Juggler: Autonomous Cost Optimization and Performance Prediction of Big Data Applications

Hani Al-Sayeh
TU Ilmenau, Germany
hani-bassam.al-sayeh@tu-ilmenau.de

Bunjamin Memishi
German Aerospace Center, Germany
bunjamin.memishi@dlr.de

Muhammad Attahir Jibril
TU Ilmenau, Germany
muhammad-attahir.jibril@tu-ilmenau.de

Marcus Paradies
German Aerospace Center, Germany
marcus.paradies@dlr.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Distributed in-memory processing frameworks accelerate iterative workloads by caching suitable datasets in memory rather than recomputing them in each iteration. Selecting *appropriate datasets* to cache as well as allocating a *suitable cluster configuration* for caching these datasets play a crucial role in achieving optimal performance. In practice, both are tedious, time-consuming tasks and are often neglected by end users, who are typically not aware of workload semantics, sizes of intermediate data, and cluster specification.

To address these problems, we present Juggler, an end-to-end framework, which autonomously selects appropriate datasets for caching and recommends a correspondingly suitable cluster configuration to end users, with the aim of achieving optimal execution time and cost. We evaluate Juggler on various iterative, real-world, machine learning applications. Compared with our baseline, Juggler reduces execution time to 25.1 % and cost to 58.1 %, on average, as a result of selecting suitable datasets for caching. It recommends optimal cluster configuration in 50 % of cases and near-to-optimal configuration in the remaining cases. Moreover, Juggler achieves an average performance prediction accuracy of 90 %.

CCS CONCEPTS

• **Information systems** → **Main memory engines**; **Autonomous database administration**.

KEYWORDS

database caching; cluster configuration; performance prediction

ACM Reference Format:
Hani Al-Sayeh, Bunjamin Memishi, Muhammad Attahir Jibril, Marcus Paradies, and Kai-Uwe Sattler. 2022. Juggler: Autonomous Cost Optimization and Performance Prediction of Big Data Applications. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517892

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517892

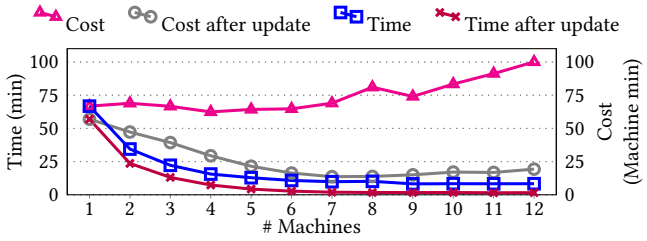

Figure 1: Selection of appropriate datasets for caching (LIR).

1 INTRODUCTION

With the recent advent of iterative machine learning applications, modern data-intensive systems such as Spark [68] maximize the performance efficiency of iterative workloads by caching crucial datasets in memory, to avoid recomputing or fetching them in each iteration from slower storage (e.g., disk or HDFS) [65].

Optimal selection of datasets to cache. Typically, application developers decide which datasets to cache based on their knowledge of the application's data flow dependencies [37, 55, 67]. But beforehand, they do not know the cluster environment (number of machines, total memory capacity), data size, application parameters, etc. It is the end users who configure the cluster environment and finally run the application. And usually, they handle the application binaries without the possibility to inspect the application logic and deduce an optimal selection of datasets to cache [32]. For specialized libraries, such as Spark MLlib [42], the decision of which datasets to cache is not even up to the application developers but instead chosen by the library developers. All these aspects lead to poor caching decisions, which ultimately result in low performance.

For example, the developers of *Linear Regression* (LIR) application in the HiBench benchmark [8, 26] do not cache any datasets. The dataset that is read in each iteration and thus should be cached is the potentially large original input dataset. To demonstrate the impact of caching on the overall performance, we modify the LIR application by caching the input dataset (35.9 GB) in memory and run the application on our private cluster (cf. § 7 for details) with different cluster configurations (1–12 machines). With this modification, the execution time and cost (#machines × time) decrease, on average, to 54.8 % and 34.3 % respectively, as shown in Figure 1.

Optimal selection of cluster configuration. To illustrate the impact of the cluster configuration (#machines), we select *Support Vector Machine* (SVM) application that contains a single *developer-cached dataset* (a dataset cached by HiBench developers). We run

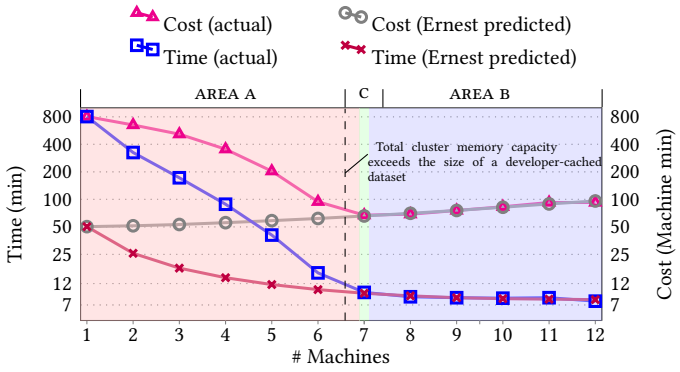

Figure 2: Selection of a suitable cluster configuration (SVM).

it on an input dataset of 59.5 GB on our private cluster with different cluster configurations (1–12 machines). For each cluster configuration, we measure the actual execution time and cost of the application runs. As depicted in Figure 2, we distinguish three areas:

- Area A (1–7 machines): Allocating more machines decreases both time and cost.
- Area B (7–12 machines): Allocating more machines decreases time but increases cost (time vs cost trade-off).
- Area C (7 machines): The junction of area A & B, where execution cost is at a minimum (optimal cluster configuration).

In area A, fewer machines mean less *cluster memory* (#Machines × memory per machine) for caching partitions of the crucial developer-cached dataset (35.7 GB) in SVM, considering each machine with processing power and 5.6 GB of memory for caching (cf. § 2.2). As a result, many partitions are evicted from memory and re-computed in every iteration, which is too costly. An in-depth look into a single iteration reveals that: (1) The percentage of data partitions evicted from cache in area A for 1 to 7 machines are 83 %, 65 %, 48 %, 30 %, 13 %, 8 % and 0 %, respectively. (2) On average, a task that recomputes an evicted partition takes 97× longer than a task that reads an already cached partition of equal size. In area C, cluster memory (39.2 GB) fits the whole 35.7 GB developer-cached dataset. In area B, adding more machines reduces the time of the parallel part of the processing pipeline but increases data transfer overhead. Moreover, the number of machines does not influence the time of the serial part [14]. All these result in increased cost.

Amdahl [14], Ousterhout et al. [47], Venkataraman et al. [59], and Alipourfard et al. [13], among others, study the overhead introduced by an increasing number of machines (area B). However, they do not consider the impact of memory limitation with regards to caching crucial datasets (area A). We make predictions for the same experiments using Ernest [59] and realize that its prediction is accurate only in area B. But since it does not factor in memory limitation, its prediction is inaccurate in area A. Even worse, Ernest predicts the cluster configuration with minimum cost to be a single machine. In reality, the actual cost on a single machine is higher than Ernest's prediction by 16× and higher than the optimal cost (on 7 machines) by 12×, as seen in Figure 2. In contrast to data-intensive applications (e.g., LIR and SVM), data processing is the dominant delaying factor in CPU-intensive applications that run for many hours or even days [59]. This explains why the benefit of caching some datasets becomes relatively low in such applications, and why Ernest predicts their performance accurately.

Cache eviction policies like LRU, LRC [65, 66], and MRD [49] tackle the cache limitation problem by prioritizing datasets to cache over those to evict. We apply them on the SVM experiments and do not realize any performance improvement because SVM contains a single developer-cached dataset. We further study all workloads in HiBench and realize that most of them contain at most one developer-cached dataset. For those that contain several developer-cached datasets, all these policies mostly make the same decision.

Optimizing execution cost is important for recurring applications that are executed repeatedly on various datasets with different user-selected parameters. In operational clusters, up to 60 % of analytical applications are recurring [29]. Some of them are highly repetitive [10, 19, 20]. In addition, some studies report that the majority of executed applications are short-running [52] (80 % of applications running on Yahoo, Facebook and Microsoft production clusters take less than 10 minutes [18]). While some experiments are required to construct performance optimization and prediction models, in the case of highly repetitive short-running applications, it is important to optimize and predict their performance without delays. To achieve this, an offline training approach is useful for two reasons. First, since the workloads are short-running, optimization and prediction models are constructed during the offline training and ready for usage in each run without delays. Second, since the applications are highly repetitive, the cost of the offline training can be amortized across many subsequent runs.

Contribution. Although the problem of cost optimization and performance prediction via fully automated caching of appropriate datasets and recommendation of cluster configurations is essential, we are not aware of any end-to-end approach that fully addresses it. By end-to-end here, we refer to a complete functional solution available to end users, independent of any human interaction, and other external components. Recent work mostly focus on contributions related to performance prediction [13, 59], cost optimization [27, 56], caching of datasets [49, 62, 65], or cluster configuration [31, 32, 38, 39, 48]. Combining and advancing all these techniques in a unified approach remains an open challenge.

The scope of this paper are iterative data-intensive applications, which are run repetitively on cloud platforms [1, 3, 4] with various application parameters (e.g., CODE-DE [57]) in a black-box manner through a pay-as-you-go pricing model [22]. The main goal is to efficiently run these applications with minimal execution cost (area C) by selecting appropriate datasets to cache in memory and recommending suitable cluster configurations for caching them.

In this paper, we present JUGGLER, an autonomous training-based framework, which selects appropriate datasets for caching and recommends optimal cluster configuration to cache the datasets. JUGGLER makes recommendations to end users based on performance prediction and performance-cost trade-off. It achieves these via four stages in order, each of which addresses a particular sub-problem. Below, we formulate the sub-problems as questions:

(1) Considering computation time, size and number of computations of each dataset, *which datasets should be cached?*
(2) Considering different application parameters selected by end users, *what is their impact on the size of each cached dataset?*

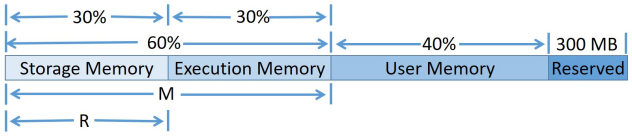

Figure 3: Spark memory layout. M indicates Spark unified memory, and R indicates storage memory for caching.

(3) Considering the variation in memory footprint of different applications, *what is the optimal cluster configuration (area C) for caching the datasets without eviction?*
(4) Considering a selection of optimal cluster configuration, *what is the estimated execution time and cost?*

We evaluate Juggler using five real-world machine learning applications. Juggler selects suitable datasets to cache, which reduces execution time and cost, on average, to 25.1 % and 58.1 % respectively, compared to developer-cached datasets in our baseline (HiBench). In 50 % of cases, Juggler recommends an optimal cluster configuration, while the remaining cases are near-to-optimal. Juggler achieves an average performance prediction accuracy of 90 %.

2 BACKGROUND

We discuss the execution model and memory management of Spark, a distributed in-memory processing framework use case.

2.1 Execution model

Spark runs applications on a set of *executor* processes that execute various operations, in parallel, on a collection of partitioned data called Resilient Distributed Dataset or *RDD*. RDD is the primary abstraction for distributed data processing in Spark [67]. Although RDDs are immutable, a class of operations called *transformations* (e.g., map, filter) create new RDDs from existing ones. Another class of operations called *actions* (e.g., collect, count) return a value to a central process (*driver*) after running a computation over RDDs.

An execution in Spark begins with the creation of a logical plan for the set of transformation and action operators in the application. The application level is the highest level of computation and consists of one or more sequential *jobs*. Each action triggers the launch of a job. Thus, a job comprises of a single action and a sequence of the transformations preceding it, represented by a *DAG* of transformations. When a transformation is applied on a (parent) RDD, a new (child) RDD is created. The logical plan entails the parent-child dependency between RDDs, by way of a lineage or DAG. Each child RDD in the DAG points back to its parent, representing how Spark will run the transformation. It should be noted that the direction of edges between RDDs, which denotes the RDD dependency in the logical plan, is opposite that of the dataflow graph.

The DAG of transformations is constructed starting from an action. Then, parent RDDs are constructed in a backward fashion towards the root RDDs that either depend on no other RDDs or reference cached data. A transformation can be either *narrow* or *wide*. Spark stages are created by splitting the DAG at shuffle boundaries (wide transformation), whereby the scheduler *pipelines* each group of narrow transformations (e.g., map, filter) into a stage. Data shuffling between any two consecutive stages consists of two phases, namely, *Shuffle Write* (last part of the first stage) and *Shuffle Read* (first part of the second stage). Thus, a job consists of one or more stages, and each stage comprises *tasks* that perform the same computation on different data partitions. More details about Spark's execution model have been presented previously [24, 41, 47, 60, 69].

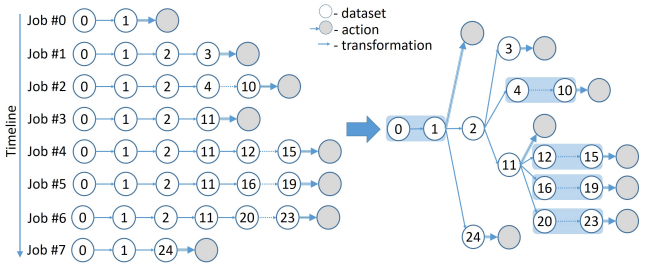

Figure 4: Merging DAGs: *Logistic Regression* use case.

2.2 Memory management

In Spark, memory is split into regions as depicted in Figure 3. In this paper, we focus on the execution and the storage memory regions, respectively used for computation and caching datasets [70]. Both regions share the same memory space as though they were a unified region *M* such that when execution memory is not used, it can be utilized for caching, and vice versa. There is a minimum storage space *R* below which cached data may not be evicted. In other words, in each executor, at least R and at most M can be utilized to cache datasets. The sizes of M and R are configurable [7]. When the limit of memory is reached, partitions of least recently used cached RDDs are evicted. This occurs when the size of cached partitions either exceeds R while the execution memory is utilized or exceeds M. A developer can *unpersist* a cached RDD when it is not needed anymore. The memory space can then be utilized to cache other useful RDDs that would have otherwise been evicted. For the remainder of the paper, we refer to RDDs simply as *datasets*.

In the experiments depicted in Figure 2, each machine has 12 GB of RAM. In this case, M = (12 GB - 300 MB) × 60 % = 7.02 GB and R = M × 50 % = 3.51 GB. In SVM, for example, 20.2 % of M is utilized for execution and the remaining 79.8 % (i.e., 5.6 GB per machine) can be utilized for caching (cf. § 5.3 for details).

3 DATASET METRICS

We discuss the derivation of dataset metrics, namely the number of times a dataset is computed, its size and its computation time.

3.1 Number of computations

An application consists of one or more sequential jobs, each of which has its DAG of transformations. As DAGs may have many transformations in common, we merge all the DAGs to represent an application in a single DAG of operators. The number of times to compute a dataset is equal to the number of its leaves in the resulting DAG. For example, after merging all the DAGs in the *Logistic Regression* (LOR) application (cf. Figure 4), the number of times to compute datasets D_1 and D_2 are 8 and 6, respectively. For the remainder of the paper, we refer to datasets computed more than once as *intermediate datasets*. The computation of the datasets can be traced in a depth-first traversal order starting from D_0.

3.2 Size

The size of a dataset equals the sum of the sizes of all its partitions.

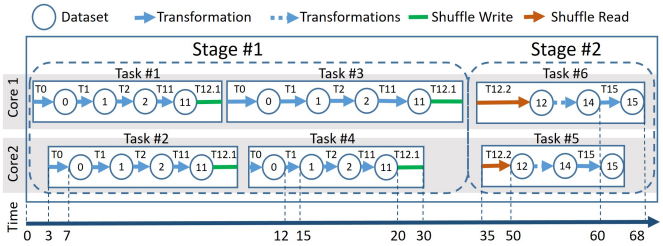

Figure 5: The DAG of transformations in the iterative job of *Logistic Regression* application. The job consists of two stages, each with several tasks running on two cores.

3.3 Computation time

In the following, we introduce a novel operator-level execution time model to calculate the dataset computation time. The execution time ET_{job} of a job is calculated as:

$$ET_{\text{job}} = \sum_{i=1}^{n} ET_{\text{T}_i} \tag{1}$$

where n is the number of transformations in the job DAG and ET_{T_i} is the execution time of the i^{th} transformation. Figure 5 depicts the DAG of transformations in an iterative job of *Logistic Regression* (job #4 in Figure 4). The job execution time equals $ET_{\text{T}_0} + .. + ET_{\text{T}_2} + ET_{\text{T}_{11}} + .. + ET_{\text{T}_{15}}$. If the DAG contains joins, some transformations run in parallel. In this case, n is the number of transformations on the path in the DAG having the longest execution time. As mentioned earlier, a job DAG consists of narrow transformations (e.g., T_1) and wide transformations (e.g., T_{12}).

Narrow transformations. To calculate the execution time ENT_{ij} of a narrow transformation i in task j, we distinguish three cases:

(1) If the transformation is the first in the task, the execution time is the difference between the transformation's finish timestamp and the task's start timestamp. In Figure 5, the execution time of T_0 in the second task ENT_{02} is 4.
(2) If the transformation is the last in the task, the execution time is the difference between the task's finish timestamp and the transformation's start timestamp. In Figure 5, the execution time of T_{15} in the sixth task $ENT_{(15)6}$ is 8.
(3) If the transformation is between two narrow transformations, the execution time is the difference between the respective transformation's finish and start timestamps. In Figure 5, the execution time of T_1 in the fourth task ENT_{14} is 3.

Therefore, the execution time of a narrow transformation (cf. Equation 1) is calculated as:

$$ET_{\text{T}_i} = \frac{\sum_{j=1}^{N_{\text{tasks}}} ENT_{ij}}{N_{\text{tasks}}} \times N_{\text{waves}} \tag{2}$$

where N_{tasks} is the total number of tasks in the stage, i is the index of the transformation in the DAG and N_{waves} is the average number of tasks in a stage that run sequentially in the same core (i.e., $\lceil \frac{\#tasks}{\#cores} \rceil$). The division over N_{tasks} is for averaging, to handle variance between skewed tasks (e.g., Task #3 and Task #4 in Figure 5). In Figure 5, the execution time of T_1 is:

$ET_{\text{T}_1} = ((ENT_{11} + ENT_{12} + ENT_{13} + ENT_{14})/4) \times 2$

Wide transformations. In data shuffling between two consecutive stages, Shuffle Write takes place in the first stage and Shuffle Read takes place in the second stage. Therefore, we consider a wide transformation as a pair of two consecutive narrow transformations and, thus, the execution time ET_{T_i} of a wide transformation i is:

$$ET_{\text{T}_i} = ET_{\text{T}_{i.1}} + ET_{\text{T}_{i.2}} \tag{3}$$

where $ET_{\text{T}_{i.1}}$ and $ET_{\text{T}_{i.2}}$ are the execution times of Shuffle Write and Shuffle Read narrow transformations respectively. As Shuffle Write (e.g., $T_{12.1}$) is the last transformation in the stage (like T_{15}), we consider it as the second of the three cases of narrow transformations. Similarly, since Shuffle Read (e.g., $T_{12.2}$) is the first transformation in the task (like T_0), we consider it as the first of the three cases of narrow transformations. In Figure 5, the execution time of T_{12} is:

$$ET_{\text{T}_{12}} = ET_{\text{T}_{12.1}} + ET_{\text{T}_{12.2}}$$
$$ET_{\text{T}_{12.1}} = ((ENT_{(12.1)1} + \ldots + ENT_{(12.1)4})/4) \times 2$$
$$ET_{\text{T}_{12.2}} = ((ENT_{(12.2)5} + ENT_{(12.2)6})/2) \times 1$$

Discussion. To calculate the metrics of each dataset, especially the size (cf. § 3.2) and the computation time (cf. § 3.3), it is necessary to obtain low-level runtime data. Some data processing engines such as Spark provide some of the required runtime data like the start and end timestamps of each task [5]. However, other runtime data are missing (e.g., the start and end timestamps of each transformation in a task, the size of each dataset partition). Therefore, in our use case, we modify Spark by adding custom instrumentations that collect the missing runtime data. We explain this in the next section.

4 INSTRUMENTATION

A task in Spark processes a dedicated data partition by applying a sequence of narrow transformations. For each narrow transformation in a task, our goal is to precisely know how much processing time it takes and the size of its resulting data partition. Because Spark by default does not provide the respective runtime data, we update the source code of Spark. This results into a modified version, Spark *Instrumentation* (SPARK$_I$), that automatically injects a special-purpose transformation (*mapPartitionsWithIndex [6]*) between each consecutive pair of transformations. Each injected transformation profiles timestamps and partition sizes, and produces an *instrumentation dataset* that is a replica of the dataset produced by the preceding transformation. In each task, the profiling transformation stores the runtime data in *TaskContext* and when the task finishes, its corresponding low-level runtime data is sent to a central profiling database. Finally, when the application ends, the (application, job, stage and task) runtime data is copied to the database.

Figure 6 illustrates how profiling transformations are added automatically. Each dataset (e.g., D_0) is followed by an automatically injected profiling transformation during its construction, which generates an instrumentation dataset (e.g., D_{i_0}) that is a copy of the dataset. A transformation on D_0 (e.g., filter) produces a new dataset D_1. Before creating a parent-child dependency from D_1 to D_0, a check is performed whether an instrumentation dataset is associated with D_0. If so, a dependency is created from D_1 to the instrumentation dataset, D_{i_0}. Otherwise, a dependency is created from D_1 to D_0. The process continues up to the creation of the last dataset. Figure 7 shows task internals of the first stage of the iterative job in *Logistic Regression* proceeding from instrumentations.

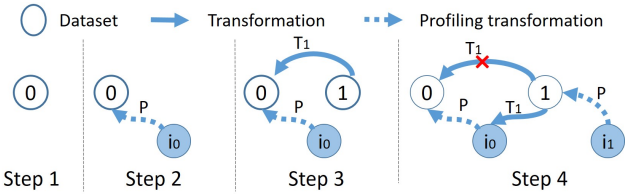

Figure 6: Spark$_i$ steps in adding profiling transformations (*P*). T_1 indicates regular transformations.

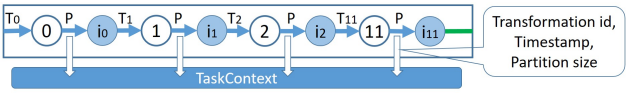

Figure 7: Task internals in the first stage of the iterative job in *Logistic Regression* proceeding from instrumentations. *P* indicates profiling transformation.

Note that adding lightweight instrumentations to Spark allows us to get low-level runtime data from application binaries, without the need to access the source code. This solution allows for obtaining metrics of all datasets, including those that are not accessible from an application layer (e.g., RDDs in Spark MLlib).

5 JUGGLER

In this section, we describe Juggler (cf. Figure 8), an autonomous framework that **optimizes** and **predicts** the execution time and cost of big data applications. Firstly, Juggler *optimizes* the execution cost by selecting appropriate datasets for caching in an application and recommending the optimal cluster configuration for caching them. Juggler presents the combination of datasets to cache in form of one or more schedules. A schedule is an ordered list of datasets to be cached. Juggler achieves this by conducting an *offline training* in three sequential stages: (1) Hotspot Detection: Identifying datasets with the potential to improve performance when cached, (2) Parameter Calibration: Building models to predict the sizes of those potential datasets with respect to user-defined application parameters, and (3) Memory Calibration: Deriving a calibration factor for predicting exactly how many machines are required to cache the datasets. Secondly, Juggler *predicts* the execution time by conducting a fourth *offline training* stage - (4) Building execution time model(s): Constructing execution time models with which to predict the execution time and, ultimately, cost.

5.1 Hotspot detection

This stage identifies *hotspots*, i.e., appropriate intermediate datasets for caching, based on their computation time, size, and the number of times they are computed. Juggler conducts a single sample run of the application on Spark$_i$ to collect these metrics for each dataset (d). It keeps the training overhead to a minimum by running the application on a small data sample and with few iterations.

The hotspots detection algorithm is shown in Algorithm 1. Juggler begins by making a list of all intermediate datasets (Line 1). It initializes an empty current schedule and an empty list of schedules (Line 2). Then for each dataset in the dataset list, it calculates the computational overhead saved by caching it (Line 4–9). We term this the *benefit* of a dataset. As the computation of a dataset triggers the computation of all its parent datasets, caching a dataset that is computed *n* times saves the time of recomputing it and each

Algorithm 1: Hotspot detection.

Input : Dependencies, computation time (ET), number of computations (n) and size of each dataset
Output : List of schedules

1 datasets ⟵ list of all datasets with $n > 1$;
2 schedules ⟵ ∅; schedule_{curr} ⟵ ∅;
3 **while** *datasets* ≠ ∅ **do**
4 **foreach** *dataset* D_i *in datasets* **do**
5 $\text{benefit}_{D_i} = (n_{D_i} - 1) \times ET_{T_i}$;
6 **foreach** *parent dataset* D_p *of* D_i **do**
7 **if** D_p *is in* schedule_{curr} **then**
8 break;
9 benefit_{D_i} += $(n_{D_i} - 1) \times ET_{T_p}$;
10 $\text{BCR}_{D_i} = \text{benefit}_{D_i} / \text{size}_{D_i}$;
11 D_{max} ⟵ dataset with highest BCR;
12 **while** D_{max} *is a single child of any of the datasets in* schedule_{curr} **do**
13 D_{max} ⟵ dataset with next highest BCR;
14 schedule_{curr} += D_{max};
15 datasets -= D_{max};
16 Re-evaluation ⟵ false;
17 **if** D_{prev} *is a child of* D_{max} **then**
18 datasets += D_{prev};
19 schedule_{curr} -= D_{prev};
20 Re-evaluation = true;
21 **foreach** *parent dataset* D_p *of* D_{max} **do**
22 **if** D_p *is in datasets* **then**
23 n_p -= $(n_{D_{max}} - 1)$;
24 **else if** D_p *is in* schedule_{curr} **then**
25 before caching D_{max}, unpersist D_p if it is not computed for other child datasets anymore;
26 D_{prev} ⟵ D_{max};
27 **if** *Re-evaluation == true* **then**
28 continue;
29 schedules += schedule_{curr};
30 **foreach** *schedules i and j in schedules* **do**
31 **if** schedule_i *and* schedule_i *have same cost* **then**
32 discard the schedule with lower benefit;
33 **return** *schedules*

of its parents (if any) $n-1$ times. Hence, for a dataset with ID i (D_i), its benefit, B_i, is:

$$B_i = (n_i - 1) \times (ET_{T_i} + \sum_{j}^{P} ET_{T_j}) \quad (4)$$

where n_i is the number of times the dataset is computed, ET_{T_i} is its computation time (§ 3.3) and P is the list of its parent datasets. Note that a dataset D_i is computed by transformation T_i.

Accompanying the benefit of caching a dataset is the cost of caching it, in terms of memory footprint. To account for this, we introduce the benefit-cost ratio, *BCR*, which is the ratio of a dataset's benefit to its size. After calculating the bcr of each dataset in the dataset list (Line 10), Juggler selects the dataset with the highest

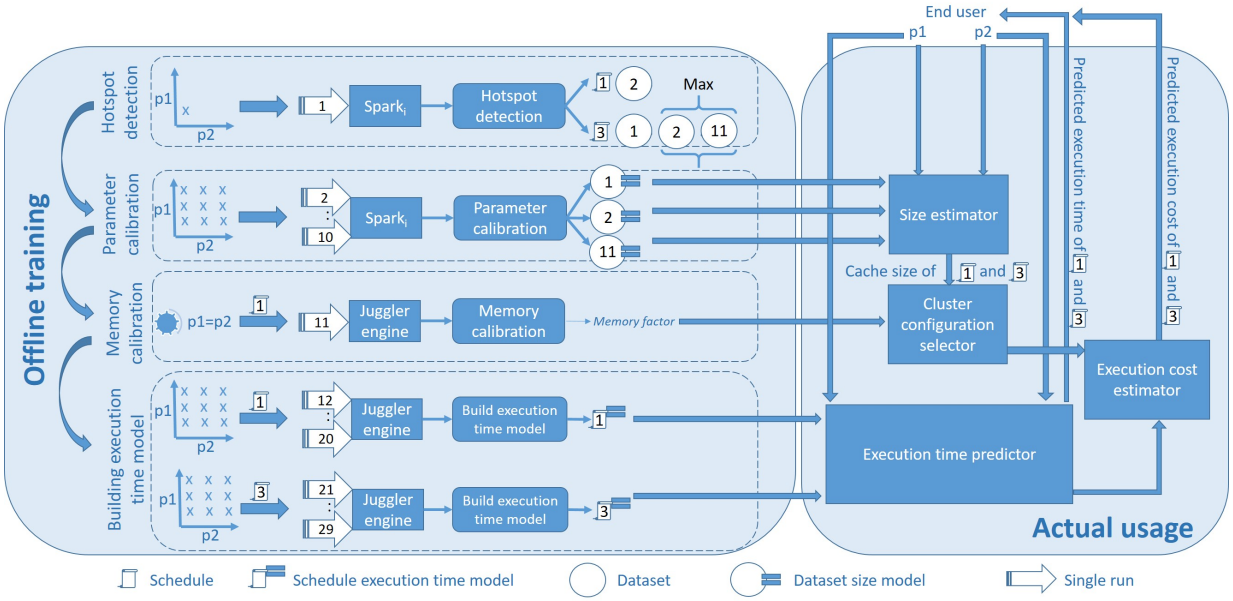

Figure 8: Overview of Juggler with illustrative example (*Logistic Regression*).

BCR (Line 11). A single-child dataset is not added to a SCHEDULE that already contains its parent (like D_0 and D_1 highlighted in Figure 4). Therefore, if the selected dataset is a single-child of any dataset currently in the SCHEDULE, it is not added to the SCHEDULE. Instead, Juggler selects the dataset with the next highest BCR (Line 12–13).

After that, Juggler removes the dataset with the highest BCR from the dataset list and adds it to the current SCHEDULE (Line 14–15). Then it checks if there is a need for *re-evaluation*: whether the last dataset in the SCHEDULE is a child of the dataset with the highest BCR. If so, Juggler performs a *re-evaluation* by removing the last dataset from the SCHEDULE and adding it back to the dataset list (Line 16–20).

When a dataset is cached, the benefit of each of its parents decreases because caching it decreases the number of computations of each of its parents. As for its children, caching the dataset also decreases their benefits even though their number of computations does not change. The reason is that caching them afterwards does not save computations of the dataset (and its parents) anymore since it was already cached (Line 7–8). Therefore, upon adding a dataset to a SCHEDULE, Juggler updates the number of computations of each of its parents in the dataset list (Line 21–23). Then it adds the completed SCHEDULE to the list of SCHEDULES (Line 29).

Juggler performs this process iteratively until the dataset list becomes empty. Juggler generates SCHEDULES incrementally. As a result, the first SCHEDULE has the lowest benefit and requires the least memory budget. By caching more datasets in subsequent SCHEDULES, both the benefit and memory budget increase.

A strategy to optimize the memory budget of SCHEDULES with more than one dataset is to *unpersist* a cached dataset directly before caching the succeeding dataset in the SCHEDULE (Line 24–25). However, a cached dataset is unpersisted only if the dataset that follows it in the SCHEDULE is its child in all remaining jobs (i.e., along the lineage graph of all DAGs). For two consecutive datasets, unpersisting the first dataset decreases the SCHEDULE memory budget by the size of the smaller of the two datasets. Note that as a result of unpersisting, some SCHEDULES may end up having equal cost. In such cases, Juggler keeps the SCHEDULE with the highest caching benefits and discards the others (Line 30–32).

Example. We use *Logistic Regression* (the merged DAG of Figure 4) as an example to illustrate the workings of the hotspot detection. In the tables below, execution time is in ms and cost (size) is in MB.

	#Computations	Execution time	Benefit	Cost	BCR
D_0	8	2700	18,900	76.351	247.54
D_1	8	10	18,970	76.347	248.47
D_2	**6**	**14**	13,620	**45.961**	**296.34**
D_{11}	4	40	8,292	45.975	180.36

D_0, D_1, D_2 and D_{11} are computed more than once. Caching D_0, for example, saves its recomputation 7×, which results in a benefit of 7×2,700 (18,900). The benefit of caching D_{11} is 3×(2,700+10+14+40). Juggler calculates the BCR for each dataset and adds the dataset with the highest BCR (i.e., D_2) to the first SCHEDULE. Next, Juggler generates the second SCHEDULE starting from the datasets in the previous SCHEDULE. In this case therefore, the second SCHEDULE starts with caching D_2. After this, Juggler updates the number of recomputations, benefits and BCR of the remaining datasets; and then selects the dataset with the highest BCR.

	#Calls	Execution time	Benefit	Cost	BCR
D_0	3	2700	5,400	76.351	70.73
D_1	**3**	**10**	**5420**	**76.347**	**70.99**
D_{11}	4	40	120	45.975	2.61

After caching D_2, the benefit of caching D_{11} reduces to 3×40 because caching D_{11} will not save computation cost of D_2 or any of its parents anymore. Ultimately, D_1 has the highest BCR. However, D_2 cannot be cached in the second SCHEDULE before D_1 because D_1 is a parent of D_2. Consequently, Juggler re-evaluates, starting with D_1. After caching D_1, it updates the benefits and selects the dataset with the highest BCR, i.e., D_{11}, as shown below:

	#Calls	Execution time	Benefit	Cost	BCR
D_2	6	14	70	45.961	1.52
D_{11}	**4**	**40**	**162**	**45.975**	**3.52**

Therefore, the first SCHEDULE is caching only D_2, whereas the second SCHEDULE is caching D_1, followed by D_{11}. Similarly, JUGGLER generates the third SCHEDULE by incrementing the datasets in the second SCHEDULE. After applying the same procedure, D_2 is added to the third SCHEDULE. Now, there remains no dataset to be cached. And that completes the SCHEDULES.

Note that JUGGLER does not unpersist D_1 in the second SCHEDULE after D_{11} is computed. This is due to the last job in Figure 4 where D_1 is computed along another DAG that does not contain D_{11}. Because D_{11} is present as the child of D_2 in all the remaining DAGs, JUGGLER unpersists D_2 in the third SCHEDULE just before caching D_{11}. As a result, the cost of the third SCHEDULE reduces from being the sum of the sizes of D_1, D_2 and D_{11}, to being the sum of the size of D_1 and the maximum size between D_2 and D_{11}.

This stage ends with JUGGLER having identified SCHEDULES as shown in the table below. As the second and third SCHEDULES have the same cost, JUGGLER only keeps the third SCHEDULE because it gives more benefit by caching an additional dataset.

SCHEDULE	ID	Unpersist	Cost
1	2	-	45.961
~~2~~	~~1, 11~~	~~-~~	~~122.322~~
3	1, 2, 11	2	122.322

Discussion. At this point, JUGGLER has answered the question: *Which datasets are most appropriate for caching?* With regards to the *Logistic Regression* application, if the third SCHEDULE is selected, more datasets are cached in memory, which implies more machines are required, and lower execution time is expected (compared with the first SCHEDULE). Therein lies a trade-off. And since the users are not aware of the details of caching, they need to be provided with a measure in terms of application execution time and cost, to be able to select a suitable SCHEDULE. To this end, JUGGLER builds an execution time prediction model for each SCHEDULE (cf. § 5.4). The goal of the next stage is to predict the sizes of datasets for any user-selected application parameters in every SCHEDULE.

5.2 Parameter calibration

Machine learning applications broadly have two application parameters: *examples* (P1) and *features* (P2). Following the hotspot detection stage, JUGGLER predicts the size of each dataset included in the SCHEDULES based on the application parameters. For each parameter, JUGGLER constructs an array of parameter values to be used during the training phase. We consider $E = \{e_0, ..., e_n\}$ and $F = \{f_0, ..., f_n\}$ to be the training arrays for P1 and P2 respectively. Then it carries out the training phase using SPARK$_i$ by running a full-factorial set of experiments, which includes all parameter combinations (n^m experiments, where m is the number of application parameters). Finally, for each dataset in the identified SCHEDULES, it trains a predefined list of linear models and selects the model with the least error. In our running example with *Logistic Regression*, JUGGLER constructs a size prediction model for D_1, D_2, and D_{11}.

Our experiments have shown that a training array of size $n = 3$ works for all cases. And because there are only two application parameters, the training overhead caused by running a full-factorial design of experiments is negligible.

JUGGLER applies *cross validation* to determine the error of each model. It does this by keeping each point among the total training experiments, in turn, as a test experiment and fitting the model with the remaining $n^m - 1$ experiments. All error instances are averaged to get the model error. JUGGLER selects the model with the least error and trains it using all n^m training experiments. Among multiple models evaluated by JUGGLER, our experiments on more than 300 different datasets show that all datasets fit into one of the following models, even though JUGGLER evaluates other models:

$$D_{size} = \theta_0 \times e \times f$$
$$D_{size} = \theta_0 \times e + \theta_1 \times e \times f$$
$$D_{size} = \theta_0 \times f + \theta_1 \times e \times f$$
$$D_{size} = \theta_0 + \theta_1 \times e + \theta_2 \times e \times f$$

where e is the number of examples and f is the number of features. To train models, JUGGLER uses *curve_fit* solver [2] with enforced positive bounds, which avoids negative coefficients. Lastly, JUGGLER sums up the predicted sizes of all cached datasets to obtain the total size of each respective SCHEDULE:

$$\text{SCHEDULE}_{size} = \sum_{CachedDs} D_{size}$$

where *CachedDs* is the list of cached datasets in the SCHEDULE. In case of unpersist, *CachedDs* does not include datasets with minimal sizes. For example $\text{SCHEDULE1}_{size} = D2_{size}$ and $\text{SCHEDULE3}_{size} = D1_{size} + D11_{size}$. Although JUGGLER is not limited to ML applications, new models might be needed if new classes of parameters are introduced, such as *#vertices* and *#edges* in graphs and *#rows* and *#columns* in *SQL* applications, coupled with other considerations like selectivity and cardinality estimation [11, 30, 34, 36, 46, 64].

Discussion. By the end of the parameter calibration stage, JUGGLER has answered the question: Considering the impact of selected application parameters, *what is the size of each cached dataset?* However, since every application has its unique characteristics with regards to execution memory usage (cf. § 2.2), JUGGLER cannot yet precisely predict how many machines are needed. For instance, in a certain scenario, JUGGLER recommends to avoid running an application with less than 3 machines (otherwise, M (cf. § 2.2) will not be enough, leading to cache eviction) or more than 6 machines (otherwise, R (cf. § 2.2) will be more than needed, resulting in over-allocation of resources). To precisely predict how many machines are required to cache all datasets in the recommended SCHEDULES, JUGGLER carries out a single training run per application, for memory calibration.

5.3 Memory calibration

In this stage, JUGGLER picks the first SCHEDULE and chooses values for P1 and P2 such that the size of the SCHEDULE equals the maximum heap memory fraction that can be occupied by the cached data (M in § 2.2). To apply the SCHEDULE, the application is run with the selected values for P1 and P2 on the execution engine of JUGGLER (*JUGGLER engine* in short), which is a modified version of Spark that overwrites the developer-cached datasets with the recommended SCHEDULE by injecting cache/unpersist instructions to the DAG. Hypothetically, if all partitions of cached datasets remain in memory, then the application does not utilize the execution memory at all. If 50 % of partitions are fetched from memory, the application fully utilizes the execution memory. Thus, JUGGLER derives the *memory factor* (from 0.5 to 1) as a ratio between the number of non-evicted

partitions to the total number of partitions and predicts the actual amount of memory for caching per machine as:

$$MemoryForCaching_{PerMachine} = M \times memory\ factor \quad (5)$$

where M is the unified memory region (cf. Figure 3). Juggler then predicts the optimal cluster configuration for each schedule as:

$$\#Machines = \left\lceil \frac{\text{SCHEDULE}_{size}}{MemoryForCaching_{PerMachine}} \right\rceil \quad (6)$$

Discussion. As the last optimization stage, Juggler has answered the question: *What is the optimal cluster configuration for caching datasets without eviction?* The previous stages are conducted once per application and the extracted schedules and models can be re-used on any cluster environment (different types of machines) without any changes. In each cluster environment, Juggler recommends the optimal number of machines instantly without carrying out any additional experiments. Despite variances in computing power and network capacity across different types of machines, for cost optimization, Juggler considers only the memory size per machine. That way, M is calculated (cf. § 2.2) and used to calculate $MemoryForCaching_{PerMachine}$ (cf. Equation 5) and the recommended number of machines (cf. Equation 6).

The overhead caused by the optimization stages is negligible (cf. § 7.6) because all the experiments in these stages are conducted (1) only once per application (2) on a single machine (3) using tiny datasets and (4) their extracted models are re-usable. We refer to the constructed models of these stages as *optimization models*.

5.4 Building execution time model(s)

The goal of this stage is to predict the execution time for each schedule with respect to the selected application parameters P1 and P2. Similar to the training experiments carried out during the parameter calibration (§ 5.2), Juggler performs full-factorial training experiments of E and F for each schedule using Juggler engine. For each experiment, the application is run on the recommended number of machines, as derived in the memory calibration stage. Juggler fits the execution times to a set of predefined linear models and obtains the coefficients of the model with the least error – similar to the model fitting, cross-validation and model selection in the parameter calibration stage. The result of this stage is an execution time prediction model expressed in terms of P1 and P2.

Our experiments show that the execution time of all schedules fit into one of the following models, even though Juggler evaluates other models:

$$Execution\ time = \theta_0 \times e \times f$$
$$Execution\ time = \theta_0 + \theta_1 \times e \times f$$
$$Execution\ time = \theta_0 \times f + \theta_1 \times e \times f$$
$$Execution\ time = \theta_0 \times f^2 + \theta_1 \times e \times f$$

where e and f are the number of examples and features, respectively.

Discussion. With this stage, Juggler has answered the question: *What is the estimated time and cost* of running the application in an optimal cluster configuration? The execution time model of each schedule takes P1 and P2 as parameters. In other words, the execution time models (namely *prediction models*) are used to correlate application parameters with execution time. The number of machines is not included in this model because the model aims to predict the execution time if the optimal number of machines for the schedule is selected. The type of machines is also not included. Therefore, unlike optimization models, prediction models cannot be re-used as is on different clusters. Additional models would be required on top of them (cf. § 6) or, in the worst case, this stage is re-conducted if the cluster environment changes.

Even though the offline training is conducted in the previous four *sequential* stages, it is important to highlight that Juggler is a modular framework whose stages/components are independent such that if an error takes place in a stage, it would not be amplified in the following stages. For example, if the hotspot detector selected an inappropriate dataset (i.e., inappropriate schedule), Juggler would predict its size, recommend suitable cluster configuration to cache it, and predict the execution time of the schedule.

5.5 Actual usage

Figure 8 depicts the end-to-end process, where an end user runs an application. Initially, the end user selects application parameters (examples and features). Based on these parameters, the *size estimator* predicts the size of cached datasets in each schedule using the dataset size models extracted in the parameter calibration stage (§ 5.2). Afterwards, the *cluster configuration selector* recommends the number of machines required for caching each schedule based on the cumulative size of its respective datasets as predicted by the *size estimator*, and the respective memory factor extracted in the memory calibration stage (§ 5.3). Next, the *execution time predictor* predicts the execution time for each schedule using the extracted execution time models (§ 5.4) with respect to the selected application parameters. After that, the *execution cost estimator* predicts the execution cost of each schedule from its respective recommended cluster configuration and predicted execution time. The cost is currently expressed as $\#Machines \times Time$. But this can be replaced with other pricing models [45]. Then the *execution time predictor* and the *execution cost estimator* return the execution time and cost of each schedule respectively to the end user. All these values are calculated at once without additional experiments since all models are already available from the offline training. Finally, the end user (or scheduler) selects a suitable schedule that meets predefined constraints such as time deadline and cost budget. However, Juggler does not offer a schedule if another one is faster and cheaper.

6 DISCUSSION

In this section, we discuss changes in environment where rebuilding the execution time models can be avoided. The optimization models can always be reused independently of the changes.

6.1 Number of iterations

Iterative applications take the number of iterations as a parameter.

Optimization. Selecting datasets for caching and recommending suitable cluster configuration to cache them are independent of the number of iterations because it does not influence the size of the cached datasets. Therefore, Juggler optimizes the cost of an application run independently of the number of iterations.

Prediction. Our experiments demonstrate that the number of iterations influences Juggler's prediction models (cf. § 5.4). To add

this parameter to the execution time model, another (linear) execution time model can be extracted from the main execution time model by carrying out additional experiments. However, in many cases, the number of iterations is not known beforehand. In such cases, the application runs until either a predefined condition is met or the maximum number of iterations is reached. Consequently, predicting the number of iterations is challenging since it is influenced by various application-level aspects, such as dataset characteristics and application semantics (convergence function). This issue is already addressed by [50]. Moreover, some hyper-parameters, like the number of clusters in K-MEANS, influence the number of iterations and the execution time of each iteration [11]. Similar to the number of iterations, these hyper-parameters are to be considered when JUGGLER builds the execution time model (cf. § 5.4).

6.2 Types of machines (VMs)

Public cloud providers offer various types of instances (Azure and AWS provide 146 and 133 different instance types respectively [38]).

Optimization. Selecting datasets for caching and predicting their sizes are not influenced by the hosting machine. Rather, it is the cluster configuration that depends on the size of allocated memory per machine (Equations 5 and 6), which is known in advance. Thus, JUGGLER does not need additional experiments to recommend the cluster configuration in different environments. In our evaluation (§ 7.1), JUGGLER constructs optimization models in one environment and recommends the cluster configuration for another one.

Prediction. The execution time of a SCHEDULE varies between different types of machines [59]. To avoid conducting similar experiments (§ 5.4) on all types of machines, additional experiments on a few of them are required to build a new execution time model on top of that of JUGGLER. CherryPick [13] builds such a model by predicting the execution time of application runs on many instance families without carrying out experiments in all of them. It achieves this by defining a set of features for machine types (e.g., CPU count and speed, the size of memory, network bandwidth, etc.) and leveraging Bayesian Optimization [43] using an adaptive search methodology to reduce the number of experiments.

6.3 Multi-Tenancy

In multi-tenant environments, where multiple applications are executed concurrently on the same machines, the utilization of shared resources varies over time.

Optimization. Applications are deployed in isolated virtual machines and the allocated memory for a virtual machine is not shared with others. This is because cluster managers (e.g., YARN [58]) will not offer an occupied memory region as a resource for a newly submitted application. Consequently, the unified memory (M in § 2.2) is not shared among multiple concurrent applications. Therefore, the cluster configuration recommended by JUGGLER is not affected by concurrent application runs hosted on the same machines.

Prediction. Ernest [59] realizes a very small execution time variance (less than 2 % of mean standard deviation) between repeated *identical runs* (same application, same data, and same configuration) for a 24-hour time frame on a multi-tenant public cloud (Amazon EC2). Contrarily, with more extensive experiments, others realize a considerable variance between identical runs [13, 29, 53, 63]. To adapt JUGGLER in such dynamic clouds, monitoring and modeling the non-deterministic behaviour of such multi-tenant clouds are required whether by measuring the noise caused by concurrently running applications during the experimental phase [13], predicting the performance variance in an online-training phase [63], performing online dynamic reprovisioning [29] or applying rule-based noise prediction techniques [61, 63].

7 EVALUATION

As there is no other end-to-end framework that addresses the same problem of configuration-based recommendation with fully automated caching of datasets, we compare JUGGLER with our baseline HiBench to see how much performance improvement JUGGLER brings. However, some stages/components of JUGGLER are similar to components of other state-of-the-art frameworks [9, 17, 23, 28, 33, 44, 59, 62, 65] that address different use cases such as cache eviction policies, resource-constrained clusters, materialized views and selection of shared sub-expressions. As we will show later, we enforce appropriate assumptions that ensure a fair comparison. It is thus important to state that the aim of these comparisons is not to show that the stages/components of JUGGLER outperform or replace the corresponding state of the art components, but rather to give empirical grounds for not using them in JUGGLER.

7.1 Workloads and experimental setups

We select five real-world iterative machine learning applications (cf. Table 1) from HiBench.

For the *offline training* of each application, JUGGLER runs a single experiment for hotspot detection (cf. § 5.1) and 9 experiments with randomly selected examples and features for parameter calibration (cf. § 5.2) on SPARK$_i$. Next, JUGGLER conducts a single experiment for memory calibration (cf. § 5.3) on JUGGLER engine. In these three stages, JUGGLER creates SCHEDULES for an application and predicts the required number of machines for each SCHEDULE with respect to selected application parameters. For conducting these stages and measuring the effect of cluster setup on the accuracy of models extracted therein, we use a single node equipped with Intel Core i3-2370M CPU running at 4x 2.40GHz, 3.8 GB RAM, and 388 GB disk. To construct execution time models (cf. § 5.4) for all SCHEDULES, JUGGLER carries out 9 experiments per SCHEDULE on JUGGLER engine.

For the *actual runs* (cf. § 5.5), we run the application with each SCHEDULE on JUGGLER engine. Even though JUGGLER selects a single configuration for each SCHEDULE, we run every SCHEDULE on 12 different configurations (1–12 machines) to validate whether JUGGLER recommends (near-to) optimal cluster configuration or not. Details of the actual runs of all evaluated applications are depicted in Figure 9. Throughout the evaluation, costs of runs are expressed in machine minutes (#machines × time).

We construct the execution time models (cf. § 5.4) and conduct the actual runs (cf. § 5.5) on our 12-node private Spark cluster. Each node is equipped with an Intel Core i5 CPU running at 4x 2.90 GHz, 16 GB RAM, 1 TB disk, and 1 GBit/s LAN. All nodes (including the previously mentioned single node) run Hadoop MapReduce 2.7, Spark 2.4.0, Java 8u102, Apache YARN, and HDFS.

Table 1: Details of evaluated applications.

Application	Examples	Features	Iterations	Input data	Datasets	Intermediate datasets	SCHEDULES
Linear Regression (LIR)	40k	120k	10	35.8 GB	111	16	2
Logistic Regression (LOR)	70k	50k	50	26.1 GB	210	4	2
Principal Components Analysis (PCA)	6k	5k	100	229.2 MB	1833	5	1
Random Forest Classifier (RFC)	100k	40k	3	29.8 GB	26	8	3
Support Vector Machine (SVM)	40k	80k	100	23.8 GB	524	9	2

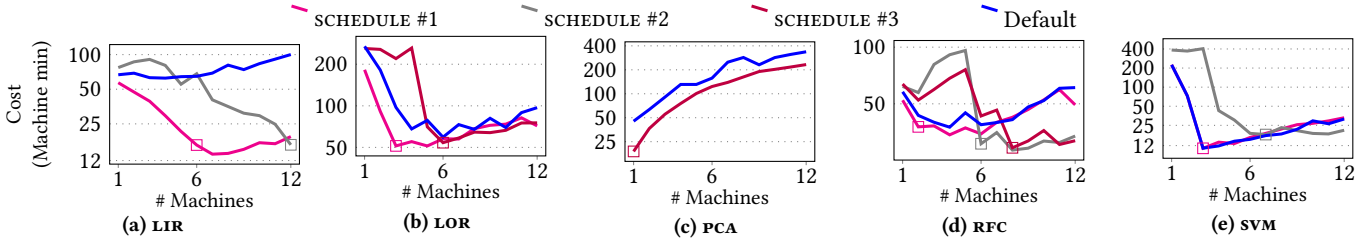

Figure 9: Actual runs with JUGGLER and HiBench SCHEDULES. JUGGLER's recommended configuration is indicated by a □ .

Table 2: JUGGLER's SCHEDULES & default schedules.

Application	ID	SCHEDULE
LIR	1	$p(1)$
LIR	2	$p(1)$ $p(3)$
LIR	HiBench	-
LOR	1	$p(2)$
LOR	3	$p(1)$ $p(2)$ $u(2)$ $p(\mathbf{11})$
LOR	HiBench	$p(2)$ $p(\mathbf{11})$
PCA	3	$p(1)$ $u(1)$ $p(2)$ $u(2)$ $p(13)$
PCA	HiBench	$p(2)$
RFC	1	$p(\mathbf{11})$
RFC	2	$p(1)$ $p(\mathbf{12})$
RFC	3	$p(1)$ $p(\mathbf{5})$ $u(\mathbf{5})$ $p(\mathbf{12})$
RFC	HiBench	$p(\mathbf{12})$
SVM	1	$p(2)$
SVM	2	$p(1)$ $p(\mathbf{6})$
SVM	HiBench	$p(2)$

7.2 Dataset selection

Table 2 presents a summary of *default schedules* (i.e., developer-cached datasets in HiBench) alongside the corresponding SCHEDULES selected by JUGGLER (cf. § 5.1). $p(i)$ and $u(i)$ respectively denote persisting and unpersisting dataset D_i. Through the framework-based instrumentation, JUGGLER caches framework-defined datasets (i.e., RDDs in Spark MLlib; cf. bold numbers in Table 2), which are not accessible by the application layer.

JUGGLER vs baseline. We evaluate JUGGLER's selected SCHEDULES by comparing each of them with the default one (cf. Figure 9). To appropriately compare JUGGLER's SCHEDULES with the default ones, but independently from cluster configuration, we select the execution cost of each SCHEDULE on optimal cluster configuration for each case. We achieve this by running each SCHEDULE on all cluster configurations and selecting the minimal execution cost. Our experiments demonstrate a substantial reduction of execution cost for most applications. In LIR, LOR, PCA and RFC, the minimal cost of every SCHEDULE generated by JUGGLER is lower than the minimal cost of the one presented by the default schedule. However, SVM is an exception. We can observe that the default schedule caches the same datasets selected by JUGGLER in SCHEDULE #1 (cf. Table 2). Even though JUGGLER selects SCHEDULE #2 as a solution despite its higher execution costs, JUGGLER nevertheless gives the option of running the application with lower execution time. As for PCA, all cached datasets fit into the memory of a single machine because their size is tiny. This explains the increasing cost whenever the number of machines exceeds one. Here, changing the cached datasets significantly reduces the time and cost of running PCA in all cluster configurations (41.7 % lower cost in the optimal cluster configuration). It is important to highlight that selection of suitable datasets to cache must go hand in hand with the selection of appropriate cluster configuration. For example, running LIR with SCHEDULE #2 on one machine leads to poor performance that costs more than the default schedule (cf. Figure 9a). However, running it on optimal cluster configuration (12 machines) reduces the cost to 24.6 %. For all applications, JUGGLER, on average, reduces execution time to 25.1 % and cost to 58.1 %, compared with HiBench.

Related components. We compare the hotspot detection with the following related components:

–*Cache eviction policies.* LRC [65, 66] and MRD [49] are DAG-aware cache eviction policies in Spark that rank cached datasets based on their reference count and reference distance, respectively, but without considering their size and computation time. They present cost models to rank datasets in order to select which to keep in case of cache limitation. We consider them as dataset selection policies rather than cache eviction policies. In addition, we assume that we have an unlimited amount of memory and thus all datasets are cached. After we apply their approaches, we select the first SCHEDULE, whose dataset has the highest rank. For the second SCHEDULE, we update the reference count with respect to the selected dataset in the first one, and successively select the highest-ranked dataset. Similarly, this procedure applies to consecutive SCHEDULES.

–*Recycling intermediate results.* Hagedorn et al. [23] propose a cost model to calculate the benefits of materializing common datasets between various Spark workloads in HDFS. Assuming that the capacity of HDFS is sufficient to materialize huge datasets, the presented cost model relies only on the computation time of datasets and the number of times they are computed, but does not take the size of datasets into account. Nagel et al. [44] present another cost model that calculates the benefit of materializing intermediate results in a limited cache. Similar to hotspot detection, its cost model takes

Table 3: Extra cost and time of related components compared to Juggler: Dataset selection.

	[44]	[28]	[23]	LRC	MRD
Cost	29 %	32 %	17 %	32 %	33 %
Time	22 %	30 %	10 %	37 %	49 %

into consideration the size of datasets, their computation time and the number of times they are computed. However, unlike hotspot detection, it neither re-evaluates nor unpersists stored datasets in previous schedules. To produce schedules from these cost models, we assume the storage is limited to cache one dataset for the first schedule, and then produce later schedules by incrementally extending the cache size.

–*Materializing common sub-expressions.* Jindal et al. [28] present a cost model that materializes the results of expressions common between different workloads, considering a limited storage budget. Their cost model relies on the utility of the sub-expression, which is the amount of time that is saved across all workloads if this sub-expression is materialized. To produce schedules, we apply the same methodology as in the case of recycling intermediate results.

Juggler vs related components. Similar to the comparison between Juggler and HiBench, we select the optimal cluster configuration for each schedule recommended by related components by running it on all cluster configurations and selecting the one with minimal execution cost. Figure 10 shows the cost of various schedules. It can be observed that some related components recommend more schedules than others. But Juggler is able to compare and omit inefficient schedules. For example, Juggler recommends one efficient schedule in pca that results in minimal execution cost compared to all other schedules (cf. Figure 10c). In some cases, Juggler and the related components recommend the same schedules. Since [23] and [28] do not consider the size of datasets, both of these approaches present schedules that recommend caching large datasets, which require more machines to fit them in memory. LRC and MRD have the same limitation, besides not considering the computation time of datasets. This also leads to a sub-optimal selection of datasets. [44] recommends efficient schedules when one dataset is to be cached. However, caching additional datasets leads to an inefficient combination of cached datasets. This is because it does not re-evaluate previously cached datasets, when required, upon caching new ones. As seen in Table 3, Juggler recommends schedules with not only minimal execution cost but with minimal execution time as well. For every application, Figure 11 shows the average cost of the approaches. In all applications, Juggler recommends schedules with minimal execution cost.

7.3 Performance prediction

We compare execution time prediction of Juggler with Ernest [59], which relies on a sampling-based approach to provide accurate predictions for long-running Spark workloads. To ensure low overhead, it collects training data points by applying *optimal experiment design* [51]. Ernest presents an execution time model which considers serial parts, parallel parts, and the overhead of a higher number of machines, but without taking cache limitations into account.

As a reminder, to predict the execution time of a schedule, Juggler does not use the operator-level execution time model (cf. § 3.3) used in the hotspot detection. Rather, Juggler trains the execution time models (cf. § 5.4) by carrying out 9 experiments of each schedule with randomly selected application parameters (examples and features). Each experiment is carried out using the optimal cluster configuration, which Juggler predicts in advance (§ 5.3). To train the execution time models using Ernest, we carry out 7 experiments, as per its optimal experiment design [51], on 1–12 machines with tiny datasets (1 %–10 %). For each schedule, Figure 12 shows the execution time prediction accuracy of Juggler and Ernest on the optimal cluster configuration. On average, the accuracies of Juggler and Ernest are 90.6 % and 53.2 % respectively.

In Juggler's case, the training experiments are not short-running. In Ernest's case, the experiments are short (mostly less than a minute for each run) because they are carried out with tiny datasets and thus influenced by relatively big noise due to uncertain internal cluster dynamics and stragglers [13, 47]. This explains why Ernest's prediction is inaccurate in most of the cases.

7.4 Dataset size prediction

We compare the sizes of cached datasets in each schedule (cf. Table 2) in actual runs (cf. Table 1) with the sizes that Juggler (cf. § 5.2) predicts with respect to the #Examples and #Features in Table 1. Figure 13 shows that the predicted and actual sizes of cached datasets are almost equal in all cases. In the worst case, the error of Juggler is 0.91 %.

7.5 (Near-to) optimal cluster configuration

In the following, we evaluate Juggler's recommendation of cluster configuration by comparing it with related components with respect to the optimal cluster configuration. Note that we obtain the optimal cluster configuration by running the schedules on all possible cluster configurations (1–12 machines) to get the configuration with the minimal cost.

Juggler compared to optimal. For every schedule, we compare the optimal cluster configuration with Juggler's recommendation, as shown in Figure 14. We see that Juggler selects an optimal solution in 50 % of cases. In the remaining 50 % of cases, Juggler's recommendation is near-to optimal. The same observation is also shown previously in Figure 9. On average, the error incurs an additional execution cost of 7.3 %. It is important to highlight that in some cases (especially in CPU-intensive applications), datasets are tiny and might fit into the memory of a single machine. In such cases, Juggler recommends a single machine to run the application which leads to the longest execution time but also minimal execution cost (e.g., pca - see Figure 9c), which is acceptable since the goal of Juggler is to minimize the execution cost.

Related components. We select the following related components which share with Juggler the same goal of predicting the required memory budget for Spark applications.

–*MemTune* [62] is a dynamic memory manager that predicts memory usage and dynamically tunes the memory fraction (execution-caching ratio) in resource-constrained clusters. It prioritizes memory allocation for execution over caching to minimize GC overhead. We apply its approach to tune the number of machines instead of the memory fraction.

–*RelM* [33] introduces a safety factor to ensure error-free runs in resource-constrained clusters. It also considers cache eviction ratio,

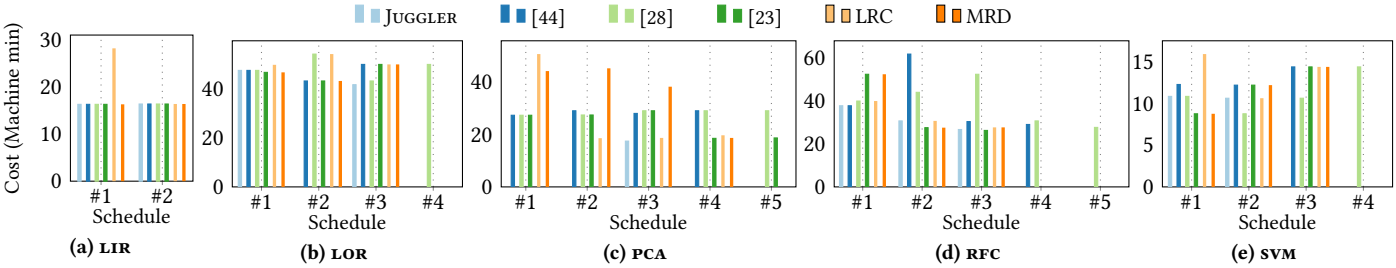

Figure 10: Juggler vs related components: Dataset selection.

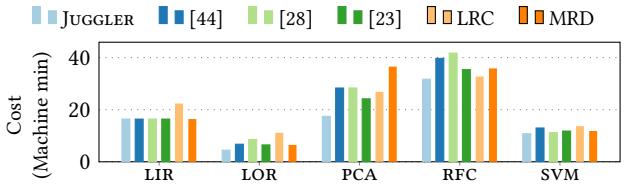

Figure 11: Juggler vs related components: Aggregated view of dataset selection.

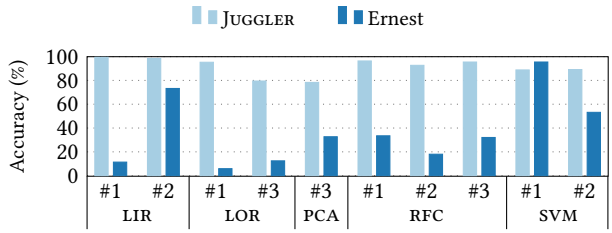

Figure 12: Juggler vs Ernest: Prediction accuracy.

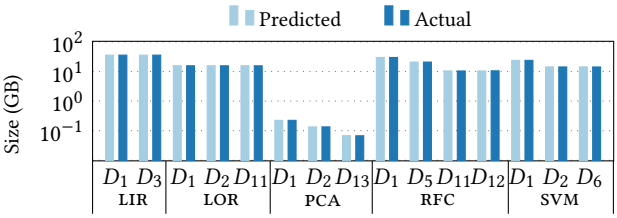

Figure 13: Juggler's dataset prediction accuracy.

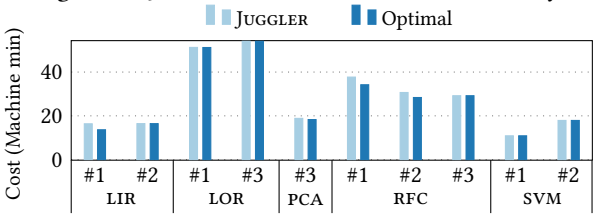

Figure 14: Juggler's recommendation compared to optimal cluster configuration.

task concurrency and low GC overhead. Similar to MemTune, we apply its white-box cost model to tune the number of machines.

–*SystemML* [17] relies on worst-case memory estimates considering fitting of all input, intermediate and output data in memory.

Juggler vs related components. We analyze the memory footprint and data sizes of actual runs with schedules recommended by Juggler and select a cluster configuration that satisfies each related component. For example, Juggler recommends to run *Logistic Regression* with schedule #1 on 3 machines, while SystemML recommends 4 machines to fit input and output data in memory, in addition to cached datasets. Figure 15 and Table 4 show that Juggler recommends cluster configuration with minimal cost, compared to the related components. MemTune, in some cases, selects cluster configuration which leads to cache eviction, and in other cases, over-allocates cluster configuration. To fit data input and output, SystemML always over-allocates cluster configuration. RelM over-allocates cluster configuration in accordance with its safety factor and also to ensure low GC overhead. During our experiments, we realize that RelM recommends more machines than all others. On the one hand, this results in RelM having the highest cost (cf. Table 4). On the other hand, RelM has the lowest execution time because over-allocating machines still increases the degree of parallelism. The same reason holds for SystemML, even though it recommends less number of machines compared to RelM.

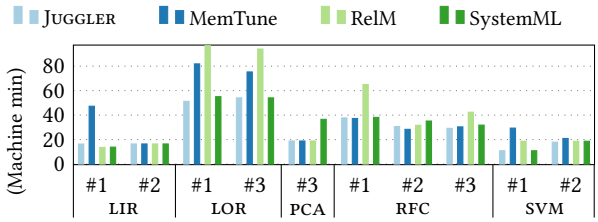

Figure 15: Juggler vs. related components: Recommended cluster configuration.

Table 4: Cost and time ratio of related components compared to Juggler: Recommended cluster configuration.

	MemTune	RelM	SystemML
Cost	36 %	46 %	9 %
Time	−9 %	−46 %	−18 %

Variance in data partition sizes. From our experiments, we observe that the sizes of data partitions are usually not equal. For example, in a single SVM run with schedule #2, some partitions are two times larger than others. Despite this variance, all data partitions remain in memory during the run. The reason behind this is that the execution time of tasks varies with varying data sizes. *TaskScheduler* in Spark manages this execution time variance between tasks by assigning tasks to machines that have free execution slots. There is thus an unequal distribution of tasks among machines. But on the other hand, the total cached data partitions in each machine is almost equal. In some iterations, we observe that apart from data size, stragglers also introduce runtime variance between tasks, which causes cache eviction. However, in later iterations, the evicted partitions fit in memory when they are recomputed on other machines. In the same SVM example, 14 partitions out of 362 are evicted in the first iteration, only 3 are evicted in

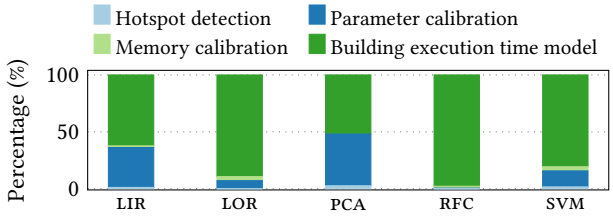

Figure 16: Training cost of Juggler's stages.

the second one, and all partitions remain in memory in the third one. The execution cost, in this case, is near-to optimal. And this explains why Juggler in 50 % of cases recommends near-to-optimal instead of the optimal cluster configuration.

7.6 Training overhead and general gains

Figure 16 presents the percentage training costs of the individual training stages in Juggler. For all applications, most of the overall offline training cost comes from building the execution time model.

To evaluate the efficiency of Juggler, we compare, for each application, the average execution cost of HiBench implementation (Table 5: Line 1 refers to Figure 9:) on all cluster configurations (i.e., the developer-cached datasets without recommendations of cluster configuration) with the average execution cost using Juggler's selected schedules and recommended cluster configuration (Table 5: Line 2 refers to Figure 9: , ,). From these two measurements, we observe how much the cost savings per run is (Table 5: Line 3, 57.8 % on average), and how many runs are required to start achieving the benefits of using Juggler. Note that the cost savings per run is achieved from the optimization models (cf. § 5.1, § 5.2 and § 5.3). On average, 4 actual runs are required to start gaining optimization benefits. To build the execution time models for the first time or re-build them, in the worst case, 43 actual runs are required, on average, to start gaining prediction benefits. However, as mentioned earlier, re-building them can be avoided (cf. § 6). As shown in Table 5, for pca, we achieve the benefits of using Juggler from a single actual run. While in rfc, more actual runs are required to achieve the benefits of using Juggler. The reason behind this is the large training overhead caused by building the execution time models of three schedules. In addition, the rfc cost savings per run (31 %) is lower than that of other applications due to the small number of iterations per run (cf. Table 1). The purpose of choosing this small number of iterations is to validate the efficiency of Juggler in such cases. The required number of runs to start gaining benefits of using Juggler (as shown in Table 5) is negligible since, on a daily basis, the usage of these applications and their likes is immense. In addition to the number of runs, the problem size (i.e., the input data size and the number of iterations) also plays a role in evaluating the efficiency of Juggler. For example, by increasing the input data size of svm to 40 GB and the number of iterations to 200, the cost saving per run increases to 79 % and, thus, 11 actual runs are needed (instead of 26) to benefit from Juggler.

8 RELATED WORK

Apart from our baselines (cf. § 7), we group the following related work according to the topic they have addressed.

Dataset selection. The majority of contributions [16, 40, 54] that propose solutions regarding selecting datasets to cache assume limited cache storage and, thus, consider it as a knapsack problem and propose cache eviction schemes. These studies either assume that the datasets are not interdependent and thus caching a dataset has no bearing on the benefits of caching other datasets or are limited to certain types of join operators or propose caching models that reduce network traffic in geographically distributed networks.

Cluster configuration. Several studies address the issue of selecting appropriate cluster configuration in public clouds [13, 31, 32, 38, 39, 48]. These approaches rely on sample runs, whose overhead is relatively high in case of short actual runs, and study applications in a black-box manner without considering their internals (e.g., application parameters, caching of intermediate results, etc.).

Performance prediction. Several contributions [25, 50, 60, 61, 69] present fine-grained approaches for execution time prediction of big data applications that consider application internals (e.g., I/O cost, shuffling, degree of parallelism, interference between concurrent tasks, etc.). In contrast, others [12, 21] present black-box performance prediction methodologies relying on ML models without diving into application internals. None of these contributions considers the impact of cache limitation and application parameters in their presented execution time models.

Cardinality estimation. Lastly, many studies focus on cardinality estimation [30, 35, 36, 46, 64]. Catalyst optimizer [15] runs on top of *spark-sql* for rule and cost-based optimization. However, all these contributions focus on database operators with clear knowledge of their internals, but without considering black-box operators.

Table 5: Juggler's training cost efficiency and general gains.

	LIR	LOR	PCA	RFC	SVM
Default cost (machine min)	73.8	102.7	193.2	47.1	24.2
Juggler cost (machine min)	16.4	52.7	18.9	32.5	14.4
Cost savings per run	78 %	49 %	90 %	31 %	41 %
			Optimization		
Training cost (machine min)	89.4	228.1	20.91	49.26	48.43
#Runs to start gain benefits	2	5	1	4	5
			Prediction		
Training cost (machine min)	147.8	1912.2	22.58	2185.2	202.19
#Runs to start gain benefits	3	39	1	151	21
			Total		
Training cost (machine min)	237.29	2140.3	43.5	2234.5	250.6
#Runs to start gain benefits	5	43	1	154	26

9 CONCLUSION

Juggler is an autonomous optimization and prediction framework that minimizes the execution time and cost of application runs. It achieves this by selecting appropriate datasets for caching and recommending optimal cluster configuration with accurate performance prediction. Overall, the evaluation of Juggler shows very good results, in comparison with its baseline and related work.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their helpful comments. 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 "Memristive Materials for Neuromorphic Electronics (MemWerk)".

REFERENCES

[1] [n.d.]. Amazon Web Services. https://aws.amazon.com. Accessed: 2022-02-28.
[2] [n.d.]. Curve_fit Solver. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html. Accessed: 2022-02-28.
[3] [n.d.]. Google Cloud Platform. https://cloud.google.com. Accessed: 2022-02-28.
[4] [n.d.]. Microsoft Azure. https://azure.microsoft.com/en-us/. Accessed: 2022-02-28.
[5] [n.d.]. Monitoring and Instrumentation in Spark. https://spark.apache.org/docs/2.4.0/monitoring.html. Accessed: 2022-02-28.
[6] [n.d.]. The Apache Spark. https://spark.apache.org. Accessed: 2022-02-28.
[7] [n.d.]. The Apache Spark Configuration. https://spark.apache.org/docs/2.4.0/configuration.html. Accessed: 2022-02-28.
[8] [n.d.]. The HiBench Suite. https://github.com/Intel-bigdata/HiBench/tree/HiBench-7.0/. Accessed: 2022-02-28.
[9] Mania Abdi, Amin Mosayyebzadeh, Mohammad Hossein Hajkazemi, Ata Turk, Orran Krieger, and Peter Desnoyers. 2019. Caching in the Multiverse. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.
[10] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Reoptimizing data parallel computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 281–294.
[11] Hani Al-Sayeh, Stefan Hagedorn, and Kai-Uwe Sattler. 2020. A gray-box modeling methodology for runtime prediction of apache spark jobs. *Distributed and Parallel Databases* 38, 4 (2020), 819–839.
[12] Hani Al-Sayeh, Bunjamin Memishi, Marcus Paradies, and Kai-Uwe Sattler. 2020. Masha: Sampling-Based Performance Prediction of Big Data Applications in Resource-Constrained Clusters. In *The 1st Workshop on Distributed Infrastructure, Systems, Programming and AI (DISPA)*. Very Large Data Base Endowment Inc.(VLDB Endowment).
[13] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. {CherryPick}: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.
[14] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.
[15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.
[16] Shivnath Babu, Kamesh Munagalat, Jennifer Widom, and Rajeev Motwani. 2005. Adaptive caching for continuous queries. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 118–129.
[17] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.
[18] Khaled Elmeleegy. 2013. Piranha: Optimizing short jobs in hadoop. *Proceedings of the VLDB Endowment* 6, 11 (2013), 985–996.
[19] Brad Everman, Narmadha Rajendran, Xiaomin Li, and Ziliang Zong. 2021. Improving the cost efficiency of large-scale cloud systems running hybrid workloads-A case study of Alibaba cluster traces. *Sustainable Computing: Informatics and Systems* 30 (2021), 100528.
[20] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*. 99–112.
[21] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. 2021. On the Use of {ML} for Blackbox System Performance Prediction. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 763–784.
[22] Atul Gohad, Nanjangud C Narendra, and Parathasarthy Ramachandran. 2013. Cloud Pricing Models: A Survey and Position Paper.. In *2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 1–8.
[23] Stefan Hagedorn and Kai-Uwe Sattler. 2018. Cost-based sharing and recycling of (intermediate) results in dataflow programs. In *European Conference on Advances in Databases and Information Systems*. Springer, 185–199.
[24] Álvaro Brandón Hernández, María S Perez, Smrati Gupta, and Victor Muntés-Mulero. 2018. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems* 86 (2018), 1076–1092.
[25] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*, Vol. 11. 261–272.
[26] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 41–51.
[27] Muhammed Tawfiqul Islam, Shanika Karunasekera, and Rajkumar Buyya. 2017. dSpark: deadline-based resource allocation for big data applications in Apache Spark. In *2017 IEEE 13th International Conference on E-Science (e-Science)*. IEEE, 89–98.
[28] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.
[29] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards Automated {SLOs} for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 117–134.
[30] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).
[31] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 759–773.
[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.
[33] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.
[34] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 175–186.
[35] Per-Ake Larson, Wolfgang Martin Josef Lehner, Jingren Zhou, and Peter Alfred Zabback. 2008. Cardinality estimation in database systems using sample views. US Patent App. 11/760,203.
[36] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.
[37] Hui Li, Dong Wang, Tianze Huang, Yu Gao, Wensheng Dou, Lijie Xu, Wei Wang, Jun Wei, and Hua Zhong. 2020. Detecting cache-related bugs in Spark applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 363–375.
[38] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 189–203.
[39] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. 2019. {SOPHIA}: Online Reconfiguration of Clustered {NoSQL} Databases for {Time-Varying} Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 223–240.
[40] Tanu Malik, Randal Burns, and Amitabh Chaudhary. 2005. Bypass caching: Making scientific databases good network citizens. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 94–105.
[41] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. 2017. Improving spark application throughput via memory aware task co-location: A mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 95–108.
[42] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
[43] Jonas Mockus. 2012. *Bayesian approach to global optimization: theory and applications*. Vol. 37. Springer Science & Business Media.
[44] Fabian Nagel, Peter Boncz, and Stratis D Viglas. 2013. Recycling in pipelined query evaluation. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 338–349.
[45] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Analysis and exploitation of dynamic pricing in the public cloud for ml training. In *VLDB DISPA Workshop 2020*.
[46] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 984–994.
[47] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 293–307.
[48] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.
[49] Tiago BG Perez, Xiaobo Zhou, and Dazhao Cheng. 2018. Reference-distance eviction and prefetching for cache management in spark. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.
[50] Adrian Daniel Popescu, Andrey Balmin, Vuk Ercegovac, and Anastasia Ailamaki. 2013. Predict: towards predicting the runtime of large scale iterative analytics. *Proceedings of the VLDB Endowment* 6, CONF (2013).

[51] Friedrich Pukelsheim. 2006. *Optimal design of experiments*. SIAM.
[52] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*. 1–13.
[53] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.
[54] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. Watchman: A data warehouse intelligent cache manager. In *Proceedings of the VLDB Conference*.
[55] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*. 1135–1149.
[56] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. 2016. Optex: A deadline-aware cost optimization model for spark. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 193–202.
[57] Tobias Storch, Christoph Reck, Stefanie Holzwarth, and Vanessa Keuck. 2018. Code-de-the german operational environment for accessing and processing copernicus sentinel products. In *IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, 6520–6523.
[58] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.
[59] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for {Large-Scale} Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 363–378.
[60] Kewen Wang and Mohammad Maifi Hasan Khan. 2015. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 166–173.
[61] Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen, and Swapna Gokhale. 2016. Modeling interference for apache spark jobs. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 423–431.
[62] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 383–392.
[63] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*. 452–465.
[64] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).
[65] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.
[66] Yinghao Yu, Chengliang Zhang, Wei Wang, Jun Zhang, and Khaled Letaief. 2019. Towards Dependency-Aware Cache Management for Data Analytics Applications. *IEEE Transactions on Cloud Computing* (2019).
[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.
[68] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.
[69] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. 2018. Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 22–32.
[70] Ziyao Zhu, Qingni Shen, Yahui Yang, and Zhonghai Wu. 2017. MCS: Memory Constraint Strategy for Unified Memory Manager in Spark. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 437–444.