentConnected (CC) workload. This is due to that SMConf achieves the similar performance on all the four ad-hoc workloads and demonstrates the most performance deviation on CC workload compared to the baseline methods. Fig. 9 demonstrates that, compared to the default configuration, the memory capacity configuration is reduced by 36.1% by average and 46% by maximum with SMConf. Meanwhile, compared to the proper configuration, the memory capacity configurations with SMConf are lower in test case #6, #10, #16, #19, and #20, and higher in other cases. Taking the proper configuration as baseline, the relative error of SMConf is less than 9% in all twenty test cases. In general, the memory capacity decisions between SMConf and the proper configuration are quite similar. Compared to the default setting, the increase of ad-hoc workload's execution time incurred with SMConf is slight, which is 5% by average and 11% by maximum. The comparison results of workload performance between SMConf and the proper configuration are in accordance with those of memory configuration. That is, CC workload achieves lower execution time with SMConf in test except of case #6, #10, #16, #19, and #20, due to being allocated with more memory resource. The relative error of SMConf with respect to the proper configuration is less than 0.05 in all cases. The results prove that SMConf can adapt to the variation of parameter settings and achieve accurate memory capacity configuration decision in an automated way.

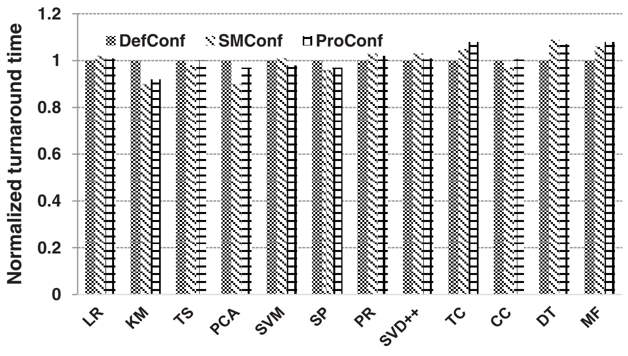
6.3 Performance Improvement in Multi-tenant Environments

In this section, we verify that, by reducing the excess memory capacity configurations, SMConf can help to reduce the turnaround time of Spark workloads in the multi-tenant environments, particularly in the situation of memory resource contention. We vary the memory resource provision of each node in the Spark cluster as 6 GB, 8 GB an 12 GB, to form the situation of intensive, normal and slight memory resource contentions, respectively. We use all the twelve workloads, that is eight typical workloads and four ad-hoc workloads, from SparkBench suite. The input data size and the parameter of *Spark.executor.cores* are set to 8 GB and 2, respectively and all other configuration parameters are set as default. Every two workloads are paired as a group and 66 workload groups are generated. In each memory provision situation, all 66 workload groups are submitted and executed in turn, where the CPU core provision in the Spark cluster is enough to accommodate two workloads in a group to execute concurrently. For each workload, all its turnaround times among the 66 group executions are collected to calculate its average turnaround time.

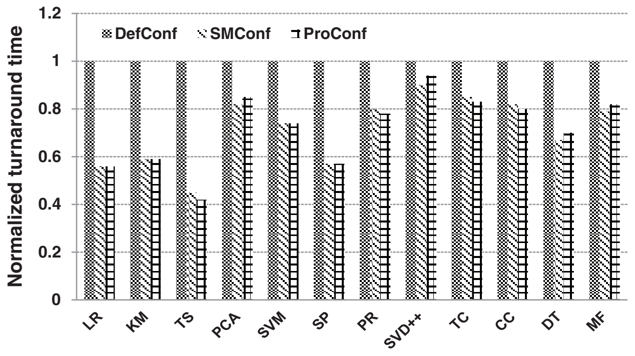
Fig. 10 demonstrates the normalized average turnaround time of Spark workloads under slight, normal and intensive memory contentions. As Fig. 10(a) shown, with the 12 GB memory provision on each node, the memory contention is slight; workloads within any workload pair can be executed concurrently. With SMConf configuration, the average turnaround time of Spark workloads is increased by 30% by maximum and 3% by minimum, compared to the default memory capacity configuration. As Fig. 10(b) shown, under the situation of normal memory contention, with SMConf configuration, five of the twelve workloads achieve lower average turnaround time than that with the default configuration. The reduction of average turnaround time is 10% by maximum and 2% by minimum. This is due to that, compared to the conservative default configuration, a workload with SMConf configuration needs less allocated memory resource to launch, which relieves the memory contention between two workloads within a group and leads to over half of workload groups can execute their workloads concurrently. The advantage of SMConf is more significant under the intensive memory resource contention. As shown in Fig. 10(c), all workloads achieve lower average turnaround time with SMConf. The reduction of average turnaround time is 29% by average, 55% by maximum and 10% by minimum. On the other hand, the average turnaround time of Spark workloads with SMConf is similar to that with the proper configuration, with the relative error no more than 0.07 in all test cases. It proves that, by using SMConf in the multi-tenant environment, the higher Spark workload throughput can be achieved via the more efficient memory utilizations.



(a)



(b)



(c)

Figure 10: Impact of SMConf on workload performance for various multi-tenant environments. (a) 12 GB, (b) 8 GB, (c) 6 GB

7 Related Work

Recently in-memory data analytic platform and its representative implementation Spark have gained much research [24–27]. As the crucial resource, Spark memory management has been the research hotspot [28–33]. Works on Spark memory management mainly focus on two aspects: One is memory capacity configuration of Spark workloads. The other is data caching strategies and mechanisms for Spark memory space.

On memory capacity configuration of Spark workloads, MEMTUNE [5] dynamically tunes the computation/caching partitions for Spark memory space, based on workload memory demand and data cache need. MEMTUNE determines Spark workload's demand and data cache need by monitoring the memory contention occurring during the RDD caching and data shuffling operations, and gradually increasing or reducing the memory capacity for a specific partition by one unit. DSpark [29] aims to dynamically tunes the total memory space allocated to the Spark workloads by adjusting the total number of its assigned executors. DSpark decides the executor number by modelling the application cost and completion time with respect to the number of allocated executors. To our best knowledge, SMConf is the first approach to conduct the memory capacity configuration by determining the proper memory capacity of each Spark executor and performing the more fine-grained excess memory allocation reduction. On the other hand, the above works conduct the apple-to-apple configurations, while SMConf is the first approach to propose the one-size-fit-bunch solution and can be more generalized.

More works focus on data caching strategies and mechanisms for Spark memory space. CSAS [30] establishes the cost-based data caching model and determines the optimized cache level for each partition of RDD, the data serialization and compression policy. Koliopoulos et al. [31] proposed a mechanism of automatic cache tuning for Spark. The mechanism used heuristic algorithms to guarantee the data caching performance as well as to reduce the footprint of memory space as much as possible. S-cache [32] is a caching model based on time cost, which chooses different caching strategies for a Spark job's different stages in its DAG model. S-cache adopts the graph search algorithm to optimize the data replacement strategy in Spark data cache under the constraint of minimizing the time cost. However, S-cache makes the data caching decision without considering the variations of memory utilization during a Spark job execution and can only form the theoretically optimal decision. Perez et al. [33] proposed a reference distance-based data prefetching and eviction strategy for Spark data cache, which uses stage distance and job distance to calculate the reference distance of data. The strategy chooses the RDD with the similar reference distance to prefetch and evicts the RDDs with the largest reference distance. This method only suited for the iterative jobs. SADP [34] is a data prefetching mechanism based on the scheduling information of Spark tasks for the multi-job scenario. SADP proposes the data replacement strategy based on data value and can be applied to the Map/Reduce frameworks. JeCache [35] is a time-constrainted and task-level data prefetching mechanism. JeCache also concerns the excessive memory allocation in Spark. However, it achieves the efficient memory utilization by prefetching data on demand and can also reduce the execution time of Spark job. Liu et al. [36] adopts the data prefetching mechanism to reduce the I/O cost in the Shuffle phase of Spark jobs. Through prefetching the data, which is transferred from mapper task to reducer task, the efficiency of I/O and network resource utilization can be prompted. Wang et al. [37] propose the optimized memory allocation strategy, which includes the optimized the cache replacement algorithm according to the characteristics of the RDD partition and the PCA dimensionality reduction mechanism. FlexFetch [38] estimates the data prefetching demands and uses the critical-aware deadlock-driven algorithm to schedule parallel prefetching with SDN technology. Though all these works focus on the data caching in Spark, the consideration factors and tools involved in the strategy designs are worth to learn in our solution, particularly going deep into the stage-based operations in a Spark job execution and the metric collector tools. Wang et al. [39] propose the optimization approach for RDDs cache and LRU based on the features of partitions.

There are still works on the parameter configurations of Spark. Most of these works focus on handling with the large-amount configuration parameters in Spark and try to learn the correlations among them. Wang et al. [40] proposes a machine learning-based automated method for Spark parameter tuning, which is composed of binary classification and multi-classification. Nguyen et al. [41] proposes an execution model-driven framework to quantify the influence of parameter settings in Spark. Gu et al. [42] adopts a neural network model to determine the increasing or decreasing operation in optimal parameter search and proposes a random forest-based performance model to predict Spark workload's performance with various parameter settings. Nguyen et al. [43] use Latin hypercube design strategy to identify configurations for benchmarking the system, compare three machine learning methods in Spark workload performance modelling, and finally adopt the recursive random search algorithm to achieve the optimized parameter setting. Although diverse machine learning methods applied in the parameter settings, these works conduct the case-to-case performance modelling and their focus mainly lies on the exploration of complex correlations among over 180 Spark parameters.

Works on data center workload characteristics analysis, that learned in our paper, are as follows. Jiang et al. [14] make a comprehensive, quantitative analysis of Spark workload characteristics on the system and micro-architecture levels. Zhen et al. [44] provide the insights of system-level behaviour patterns of big data analytic workloads. Liu et al. [45] analyze the memory content similarity of workloads in data center and conduct the virtual machine migration to improve the memory utilization.

8 Conclusion and Future Work

Spark is the most representative in-memory distributed processing framework for big data analytics. Memory is the key resource to accelerate Spark workload's performance. However, the current approach adopted in Spark is to statically configure the workload's memory capacity based on user's specifications. To guarantee workload performance, non-expert users usually overbook the memory capacity for their workloads, which leads the severe waste of memory resource in Spark clusters. SMConf is proposed and developed to enable automated memory capacity configuration for big data analytics workloads in Spark, which guarantees both the workload performance and the memory utilization improvement. SMConf is designed based on the observation that, though there is not one-size-fit-all proper configuration, the one-size-fit-bunch configuration can be found for in-memory computing workloads. With the proposed workload classification and the memory requirement modelling methods, SMConf is able to make optimal memory capacity configuration decision for ad-hoc Spark workloads, which improve the memory utilization and the workload throughput in the multi-tenant environments.

In the future work, we will extend SMConf to other in-memory processing frameworks and optimize the memory requirement modelling method in SMConf.

Funding Statement: This work is supported by National Key R&D Program of China (No. 2017YFC0803300) and the National Natural Science of Foundation of China (No. 61703013).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] L. Dong, Z. Y. Lin, L. Yan, H. Ling, Z. Ning *et al.,* "A hierarchical distributed processing framework for big image data," *IEEE Transactions on Big Data*, vol. 2, no. 4, pp. 297–309, 2016.

[2] H. Fei, C. W. Yang, J. Schnase, D. Duffy, M. C. Xu *et al.,* "ClimateSpark: An in-memory distributed computing framework for big climate data analytics," *Computers & Geosciences*, vol. 115, no. 6, pp. 154–166, 2018.

[3] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. IEEE Int. Conf. on Cloud Computing Technology and Science*, Berkeley, CA, USA, pp. 10, 2010.

[4] M. Kang and J. Lee, “An experimental analysis of limitations of MapReduce for iterative algorithms on Spark,” *Cluster Computing*, vol. 20, no. 4, pp. 3593–3604, 2017.

[5] L. Xu, M. Li, L. Zhang, A. Butt, W. Yandong *et al.,* “MEMTUNE: Dynamic memory management for in-memory data analytic platforms,” in *Proc. Int. Parallel and Distributed Processing Sym.*, Chicago, IL, USA, pp. 383–392, 2016.

[6] K. Shanmugam, *Best Practices for Successfully Managing Memory for Apache Spark applications on Amazon EMR*. Seattle, WA, USA: AWS Big Data, 2019. [Online]. Available: https://aws.amazon.com/cn/blogs/big-data/best-practices-for-successfully-managing-memory-for-apache-spark-applications-on-amazon-emr/.

[7] Z. Tang, A. Zeng, X. D. Zhang, L. Yang and K. L. Li, “Dynamic memory-aware scheduling in Spark computing environment,” *Journal of Parallel and Distributed Computing*, vol. 141, no. 2, pp. 10–22, 2020.

[8] Big Data Management Group, *Performance Tuning for the Spark Engine*. Redwood, CA, USA: Informatica, 2020. [Online]. Available: https://kb.informatica.com.

[9] C. Reiss, A. Tumanov, G. Ganger, R. Katz and M. Kozuch, “Heterogeneity and dynamicity of clouds at scale: Google trace analysis,” in *Proc. ACM Sym. on Cloud Computing*, San Jose, CA, USA, pp. 1–13, 2012.

[10] L. Barroso, U. Holzle, P. Ranganathan and M. Martonosi, “Modeling costs, ” in *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed., vol. 1. San Rafarel, CA, USA: Morgan Claypool Publishers, 2018.

[11] Y. Q. Zhu, J. X. Liu, M. Y. Guo, Y. G. Bao, W. Ma *et al.,* “BestConfig: tapping the performance potential of systems via automatic configuration tuning,” in *Proc. ACM Sym. on Cloud Computing*, Santa Clara, CA, USA, pp. 38–350, 2017.

[12] M. Li, J. Tan, Y. D. Wang, L. Zhang and V. Salapura, “SparkBench: A comprehensive benchmarking suite for in memory data analytic platform Spark,” in *Proc. ACM Int. Conf. on Computing Frontiers*, Ischia, Italy, pp. 1–8, 2015.

[13] C. Bienia, S. Kumar, J. Singh and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, New York, NY, USA, pp. 72–81, 2008.

[14] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. McKee *et al.,* “Understanding the behavior of in-memory computing workloads,” in *Proc. IEEE Int. Sym. on Workload Characterization*, Raleigh, NC, USA, pp. 22–30, 2014.

[15] D. Patterson and J. Hennessy, “Memory hierarchy design, ” in *Computer Architecture: A Quantitative Approach*, 6th ed., vol. 1. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2019.

[16] Y. Mu, X. Liu and L. X. Wang, “A Pearson’s correlation coefficient based decision tree and its parallel implementation,” *Information Sciences*, vol. 435, pp. 40–58, 2018.

[17] X. L. Meng, “Posterior predictive *p*-values,” *Annals of Statistics*, vol. 22, no. 3, pp. 1142–1160, 1994.

[18] P. HaeSang and J. ChiHyuck, “A simple and fast algorithm for k-medoids clustering,” *Expert Systems with Applications*, vol. 36, no. 2, pp. 3336–3341, 2009.

[19] D. Faith, P. Minchin and L. Belbin, “Compositional dissimilarity as a robust measure of ecological distance,” *Vegatatio*, vol. 69, no. 1–3, pp. 57–68, 1987.

[20] K. Weinberger and K. Saul, “Distance metric learning for large margin nearest neighbor classification,” *Journal of Machine Learning Research*, vol. 10, no. 9, pp. 207–244, 2009.

[21] E. Steyerberg, J. Eijkemans and J. Habbema, “Stepwise selection in small data sets: A simulation study of bias in logistic regression analysis,” *Journal of Clinical Epidemiology*, vol. 52, no. 10, pp. 935–942, 1999.

[22] U. Girosi, “On the noise model of support vector machine regression,” in *Proc. Int. Conf. on Algorithmic Learning Theory*, Washington, DC, USA, pp. 316–324, 2001.

[23] C. C. Chang and C. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 1–27, 2011.

[24] J. N. Chen, K. L. Li, Z. Tang, K. Bilal, S. Yu *et al.,* “A parallel random forest algorithm for big data in a Spark cloud computing environment,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 919–933, 2017.

[25] K. W. Wang and M. Khan, "Performance prediction for Apache Spark platform," in *Proc. IEEE Conf. of High Performance Computing and Communications*, New York, NY, USA, pp. 166–173, 2015.

[26] D. Zhou, J. H. Guo, Y. Zhang, J. D. Chai, H. Liu *et al.,* "Distributed data analytics platform for wide-area synchrophasor measurement systems," *IEEE Transactions on Smart Grid*, vol. 7, no. 5, pp. 2397–2405, 2016.

[27] A. Maarala, R. Mika and S. Miikka, "Low latency analytics for streaming traffic data with Apache Spark," in *Proc. Int. Conf. on Big Data*, Santa Clara, CA, USA, pp. 2855–2858, 2015.

[28] X. Zhang, Z. Li, G. Liu, J. Xu, T. Xie *et al.,* "A Spark scheduling strategy for heterogeneous cluster," *Computers, Materials & Continua*, vol. 55, no. 3, pp. 405–417, 2018.

[29] M. slam, S. Karunasekera and R. Buyya, "DSpark: Deadline-based resource allocation for big data applications in Apache Spark," in *Proc. IEEE Int. Conf. on e-Science (e-Science)*, Auckland, New Zealand, pp. 89–98, 2017.

[30] B. Wang, J. Tang, R. Zhang and Z. M. Gu, "CSAS: Cost-based storage auto-selection, a fine grained storage selection mechanism for Spark," in *Proc. IFIP Int. Conf. on Network and Parallel Computing*, Hefei, China, pp. 150–154, 2017.

[31] K. Koliopoulos, P. Yiapanis and F. Tekiner, "Towards automatic memory tuning for in-memory big data analytics in clusters," in *Proc. IEEE International Congress on Big Data*, San Francisco, CA, USA, pp. 353–356, 2016.

[32] G. Vinicius, M. Kilchenman and V. Patoglu, "Automatic caching decision for scientific dataflow execution in Apache Spark," in *Proc. ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, Huston, TX, USA, pp. 1–10, 2018.

[33] T. Perez, X. Zhou and D. Cheng, "Reference-distance eviction and prefetching for cache management in Spark," in *Proc. Int. Conf. on Parallel Processing*, Eugene, OR, USA, pp. 1–10, 2018.

[34] M. Zhang, R. Chen and X. Zhang, "Intelligent RDD management for high performance in-memory computing in Spark," in *Proc. Int. Conf. on World Wide Web Companion*, Perth, Australia, pp. 873–874, 2017.

[35] Y. Luo, S. Jia and S. Zhou, "JeCache: Just-enough data caching with just-in-time prefetching for big data applications," in *Proc. IEEE Int. Conf. on Distributed Computing Systems*, pp. 2405–2410, 2017.

[36] S. Liu, H. Wang and B. Li, "Optimizing shuffle in wide-area data analytics," in *Proc. IEEE Int. Conf. on Distributed Computing Systems*, Atlanta, GA, USA, pp. 560–571, 2017.

[37] S. Z. Wang, S. S. Geng and Z. F. Zhang, "A dynamic memory allocation optimization mechanism based on Spark," *Computers, Materials & Continua*, vol. 109, no. 2, pp. 537–554, 2019.

[38] Z. Yu, M. Li, X. Yan, H. Zhao and X. L. Li, "Taming non-local stragglers using efficient prefetching in MapReduce," in *Proc. Int. Conf. on Cluster Computing*, Chicago, IL, USA, pp. 52–61, 2015.

[39] S. Z. Wang, Y. P. Zhang, L. Zhang, N. Cao and C. Y. Pang, "An improved memory cache management study based on Spark," *Computers, Materials & Continua*, vol. 56, no. 3, pp. 415–431, 2018.

[40] G. L. Wang, J. G. Xu and B. He, "A novel method for tuning configuration parameters of Spark based on machine learning," in *Proc. Int. Conf. on High Performance Computing and Communication*, Bangkok, Thailand, pp. 586–593, 2017.

[41] N. Nguyen, M. Khan, Y. Albayram and K. Wang, "Understanding the influence of configuration settings: An execution model-driven framework for Apache Spark platform," in *Proc. IEEE Int. Conf. on Cloud Computing*, Honolulu, Hawaii, USA, pp. 802–807, 2017.

[42] J. Gu, Y. Li, H. Y. Tang and Z. H. Wu, "Auto-tuning Spark configurations based on neural network," in *Proc. Int. Conf. on Communications*, Kansos City, MO, USA, pp. 1–6, 2018.

[43] N. Nguyen, M. Khan and K. Wang, "Towards automatic tuning of Apache Spark configuration," in *Proc. Int. Conf. on Cloud Computing*, San Francisco, CA, USA, pp. 417–425, 2018.

[44] J. Zhen, L. Wang, J. F. Zhan, L. X. Zhang and C. J. Luo, "Characterizing data analysis workloads in data centers," in *Proc. IEEE Int. Sym. on Workload Characterization*, Raleigh, North Carolina, USA, pp. 66–76, 2014.

[45] H. X. Li, W. J. Li, H. D. Wang and J. X. Wang, "An optimization of virtual machine selection and placement by using memory content similarity for server consolidation in cloud," *Future Generation Computer Systems*, vol. 84, pp. 98–107, 2018.