Blink: Lightweight Sample Runs for Cost Optimization of Big Data Applications

Hani Al-Sayeh[1](✉), Muhammad Attahir Jibril[1], Bunjamin Memishi[2], and Kai-Uwe Sattler[1]

[1] TU Ilmenau, Ilmenau, Germany
{hani-bassam.al-sayeh,muhammad-attahir.jibril,kus}@tu-ilmenau.de
[2] Riinvest College, Pristina, Kosovo
bunjamin.memishi@riinvest.net

Abstract. Distributed in-memory data processing engines accelerate iterative applications by caching datasets in memory rather than recomputing them in each iteration. Selecting a suitable cluster size for caching these datasets plays an essential role in achieving optimal performance. We present Blink, an autonomous sampling-based framework, which predicts sizes of cached datasets and selects optimal cluster size without relying on historical runs. We evaluate Blink on iterative, real-world, machine learning applications. With an average sample runs cost of 4.6% compared to the cost of optimal runs, Blink selects the optimal cluster size, saving up to 47.4% of execution cost compared to average cost.

1 Introduction

Modern distributed systems such as Spark [17] enhance the performance of iterative applications by caching crucial datasets in memory instead of recomputing or fetching them from slower storage (e.g., HDFS) in each iteration [16]. To measure the impact of repetitive re-computations on system performance, we run *Support Vector Machine* application (svm in Spark MLLib 2.4.0 [11]) on an input dataset of 59.5 GB using different cluster sizes (1–12 machines) on our private cluster (cf. Sect. 6). We measure the execution time and the cost (#machines × time) of each run. As depicted in Fig. 1, we distinguish three areas:

- Area A : Increasing the cluster size decreases both execution time and cost.
- Area B : Increasing the cluster size decreases time but increases cost.
- Area C : The junction of A&B , where the highest cost efficiency is achieved.

In area A, the total memory capacity of the cluster machines is not enough for caching all partitions of a certain crucial dataset in svm. As a result, many of its partitions do not fit in memory and are re-computed in all iterations, which is very expensive. A deeper dive into a single iteration shows that: 1. The percentage of cached data partitions in area A for 1 to 7 machines are 17%, 35%, 52%, 70%, 87%, 92% and 100% respectively. 2. On average, a task that reads an already cached partition runs 97× shorter than a task that recomputes a partition of equal size. In area B, increasing the cluster size reduces the execution time of the parallel part of the application but does not influence the serial part [6]. The data transfer overhead between machines also increases. These factors decrease cost efficiency.

Currently, optimal resource provisioning based on accurate prediction of the size of cached datasets remains an open challenge. In summary, we make the following contributions:

- We introduce an efficient approach for minimizing the cost of sample runs.
- We present BLINK, a lightweight sampling-based framework that predicts the size of cached datasets and selects an optimal cluster size (area C).
- We perform an extensive analysis of machine learning applications and stress their minimal sampling requirements for an optimal cluster size selection.

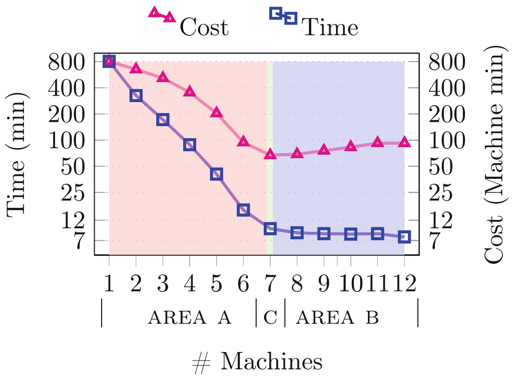


Fig. 1. Selection of cluster size (SVM)

We evaluate BLINK on 8 real-world applications. Relying on tiny sample datasets, BLINK selects the optimal cluster size for all 8 actual runs, which reduces execution cost to 52.6% compared to the average cost across all cluster sizes with an average sample runs cost of 4.6%.

2 Related Work

Caching decision support tools help application developers to determine which datasets to cache and when to purge them from memory [1,10]. However, these tools do not consider the size of the datasets and the required cluster configuration that guarantees eviction-free runs.

Cache eviction policies and **approaches to auto-tuning of memory configuration** tackle cache limitation in a best-effort manner but with penalties caused by cache eviction. This makes them suitable solutions if an inappropriate cluster size is selected (area A in Fig. 1). MRD [12] and LRC [16] are DAG-aware cache eviction policies in Spark that rank cached datasets based on their reference distance and reference count respectively. We apply both policies for the same SVM experiments (depicted in Fig. 1) and do not realize any performance improvement. This is because only one dataset is cached in SVM. MemTune [15] is a memory manager that re-adjusts memory regions during application run. RelM [9] introduces a safety factor to ensure error-free execution in resource-constrained clusters.

Runtime Prediction Approaches. Ernest [14] is a sampling-based framework that predicts the runtime of compute-intensive long-running Spark applications.

To reduce the overhead of sample runs, it decreases the number of iterations during these sample runs to make their overhead tolerable. This is not always practical because tuning an application parameter like the number of iterations during sample runs requires end users to have knowledge of the application and its parameters, which they might lack. Masha [4] is a sampling-based framework for runtime prediction of big data applications. Both frameworks do not address cache limitation issues.

Approaches for recommendation of cluster configuration rely on sample (or historical) runs to predict (near-to-) optimal cluster configuration. CherryPick [5] aims to be accurate enough to identify poor configurations and adaptive using a black-box approach, but without considering cache limitations. Juggler [3] considers application parameters to recommend cluster configurations with autonomous selection of datasets for caching. But, its offline-training overhead is not tolerable and, thus, it is limited to recurring applications.

3 Background

Spark runs applications on multiple *executors* that perform various parallel operations on partitioned data called Resilient Distributed Dataset or RDD [17]. A class of operations called *transformations* (e.g., filter, map) create new RDDs from existing ones while another class called *actions* (e.g., count, collect) return a value to the (*driver*) program after making computations on RDDs. An application is the highest level of computation and consists of one or more sequential *jobs*, each of which is triggered by an action. A job comprises of a sequence of transformations, represented by a DAG, followed by a single action. When a transformation is applied on an RDD, a new one is created. The parent-child dependency between RDDs is represented in a logical plan, by way of a lineage or DAG starting from an action up to either the root RDDs that are cached or original data blocks from the distributed file system. As different jobs may consist of many transformations in common, we merge all their DAGs to represent an application in a single DAG of transformations, as illustrated with the *Logistic Regression* application in Fig. 2. The number of times a dataset is computed is determined by the number of its child branches in the resulting DAG.

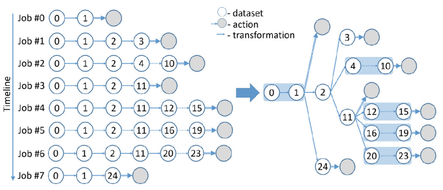


Fig. 2. Merging DAGs.

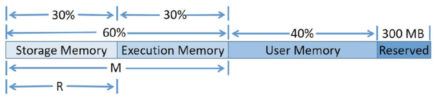


Fig. 3. Spark: Memory layout.

As depicted in Fig. 3, Spark splits memory into multiple regions. We focus on the storage and the execution regions, respectively used for caching datasets and computation [18]. Both regions share the same memory space (i.e., the unified region M) such that if the execution memory is not utilized, all the available memory space can be used for caching, and vice versa. There is a minimum storage space R below which cached data is not evicted. That is, in each executor, at least R and at most M can be utilized to cache datasets.

4 Efficient Sample Runs

In this section, we explain how to minimize the cost of sample runs with empirical evaluations. Specifically, we show that the sample run phase required for predicting the size of the cached datasets is less challenging than that required for execution time prediction. As previous studies tackle data sampling challenges [7,8], we do not address them in this work, similar to Ernest [14] and Masha [4].

4.1 Size of Sample Runs

Few sample runs are sufficient to predict the size of the cached datasets. For example, if we conduct two short-running experiments of the same application using the same data and same cluster configuration, the sizes of datasets do not vary. However, this is not the case regarding execution time. To validate this, we select SVM, which caches one dataset, to run 10 experiments on 738.1 MB (data scale 1, 12 blocks), 10 experiments on 1501.6 MB (data scale 2, 24 blocks) and 10 experiments on 2.2 GB (data scale 3, 36 blocks). We conduct all runs on a single machine. As illustrated in Fig. 4, we see that the size of the cached dataset remains constant in all runs of the same data scale. Also, we notice a considerable variance in execution time between the runs of the same data scale, which affects the construction and training of prediction models. To overcome this problem, we either run several experiments on the same data scale and obtain the statistical average (or median) or increase the size of sample datasets to make sample runs longer and, thus, the execution time variance relatively lower. However, both solutions increase the cost of sample runs tremendously, which explains why runtime prediction approaches are limited to long-running applications.

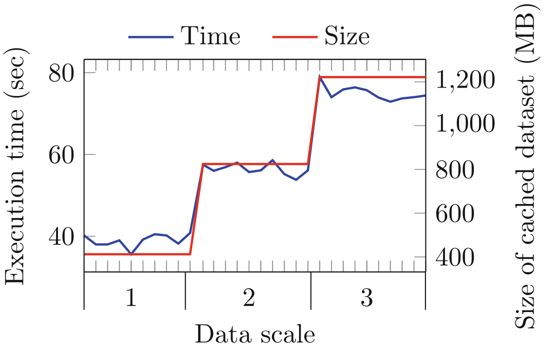


Fig. 4. Short-running experiments.

To build size prediction models of the cached datasets, we carry out sample runs on tiny datasets within the range of 0.1%–0.3% of the original data.

4.2 Parallelism

Distributed file systems (e.g., HDFS) store original data by fragmenting it into equal chunks, namely blocks. The size of blocks is configurable (64 or 128 MB by default). In order to decrease the data size during sample runs, we either (1) reduce the size of each block (BLOCK-S), or (2) select few data blocks (BLOCK-N). For example, if the block size is configured to be 64 MB, 1 TB of data is stored in 16K blocks. Thus, 16 blocks out of them could be selected for a sample run of 0.1% of the original data.

BLOCK-N is less costly than BLOCK-S because it requires selecting data blocks from a distributed file system. BLOCK-S is more complicated and brings extra overhead in preparing the sample data. Since we are not expecting memory limitation during sample runs, increasing the parallelism increases the execution time of each sample run (i.e., data shuffling and cleaning).

In order to validate this, we conduct two runs of SVM with an input data of 1.2 GB on a single machine. The number of data blocks in the first run is 10 and it takes 41 s. In the second run, the number of data blocks is 1000 and it takes 3.5 min. In addition, during the first and second runs, the size of the cached dataset is 728.9 MB and 747.8 MB, respectively. This shows that the size of datasets is influenced by the parallelism level. Hence, in the case of BLOCK-N, if we reduce the number of tasks during sample runs, then predicting the size of the cached datasets might be affected. To tackle this problem, we keep the number of tasks proportional to the data scale by fixing the block size. For example, if the full-scale dataset consists of 16K blocks, then the sample runs with 0.1%, 0.2% and 0.3% of the input data scale will contain 16, 32, and 48 tasks respectively.

For some applications, the size of the original data is relatively small (as we will show in Sect. 6) and, thus, the number of its blocks is not enough to apply BLOCK-N. In such cases, BLOCK-S is used in spite of its costs.

4.3 Cluster Configuration

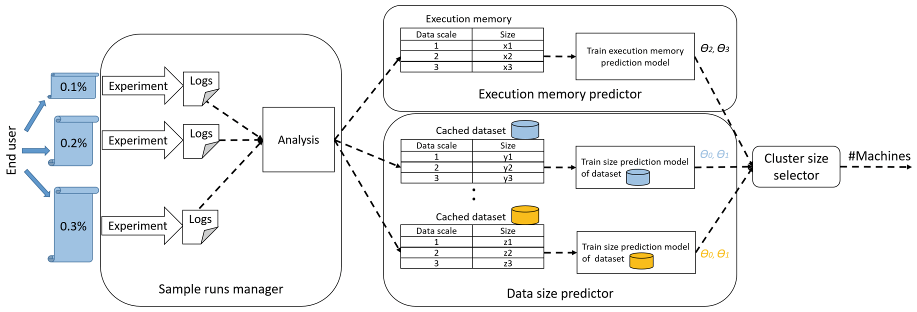
We carry out all sample runs on a single machine to reduce the cost of the sample runs. The serial part of a short-running experiment is relatively high compared with the parallel part and, hence, adding more machines during a sample run might not speed up the execution time. Rather, it leads to higher execution costs because of the increased overhead of negotiating resources (e.g., by YARN) and the increase in data transfer overhead with the addition of more machines. To validate this, we run SVM on 1.2 GB input data using a single machine and also using 12 machines. The execution cost on 12 machines is 13.9× higher than on a single machine. The exception that makes carrying out sample runs on a single machine too costly is when cached datasets do not fit in the memory of a single machine. However, this is unlikely for sample runs with tiny datasets.

4.4 Number of Sample Runs

Our experiments with all applications in HiBench 7.0 show that the prediction models for the size of the cached (and non-cached) datasets with respect to the input data scale are linear. Therefore, two sample runs are sufficient to construct a model. However, knowing that sample runs are lightweight, more sample runs could be conducted to apply *cross validation* to choose a well-fitting linear model.

5 Blink

We present Blink, a sampling-based framework that performs optimal resource provisioning for big data applications. As depicted in Fig. 5, the *Sample runs manager* (Sect. 5.1) first carries out lightweight sample runs on 0.1%–0.3% data samples of the original data. Based on these runs, the *size predictor* (Sect. 5.2) and *execution memory predictor* (Sect. 5.3) train prediction models to predict the size of cached datasets and the required amount of execution memory per machine in the actual run respectively. Finally, based on these models and the allocated memory in each machine, the *cluster size selector* (Sect. 5.4) selects the optimal cluster size that guarantees eviction-free runs.



Data scale	Size
1	x1
2	x2
3	x3

Data scale	Size
1	y1
2	y2
3	y3

Data scale	Size
1	z1
2	z2
3	z3

Fig. 5. Overview of Blink.

5.1 Sample Runs Manager

The sample runs manager carries out three sample runs on tiny data samples (0.1%–0.3% of the original data) on a single machine and monitors the sample runs to make quick decisions regarding the following atypical cases:

– If there is no cached dataset in the application, the sample runs manager selects a single machine (the longest execution time but the cheapest cost).
– If there are cached datasets and eviction occurs, which is unusual with tiny datasets, it carries out new sample runs with smaller sampling scales.

While conducting sample runs, *SparkListener* collects runtime metrics and stores them as log files in the distributed file system (e.g., HDFS). The sample runs manager analyzes the logs and collects the size of each cached dataset.

5.2 Data Size Predictor

After carrying out sample runs, the data size predictor trains the following model to predict the size of cached datasets in the actual runs:

$$\mathrm{D}_{size} = \theta_0 + \theta_1 \times datascale \tag{1}$$

Our experiments show that the sizes of all cached datasets fit into this model. For each cached dataset, the data size predictor takes the scale of the data sample as a feature and its size as a label. Thus, the scales in sample runs are 1, 2, and 3; while in the actual run, the scale is 1000. We use the *curve_fit* solver with enforced positive bounds to train the models while avoiding negative coefficients, and Root Mean Square Error (RMSE) to evaluate the models.

5.3 Execution Memory Predictor

The minimum and the maximum amount of memory for caching in each machine are known (M and R in Fig. 3) and, in turn, the minimum and the maximum number of machines can be determined using the following equations:

$$Machines_{min} = \lceil \frac{\sum^{CachedDs} \mathrm{D}_{size}}{M} \rceil$$
$$Machines_{max} = \lceil \frac{\sum^{CachedDs} \mathrm{D}_{size}}{R} \rceil$$

where $\sum^{CachedDs} \mathrm{D}_{size}$ is the total size of cached datasets, R is the memory region used for caching and M is the unified memory region for both caching and execution (cf. Fig. 3). Selecting less than $Machines_{min}$ leads to cache eviction because utilizing the whole unified memory space (i.e., M) in each machine for caching will not be enough to cache all datasets. In contrast, allocating more than $Machines_{max}$ gives no caching benefits since utilizing the storage memory (i.e., R) in each machine will be enough for caching all datasets. In other words, $Machines_{max}$ is required to cache datasets without eviction when the entire (M−R) memory region is utilized for execution. If M is not utilized at all, then the entire region can be used for caching and, hence, $Machines_{min}$ is required to cache datasets without evictions. Considering that the gap between $Machines_{min}$ and $Machines_{max}$ may be quite wide and the execution memory utilization differs from one application to another, there is a need for a precise prediction of the amount of memory required for execution. Similar to the data size predictor (cf. Sect. 5.2), the execution memory predictor analyzes the execution memory usage in sample runs and trains linear models to predict the total amount of execution memory required for the actual runs. Our experiments show that the relationship between the data sample scale and the amount of execution memory fits into the following model, although the execution memory predictor evaluates many other models:

$$Memory_{execution} = \theta_2 + \theta_3 \times datascale$$

5.4 Cluster Size Selector

Based on M and R in Fig. 3 (i.e., machine/instance specification), the cluster size selector calculates the required memory for execution per machine as follows:

$$MachineMemory_{execution} = \min(M - R, \frac{Memory_{execution}}{Machines})$$

Then, it selects the minimal number of machines that fulfills the condition below:

$$\frac{\sum^{CachedDs} \mathrm{D}_{size}}{Machines} < (M - MachineMemory_{execution}) \times Machines$$

In multi-tenant environments, the recommended cluster configuration is not affected by concurrent application runs hosted on the same machines because they are deployed in isolated virtual machines, and cluster managers (e.g., YARN [13]) do not offer an occupied memory region (i.e., M) to newly submitted applications.

Table 1. Evaluated Spark MLlib applications. Recommended cluster size is shown in bold. Shadowed cells refer to cluster sizes that do not cause cache evictions. Time unit is represented in minutes. Cost unit is represented in machine minutes.

	#Machines	ALS		BAY		GBT		KM		LR		PCA		RFC		SVM	
		Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
Sample runs	1	5.8	5.8	1.4	1.4	1.4	1.4	1.2	1.2	1.0	1.0	7.7	7.7	3.9	3.9	1.2	1.2
Approach		BLOCK-S		BLOCK-N		BLOCK-S		BLOCK-S		BLOCK-N		BLOCK-S		BLOCK-N		BLOCK-N	
Scale 100% (size)		5.6 GB		17.6 GB		30.6 MB		21.5 GB		22.4 GB		1.5 GB		29.8 GB		59.6 GB	
Scale 100% (#Blocks)		100		2K		100		200		2K		50		2K		2K	
Actual runs	1	**27.2**	**27.2**	63.3	63.3	**9.8**	**9.8**	137.2	137.2	337	337	**77.4**	**77.4**	361.6	361.6	804.8	804.8
(100% data scale)	2	14.5	29.0	29.1	58.2	6.3	12.6	45.4	90.9	133.5	266.9	41.9	83.9	125.4	250.7	325.6	651.2
	3	9.6	28.8	22.2	66.5	5.2	15.6	18.2	54.5	47.6	142.7	30.7	92	91.2	273.6	172.3	516.9
	4	8.7	34.9	14.3	57.1	8.7	34.9	**3.5**	**13.9**	17.3	69.3	28.8	115.3	**60.3**	**241**	88.5	354.1
	5	8.3	41.4	11	54.8	6.9	34.5	3.2	15.8	**8.6**	**42.9**	26.7	133.3	52.3	261.3	40.7	203.3
	6	7.5	45.2	10.1	60.8	5	29.9	2.7	16.5	7.7	46	25.2	151.2	51.4	308.4	15.7	94.4
	7	4.5	31.4	**4.1**	**28.5**	7.7	53.9	2.1	14.8	7.2	50.6	24.8	173.3	46.5	325.4	**9.6**	**67.2**
	8	4.1	33.1	3.8	30.4	4	32.2	2.3	18.8	6.9	55.6	22.4	179.5	47.2	377.8	8.6	68.9
	9	3.9	35.2	3.7	33.2	4.7	42	2.1	18.9	6.4	57.6	20.9	187.9	41.2	370.5	8.4	75.2
	10	3.6	36.3	3.5	35.3	6.2	62	1.9	19.3	6.3	63	19.5	194.6	39.8	397.5	8.3	83.5
	11	3.6	39.6	3.5	38.3	5.5	60.6	1.9	21.4	5.9	65.2	18.6	204.4	40.2	442.3	8.4	92.5
	12	3.2	38.9	3.4	41	6.1	72.9	1.9	23.2	5.5	66.2	18.3	219.1	36.7	440.6	7.7	92.9
	Avg	8.2	35.1	14.3	47.3	6.3	38.4	18.5	37.1	49.2	105.2	29.6	151	82.8	337.6	124.9	258.7

6 Evaluation

For evaluation, we use 8 applications from Spark MLlib 2.4.0: *Alternating Least Squares* (ALS), *Bayesian Classification* (BAY), *Gradient Boosted Trees* (GBT), *K-means clustering* (KM), *Logistic Regression* (LR), *Principal Components Analysis* (PCA), *Random Forest Classifier* (RFC), and *Support Vector Machine* (SVM).

Sample Runs. For conducting sample runs and measuring the robustness of the extracted models for re-usability on clusters with different machine types, we use a single node – Intel Core i3-2370M CPU running at 4x 2.40 GHz, 3.8 GB DDR3 RAM and 388 GB disk. For each application, we carried out 3 runs on sample data size in the range of 0.1%–0.3% of the complete input data scale.

Actual Runs. We made all actual runs on a private 12-node cluster equipped with Intel Core i5 CPU running at 4x 2.90 GHz, 16 GB DDR3 RAM, 1 TB disk, and 1 GBit/s LAN. All nodes used in the experiments run Hadoop MapReduce 2.7, Spark 2.4.0, Java 8u102 and Apache YARN on top of HDFS. In our extended evaluation [2], we show that the models extracted from the sample runs are reusable for larger data scales (up to $18 \times 10^4\%$) and are useful to determine the bounds on resource-constrained clusters (i.e., the maximum data scale of an application that a cluster can run without eviction).

6.1 Selected Cluster Size

As mentioned in Sect. 1, we consider an optimal cluster size as the minimum number of machines that fit all cached datasets in memory without cache eviction. The Shadowed cells in Table 1 show the cluster sizes where no eviction occur, while the bold numbers indicate the cluster sizes selected by BLINK for each application. Table 1 shows that for all applications, BLINK selects the optimal cluster size (see the first shadowed cell of each application actual run in bold). To evaluate the efficiency of BLINK, we compare the sum of sample runs cost and actual run cost for the cluster size selected by BLINK to the average and worst costs of actual runs. Figure 6 shows that compared to the average and the worst costs, BLINK reduces the cost to 52.6% and 25.1%, respectively. In some cases, the worst cluster size (that leads to the highest cost) is a single machine due to lots of recomputations (SVM) and in other cases, it is the maximum cluster size because resources are wasted during data shuffling and processing of serial parts (RFC).

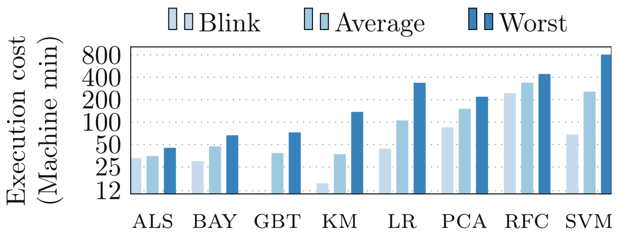


Fig. 6. BLINK cost optimization.

6.2 Overhead of Sample Runs

We compare the cost of sample runs with the cost of the corresponding actual run on optimal cluster configuration. Figure 7 shows that on average, sample runs cost 8.1% compared with the cost of the actual run on optimal cluster size. At worst, the overhead is 21.3% (ALS) while at best, it

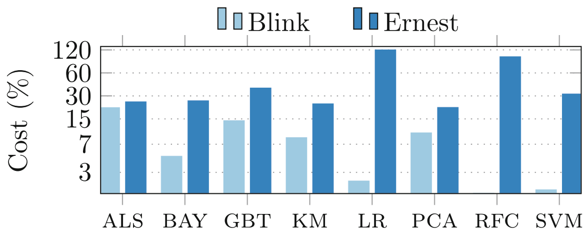


Fig. 7. Sample runs cost of BLINK and Ernest.

is 1.6% (RFC). Taking each sampling approach separately, we see that the average cost of sample runs of BLOCK-N is 2.7%, with a worst case of 5.1% (BAY) and a best case of 1.6% (RFC). For BLOCK-S, the average cost of sample runs is 13.3%, with a worst case of 21.3% (ALS) and a best case of 8.6% (KM). Altogether, BLOCK-S costs about 4.9× more than BLOCK-N. Nonetheless, the cost of BLOCK-S is still tolerable because we are comparing its cost with the costs of optimal actual runs. Note that all sample runs are carried out without changing any application parameter (e.g., number of iterations). Taking KM as a short-running application (3.5 min on the optimal cluster size; cf. Table 1), sample runs cost 8.6% of the cost of the actual run on the optimal cluster size. Hence, Blink is also effective for short-running applications.

Even though Ernest (cf. Sect. 2) predicts application runtime rather than cluster size, we compare the cost of its sample runs with the cost of those carried out by the sample runs manager (cf. Sect. 5.1). We carry out 7 sample runs, as recommended by Ernest's optimal experiment design, on 1–2 machines with sample datasets (1% –10% of the original data). The sample runs of Ernest cost 16.4× more than those of Blink (as depicted in Fig. 7).

7 Conclusion

Blink is an autonomous sampling-based framework that selects an optimal cluster size with the highest cost efficiency for running big data applications. The evaluation of Blink shows very good results in terms of selecting an optimal cluster size with high prediction accuracy.

Acknowledgement. This research was partially funded by the Thuringian Ministry for Economy, Science and Digital Society under the project thurAI and by the Carl-Zeiss-Stiftung under the project MemWerk.

References

1. Al-Sayeh, H., Jibril, M.A., Bin Saeed, M.W., Sattler, K.U.: SparkCAD: caching anomalies detector for spark applications. In: VLDB 2022 (2022)
2. Al-Sayeh, H., Jibril, M.A., Memishi, B., Sattler, K.U.: Blink: lightweight sample runs for cost optimization of big data applications. In: CoRR (2022). https://arxiv.org/abs/2207.02290
3. Al-Sayeh, H., Memishi, B., Jibril, M.A., Paradies, M., Sattler, K.U.: Juggler: autonomous cost optimization and performance prediction of big data applications. In: ACM SIGMOD 2022 (2022)
4. Al-Sayeh, H., Memishi, B., Paradies, M., Sattler, K.U.: Masha: sampling-based performance prediction of big data applications in resource-constrained clusters. In: VLDB DISPA 2020 (2020)
5. Alipourfard, O., Liu, H.H., Chen, J., Venkataraman, S., Yu, M., Zhang, M.: CherryPick: adaptively unearthing the best cloud configurations for big data analytics. In: USENIX NSDI 2017 (2017)

6. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS 1967 (Spring), Spring Joint Computer Conference (1967)
7. Chakaravarthy, V.T., Pandit, V., Sabharwal, Y.: Analysis of sampling techniques for association rule mining. In: ICDT 2009 (2009)
8. Hamidi, H., Mousavi, R.: Analysis and evaluation of a framework for sampling database in recommenders. J. Glob. Inf. Manag. **26**, 41–57 (2018)
9. Kunjir, M., Babu, S.: Black or white? How to develop an AutoTuner for memory-based analytics. In: ACM SIGMOD 2020 (2020)
10. Li, H., et al.: Detecting cache-related bugs in spark applications. In: ACM SIGSOFT 2020 (2020)
11. Meng, X., et al.: MLlib: machine learning in apache spark. J. Mach. Learn. Res. **17**, 1235–1241 (2016)
12. Perez, T.B.G., Zhou, X., Cheng, D.: Reference-distance eviction and prefetching for cache management in spark. In: ACM ICPP 2018 (2018)
13. Vavilapalli, V.K., et al.: Apache Hadoop YARN: yet another resource negotiator. In: ACM SoCC 2013 (2013)
14. Venkataraman, S., Yang, Z., Franklin, M., Recht, B., Stoica, I.: Ernest: efficient performance prediction for large-scale advanced analytics. In: USENIX NSDI 2016 (2016)
15. Xu, L., Li, M., Zhang, L., Butt, A.R., Wang, Y., Hu, Z.Z.: MEMTUNE: dynamic memory management for in-memory data analytic platforms. In: IEEE IPDPS 2016 (2016)
16. Yu, Y., Wang, W., Zhang, J., Letaief, K.B.: LRC: dependency-aware cache management for data analytics clusters. In: IEEE INFOCOM 2017 (2017)
17. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: USENIX NSDI 2012 (2012)
18. Zhu, Z., Shen, Q., Yang, Y., Wu, Z.: MCS: memory constraint strategy for unified memory manager in spark. In: IEEE ICPADS 2017 (2017)