Agile-Ant: Self-managing Distributed Cache Management for Cost Optimization of Big Data Applications

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

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

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

ABSTRACT

Distributed in-memory processing frameworks accelerate application runs by caching important datasets in memory. Allocating a suitable cluster configuration for caching these datasets plays a crucial role in achieving minimal cost. We present Agile-ant, a self-managing framework that identifies important datasets and scales out the cluster memory to cache them on the fly without any human interaction, without any prior knowledge of the application, the characteristics of the input data, the specification of the computing resources and their utilization by multiple-tenants. We evaluate Agile-ant on various real-world applications. Compared with our baseline, Agile-ant reduces execution cost by 78.3 % on average and provides better performance than the related work.

PVLDB Reference Format:
Hani Al-Sayeh, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Agile-Ant: Self-managing Distributed Cache Management for Cost Optimization of Big Data Applications. PVLDB, 17(11): 3151 - 3164, 2024.
doi:10.14778/3681954.3681990

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbis-ilm/Agile-Ant.

1 INTRODUCTION

Distributed in-memory processing platforms like Spark accelerate the performance of big data applications by utilizing multiple executors in parallel and using each executor's memory (*executor cache*) for caching [55]. Caching appropriate datasets reduces execution time and cost significantly [10, 30, 38], and increasing the *cluster size* (i.e. adding executors) to allocate sufficient *cluster cache* (executor cache × #executors) avoids cache evictions [8]. Ideally, developers of applications or libraries (e.g. Spark MLlib [37] and GraphFrames [40]) need to make correct *caching decisions*, i.e. to appropriately cache and *unpersist* (remove) datasets, and cluster administrators need to select the optimal cluster size for caching [10]. However, it is very difficult to achieve both because of the following.

Firstly, many factors like cluster cache size and data characteristics are only available during application runs. Thus, the developers make caching decisions based on conjectures (§3.1), as they lack crucial information like the computation times and sizes of datasets.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681990

Secondly, as the memory of each executor is used for execution and caching [10], to select the optimal cluster size, the administrators need to predict the required memory for execution to avoid GC overheads and the size of cached datasets to avoid evicting and, as a result, recomputing them [8]. They can use cardinality estimation techniques [22, 25, 28, 32, 51], but they need to know the inner workings of operators. Alternatively, they could use linear models for the prediction [8], but the performance might be poor (§3.2).

Thirdly, the two challenges above are compounded because developers and administrators are required to tackle them in unison (§3.3). Developers make caching decisions independently of the cluster cache size, after which administrators try to determine a suitable cluster configuration. And since administrators handle application binaries [26], they cannot modify the caching decisions.

Some contributions try to avoid poor performance in advance by supporting developers in making caching decisions [9, 30], administrators in selecting cluster configurations [8, 12, 35, 48] or both [10]. Others try to optimize performance during application runs by caching datasets [38], reconfiguring the cluster [23, 27, 52] or enforcing cache eviction policies [41, 53]. All of these tackle specific use cases with predefined assumptions (§4) e.g. available data samples, recurring applications etc. In contrast, we make a strong step towards fully self-managing clouds that minimize the execution cost of big data applications. To this end, we propose Agile-ant, a self-managing framework that makes correct caching decisions and selects optimal cluster sizes: (a) for all application types (iterative, data-intensive, compute-intensive etc.) and domains (machine learning, graph analysis etc.), (b) with no prior knowledge of the inner workings of the applications or the characteristics of the input data, (c) for a huge list of application parameters, (d) with no available history of previous runs or data samples, and (e) in the face of uncertainties due to multi-tenancy. Note that: (i) Minimizing the execution cost is not at the expense of the execution time, as there is a correlation between the two, i.e. execution cost = execution time × #executors. (ii) On public clouds where a pay-as-you-go pricing model is used [21], reducing the execution cost minimizes monetary costs. (iii) Minimizing the execution cost increases the execution efficiency and leads to better utilization of limited resources [35].

Agile-ant performs a repetitive cycle of three steps. Firstly, it caches/unpersists datasets based on their reuse pattern (§6). Secondly, it observes their computation time and size and evaluates whether caching them is beneficial or not (§7). Thirdly, it adjusts the cluster cache (i.e. adds/removes executors) to cache all partitions of the beneficial datasets (§8). It also ensures a balanced distribution of cached partitions among executors to avoid having straggler executors in case of skewed data (§7.2). It does this adaptively in the face of changes in cluster utilization resulting from

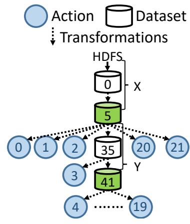

Figure 1: DAG (LIR).

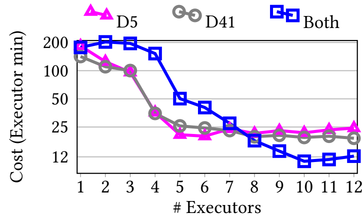

Figure 2: Caching datasets (LIR).

multi-tenancy (§8.2.2). Compared to existing work (§4), AGILE-ANT is best suitable where making the optimal caching decisions and predicting the optimal cluster size is not possible before the actual run (§3). It is not always possible to predict exactly how the application run will be because (i) some applications are non-recurring and without historical runs, (ii) sample/training data is often not representative, (iii) many application parameters contribute to the sizes of datasets and the execution DAG of data-driven applications, and (iv) users submit ad hoc queries in interactive sessions, etc. AGILE-ANT frees developers from making caching decisions and enables administrators to simply select a single executor and set the maximum number of executors. It then handles the rest.

Our evaluation using 21 applications shows that AGILE-ANT reduces execution cost by 78.3 % compared to our baseline. Its cost is 32.6 %, 57.6% and 16.3 % of that of cache eviction policies, on-the-fly caching approaches and cost optimization frameworks respectively. Its learning overhead with regards to caching decisions and cluster size selection is 13.3 % and 3.5 % respectively compared to handpicking the optimal caching decisions and cluster sizes in advance.

2 BACKGROUND

Below, we discuss Spark as a use case. The underlying concepts also apply to other distributed in-memory frameworks (see §9.1).

2.0.1 Execution Model. Spark runs applications on a *driver* and a set of *executors* that perform various operations in parallel on Resilient Distributed Datasets (*RDDs*) [54]. *Transformations* are operations that create new RDDs from existing ones while *actions* return a value to the driver. An application consists of *job(s)*, each of which is triggered by an action. A job is represented by a *DAG*, comprising a sequence of transformations followed by an action. Each RDD points back to its parent, which entails the parent-child dependencies. The DAG of each job is constructed starting from the action. Then the parent RDDs are constructed one by one, based on the parent-child dependencies, towards the root RDDs that depend on no other RDDs. Note that the direction of these dependencies is opposite that of the data flow. A transformation is either *narrow* or *wide*. Wide transformations split the DAG, at shuffle boundaries, into *stages*. Each stage comprises *tasks* that run concurrently on executors to perform the same computation on different RDD partitions [49]. The driver starts the application run by constructing the DAG of the first job (based on the first action). Then, all executors execute the job (in tasks) and return the result to the driver. The driver then constructs the DAG of the second job (based on the second action), and so on. If an RDD is cached, the driver notifies all executors to cache its partitions in their local cache. In data-driven applications, the driver constructs the DAG of a job based on the results of the previous one. Therefore, predicting the whole

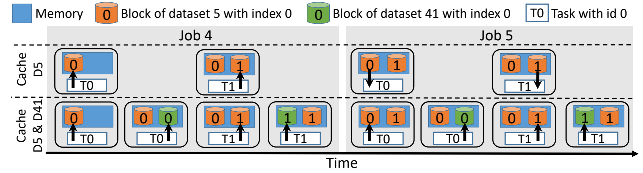

Figure 3: Cyclic eviction in Spark.

application DAG is not possible. Jobs are executed sequentially and the DAG of each job is constructed immediately before the job's execution. Thus, while executing a job, the DAGs of the subsequent jobs are not yet available. For example, in the K-Means application, the driver keeps modifying the centroid in iterations (i.e. jobs) until the convergence is reached [42]. For the remainder of this paper, we refer to an RDD as a dataset (D) and RDD partitions as blocks.

2.0.2 Caching. Jobs may have transformations in common. Fig. 1 shows the DAG of 22 jobs in the Linear Regression (LIR) implementation of Spark MLlib. The computation of the datasets can be traced in a depth-first traversal order, starting from D_0. Without caching, the number of computations of a dataset is determined by the number of its child branches, e.g. D_5, D_{35} and D_{41} will be computed 22, 17 and 16 times respectively. The number of computations of a dataset decreases if its child is cached, as it will not be recomputed to recompute the child. For example, if D_{41} is cached, it will not be recomputed and, thus, D_5 will be computed 7 times. Assuming the computation time of D_5 is x and the computation time of D_{41} starting from D_5 is y, caching D_{41} reduces the execution time by (x + y) × 15 while caching D_5 reduces it by x × 21. Developers of Spark MLlib cached D_5 and D_{41} (shown in green in Fig. 1). Spark provides an API for developers to cache/unpersist datasets in/from memory and allows replicating cached blocks across executors [5, 31, 54].

2.0.3 Memory Management. The unified memory (M) of each executor is shared between the storage and the execution regions, respectively used for caching and computing datasets such that if the execution memory is not utilized, the entire M can be used for caching, and vice versa [57]. There is a minimum storage memory (R) below which cached blocks are not evicted. Thus, if more execution memory is required during the application run, some cached blocks are first evicted (until reaching R) based on a cache eviction policy (§2.0.5) and then GC takes place to free up execution memory. The *actual caching capacity* (between R and M) of each executor depends on the application's execution memory requirements [10].

2.0.4 Skipped Stages. While conducting wide transformations, Spark persists shuffled blocks [2, 9]. Thus, a stage that already computed shuffled blocks will be *skipped* in their later usage.

2.0.5 Eviction Policy. Spark applies the LRU policy to evict some blocks in case of cluster cache limitations. It does not evict a block to cache another one from the same dataset, thereby avoiding wasteful cyclic replacement of blocks [4]. In Fig. 3, we show the impact on the performance of LIR when the cluster cache is not sufficient to cache D_5 and D_{41} and how each task caches, evicts, and reads blocks. Consider two sequential tasks on an executor with a cache that fits two blocks and all blocks are equal in size. The memory of an executor is shared among all tasks running on the executor. If only D_5 is cached, $task_0$ and $task_1$ cache D_{5_0} and D_{5_1} respectively in memory in job_4 and read them in job_5. But if D_5 and D_{41} are

cached, in job_4, $task_0$ caches D_{5_0}, then D_{41_0}. To cache D_{5_1}, $task_1$ evicts D_{41_0} not D_{5_0} because D_{5_0} belongs to the same dataset as D_{5_1}. $Task_1$ then evicts D_{5_0} to cache D_{41_1} because D_{5_1} is still locked by $task_1$. In job_5, $task_0$ recomputes D_{5_0} and D_{41_0}, and caches them after evicting D_{5_1} and D_{41_1}, and the cycle goes on.

2.0.6 Push-Based Execution. Spark follows a push-based execution model [44]. *Non-blocking* transformations (e.g. filter) process tuples one by one while *blocking* transformations (e.g. sort) accumulate all tuples in a batch. To cache a block, the execution of a transformation (whether blocking or non-blocking) is blocked in that all tuples are accumulated and then the whole block is stored in memory.

2.0.7 Call-site. Spark provides a *call-site* for each dataset that identifies the point in the application code where the dataset is defined. Datasets that share the same call-site have a comparable reuse pattern [38] and we refer to them as *similar datasets*.

2.0.8 Dynamic Resource Allocation. Spark provides the option to add and remove executors during an application run based on the waiting time of tasks in the execution queue (see [3] for details).

3 PROBLEM STATEMENT AND MOTIVATION

3.1 Caching Decisions

Input Data. We run the Principal Component Analysis (PCA) implementation of Spark MLlib on our cluster (§10.1) with a 16.8 GB input data generated using HiBench [6]. No dataset in PCA was cached by the developers. We modify Spark MLlib to cache a dataset that is reused in 600 iterations. Thereafter, the average execution time and cost across all cluster sizes reduced by 53.8 % and 71.9 % respectively. We show this in Fig. 4 and highlight three areas: (1) *Area A* (1-2 executors): the cluster cache is not enough to cache all the blocks of the cached dataset, leading to recomputations of the evicted blocks in each iteration. Adding more executors increases the cluster cache size and thus reduces the execution time and cost. (2) *Area C* (3 executors): the cluster cache is just sufficient for all cached blocks, resulting in the minimal execution cost. (3) *Area B* (4-12 executors): the cluster cache is larger than needed, which reduces the execution time but increases the execution cost [10]. However, the same dataset is used only once when running PCA on 7.5 GB input data. Thus, a sample/historical run on 7.5 GB does not capture the reuse patterns in an actual run on 16.8 GB. Henceforth, we only discuss execution costs since our target is the minimal execution cost (§1).

Application Parameters. We run the Random Forest Classifier (RFC) implementation of Spark MLlib on a 7.7 MB input data. When we set the *maxDepth* to 1, no dataset is reused, whereas when we set it to 10, one dataset is reused 30 times. Thus, caching decisions need to consider the value of certain application parameters.

Application Nature. We run the Latent Dirichlet Allocation (LDA) implementation of Spark MLlib on a 246.0 MB input data. We cache an uncached dataset that is reused 22 times, similarly to PCA, but realize no performance gains because unlike PCA, the dataset's computation time is very small compared to the total iteration time. Thus, caching a frequently reused dataset is not always beneficial.

3.2 Cluster Configuration

We run the Label Propagation (LP) application [40] on the *cit-Patents* [29] and *road-road-usa* [43] datasets of sizes 267.5 MB and

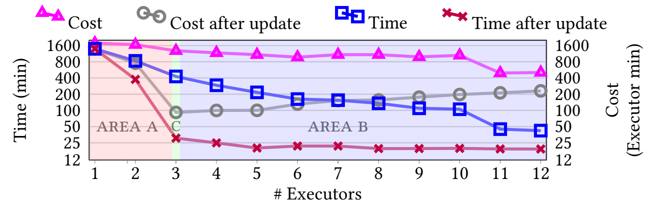

Figure 4: Caching decision and cluster configuration (PCA).

469.7 MB respectively. Although road-road-usa is only about double the size of cit-Patents, the optimal cluster sizes for their respective LP runs are 9 and 1 (§10.2). Thus, linear models may not predict the optimal cluster size accurately. Also, a slight error in selecting the cluster size could lead to a dramatic overhead. For example, our experiments show that running the Page Rank (PR) application on road-road-usa takes 12.8 minutes on 9 executors and, surprisingly, more than one week on 8 executors (see §10.2 for explanation).

3.3 Multivariate Optimization Problem

We run LIR on all cluster sizes with 44.7 GB input data, 15 iterations and three caching options (§2.0.2): (1) cache D_5 (2) cache D_{41} (3) cache both. Each of D_5 and D_{41} fits in a 5-executor cluster cache. Fig. 2 shows that caching both datasets is the best option if more than 7 executors are allocated. Otherwise, caching one dataset is the best option (see §2.0.5 for an explanation). Hence, caching decisions shall consider the cluster cache. Achieving this synergy is difficult, as developers make caching decisions regardless of the cluster size.

4 RELATED WORK

Caching Decisions. SparkCAD [9], CacheCheck [30] and ReSpark [38] cache datasets based only on their reuse patterns. SparkCAD relies on a sample run, which might not represent the actual runs (§3.1). CacheCheck requires the whole application source code to detect caching-related bugs, which is impossible if binaries of other libraries are used. ReSpark [38] caches datasets during actual runs regardless of whether they fit in the cluster cache or not.

Cache Evictions Policies. LRC [53] and MRD [41] improve performance when multiple cached datasets do not fit in the cluster cache. They neither change caching decisions nor resize the cluster cache, which renders them not useful if one or no dataset is cached and not able to completely avoid recomputations. Similarly, buffer pool management of databases loads pages from disk to memory and uses eviction policies if not all pages fit in memory [19, 20, 24].

Cluster Configuration. Ernest [48] and Masha [11] conduct sample runs to predict the performance of big data applications. Cherrypick [12] and [7] rely on historical runs to recommend cluster configurations. Although Blink [8] considers cache limitations while predicting the optimal cluster size, it relies on linear size prediction models which might lead to poor performance (§3.2). [50] and CherryPick predict the interference of multiple tenants before conducting actual runs but they do not measure cluster utilization changes during actual runs. Some frameworks support elasticity [3, 15, 23, 33, 34], but they scale out reactively i.e. after starting to incur performance penalties e.g. due to memory limitations. Juggler [10] is a training-based framework that recommends caching decisions and cluster sizes. It conducts training runs using tiny datasets, which might lead to wrong caching decisions (see PCA in §3.1).

It considers only two application parameters for training its linear models, which might lead to poor performance (§3.2). Also, buffer pool management of databases adjusts sizes of the multiple buffer pools based on prior runs and the database workload [17, 36, 45].

In summary: (1) Relying on previous runs leads to poor caching decisions (§3). (2) Not adapting caching decisions when there are cache limitations results in poor performance (§3.1 and §3.3). (3) Not extending resources on the fly incurs performance losses (§3.1 and §3.2). Unlike the works above, Agile-ant adapts both caching decisions and cluster resources in unison without relying on previous runs. It is compatible with frameworks that support elasticity. But unlike their reactive approach, Agile-ant proactively requests for resources before any performance penalties. Also, buffer pool management techniques in databases propose tuning parameters of non-extendable memory and propose eviction policies for fixed-sized pages. Contrarily, cloud computing systems support elasticity i.e. extending the cluster memory during application runs, where the sizes of blocks may vary even if they belong to the same dataset.

5 AGILE-ANT

Agile-ant caches suitable datasets and allocates a suitable number of executors to avoid evictions. To this end, we introduce three new components on the driver side, as depicted in Fig. 5: (1) The *Auto-cacher*, which makes caching decisions (§6). (2) The *Cluster--cache Manager*, which is the interface between the driver and the executors (§7). (3) The *Scale-out Manager*, which manages resources (§8). We also introduce a new component on the executor side, i.e. the *Executor-cache Manager*, which manages cache evictions (§7.1.2), blocks migration (§8.2) and communications. From this point onwards, *executor* refers to the default executor functionalities while *Executor-cache Manager* refers to the added component of ours.

5.1 Life Cycle

Before Job Execution. The Auto-cacher first evaluates the caching decisions of the previous job whereby it keeps the *beneficial datasets* cached and unpersists the *non-beneficial datasets* (§6.1.1). A dataset is *beneficial* for caching if its computation is costly. Next, the Auto-cacher traverses the DAG of the current job to cache new datasets, i.e. *recently cached datasets*, based on their number of computations and call-sites (§6.1.2). Later, it unpersists any cached datasets that do not exist in the current job's DAG, depending on the *unpersistence-distance* of their call-sites (§6.1.3). Lastly, for each recently cached dataset, each Executor-cache Manager is notified of the *target number of blocks* that the executor shall cache (§6.1.4).
During Job Execution. While an executor runs tasks and caches blocks, the Executor-cache Manager profiles the computation time and size of each block (§7.1). It first uses the metrics of a few blocks to predict whether the target number of blocks fits in the executor cache and, if not, proactively sends a notification, i.e. a *cache-limitation vote*, to the Cluster-cache Manager (§7.1.1). When the number of votes across all executors exceeds a certain threshold, the Auto-cacher conducts an *early evaluation* to check whether the recently cached datasets are beneficial or not. Accordingly, the Scale-out Manager extends the cluster cache by adding new executors to cache the blocks of the beneficial datasets (§8.1). When a new executor arrives, each Executor-cache Manager is notified to migrate some of its cached blocks to the newly arrived executor.

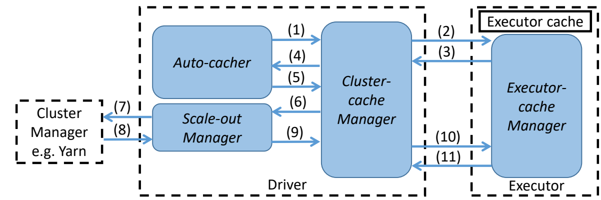

Figure 5: Life cycle of Agile-Ant.

After Job Execution. Each Executor-cache Manager sends its cache status to the Cluster-cache Manager, which then notifies the Executor-cache Managers of the overloaded executors to migrate some of their cached blocks to the less-loaded executors (§7.2).

5.1.1 *Illustrative Example.*

We select LIR (see Fig. 1 and Fig. 2) to explain how the three components of Agile-ant work together to optimize the execution cost of an actual run on our cluster (§10.1). Later in §6, §7 and §8, we justify our design decisions. The execution starts with a single executor E_0. We consider two cases, namely LIR_1 and LIR_2, where the maximum number of executors is 6 and 12 respectively.

Before job_1 is executed, the Auto-cacher detects that D_{0-5} will be computed for the second time (first in job_0). Therefore, it decides to cache D_5 (i.e. annotates it as cached), as caching it saves recomputing it and its parents (§2.0.2). With this annotation, whenever an executor computes a block of D_5, it will cache the block. At this point, D_5 is considered as a *recently cached dataset* (i.e. not evaluated). The Auto-cacher then communicates the caching decision and number of blocks of D_5 (400 in this example) to the Cluster-cache Manager (see (1) in Fig. 5), which notifies each Executor-cache Manager (see (2) in Fig. 5) with the number of blocks the executor shall cache. Since E_0 is the only executor, it shall cache 400 blocks.

E_0 then starts executing job_1 in tasks. Each task computes and caches a block of D_5, while the Executor-cache Manager of E_0 profiles the block's computation time and size. After a few tasks (8 in this example), the Executor-cache Manager conducts a *local early evaluation*, where it calculates that the size of the 400 blocks will be 35.5 GB based on the size of the 8 blocks while E_0's maximum capacity (i.e. M in §2.0.3) is 7.02 GB. As a result, it sends a cache-limitation vote to the Cluster-cache Manager, consisting of the execution time of the 8 tasks and the metrics of the 8 blocks (see (3) in Fig. 5). Based on a majority consensus, the Cluster-cache Manager triggers the *global early evaluation*, where it sends the average execution time of the 8 tasks and the average computation time of the 8 blocks to the Auto-cacher (see (4) in Fig. 5), which evaluates that D_5 is beneficial and notifies the Cluster-cache Manager (see (5) in Fig. 5). Accordingly, the Cluster-cache Manager requests for 5 executors from the Scale-out Manager (see (6) in Fig. 5), which starts a *scale out round* by requesting for E_{1-5} from the cluster manager (see (7) in Fig. 5). This round ends when the 5 executors arrive. By the time a new executor arrives, E_0 would have already cached some blocks. To ensure uniform distribution of blocks among executors, upon the arrival of each new executor (see (8) in Fig. 5), the Scale-out Manager notifies the Cluster-cache Manager (see (9) in Fig. 5), which in turn notifies E_0 to migrate a few blocks to the new executor (see (10) in Fig. 5). E_0 notifies the Cluster-cache Manager when the blocks migration ends (see (11) in Fig. 5). The Cluster-cache Manager uses this notification to trace incomplete data migrations.

Table 1: BenefitRatio of datasets (LIR). Time in milliseconds.

Dataset	Avg CT of a Block	Avg Task Execution Time	BenefitRatio	Already Cached
D_5	8473	9118	0.932	-
D_{35}	53	586	0.091	D_5
D_{41}	495	536	0.924	D_5

When job_1 ends i.e. E_{0-5} execute all the 400 tasks, each Executor-cache Manager sends the metrics of its executor's tasks and cached blocks to the Cluster-cache Manager for the *final evaluation* of D_5. Before job_4 is executed, the Auto-cacher caches D_{35} and during the early evaluation, it evaluates D_{35} as non-beneficial. Before job_5 is executed, the Auto-cacher (i) conducts a final evaluation of D_{35} and unpersists it, and (ii) caches D_{41} and, during the early evaluation, evaluates it as beneficial. In the case of LIR_1, after job_5 and the final evaluation of D_{41}, the Auto-cacher switches to the resource-constrained mode since the maximum number of executors is reached with 6 executors and unpersists D_5 as D_{41} is more beneficial. Before job_{20} is executed, the Auto-cacher unpersists D_{41} and caches D_5 again i.e. D_5 will be recomputed in job_{20}. While in the case of LIR_2, during the execution of job_5, the Auto-cacher remains in the scale-out mode and the Scale-out Manager requests for 6 more executors. Then, D_{41} is unpersisted before job_{20} is executed. **Discussion.** The above example shows how Agile-ant addresses the challenges of caching decisions (§3.1) and cluster configurations (§3.2) by observing the metrics of each cached dataset. In the case of LIR_2, it extends the cluster size to cache both datasets while for LIR_1, it adapts the caching decision by unpersisting D_5 (§3.3).

5.2 Design Considerations

Agile-ant releases executors that: (i) end up not caching any block after unpersisting datasets, or (ii) become idle when parallelism is reduced. Furthermore, Agile-ant conducts only one early evaluation per job to avoid adding further more executors for the recently cached datasets, which are not yet fully evaluated (§6.1.1).

6 AUTO-CACHER

The Auto-cacher makes caching decisions based on the Agile-ant mode: the *scale-out mode* or the *resource-constrained mode*. In the scale-out mode (the default), the Auto-cacher keeps cached datasets as long as they are used and requests for more executors in case of cluster cache limitation. When the number of allocated executors reaches the maximum number of available executors, it switches over to the resource-constrained mode, where instead of scaling out, it unpersists some datasets in case of cluster cache limitation.

6.1 Before Job Execution

6.1.1 Step 1: Evaluate the Previous Job's Recently Cached Datasets. The Auto-cacher starts by evaluating the recently cached datasets of the previous job. The reason for this evaluation is that the Auto-cacher made the caching decisions before the execution of the previous job when the computation times and sizes of datasets were not known (§6.1.2). Upon obtaining the computation times and sizes of all blocks after the execution of the previous job, the Auto-cacher conducts the *final evaluation*, where it calculates the *benefit* and the *benefit ratio* of each dataset D_i with ID i as follows:

$$\text{Benefit}_i = \frac{\text{computation time}_i}{\text{size}_i} \quad (1)$$

$$\text{BenefitRatio}_i = \frac{\text{Average computation time of } D_i \text{ blocks}}{\text{Average execution time of } D_i \text{ tasks}} \quad (2)$$

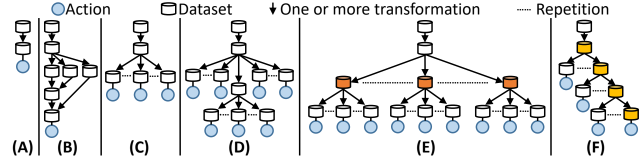

Figure 6: Typical classes of big data applications. Reused similar datasets shown in the same color.

where D_i tasks are those that compute D_i blocks and computation time$_i$ is the time required to compute D_i starting from its immediate cached parent (if any) or the root dataset. This explains why the benefit of a dataset increases when its *direct cached parent* is unpersisted (see Eq. (3)). If BenefitRatio$_i$ is greater than a configurable threshold (0.25 by default), the Auto-cacher annotates D_i as *beneficial*, annotates the call-site of D_i as *beneficial*, and keeps D_i cached. Otherwise, the Auto-cacher unpersists D_i and annotates it as *non-beneficial* to avoid caching it again. A threshold of 0.25 means the dataset is considered beneficial if caching it saves at least 25% of the job's execution time. The lower the threshold, the more the datasets that are cached and, thus, the more the executors that are added. Table 1 shows how the BenefitRatio is calculated for each dataset in the LIR example (§5.1.1). D_5 is beneficial as caching it saves 93% of the execution time. After caching D_5, the average task execution time for computing D_{35} and D_{41} drops significantly. A dataset's computation time is calculated starting from its direct cached parent. That is why the BenefitRatio of D_{35} is lower than that of D_5 and thus not cached despite D_{35} being a child of D_5. If three similar datasets are unpersisted consecutively because they are non-beneficial, the Auto-cacher adds their call-site to the list of non-beneficial call-sites to avoid caching similar datasets. The reason for delaying adding a call-site to this list until after three unpersists is that the benefit of similar datasets varies especially in the case of nested loops where a child dataset is computed starting from its parent that was computed at the same call-site in the previous loop iteration, e.g. Class F applications in Fig. 6.

After the final evaluation, there might be new datasets evaluated as beneficial. If the total size of the beneficial datasets becomes larger than the cluster cache capacity and Agile-ant is in the resource-constrained mode (§8), the Auto-cacher unpersists the least beneficial datasets, based on their benefits (Eq. 1), one by one until they all fit in the cluster cache. When a dataset p is unpersisted, the benefit of each cached child c of p increases if c is a *direct cached child* of p (i.e. there is no cached dataset between p and c) as follows:

$$\text{Benefit}_c = \frac{\text{computation time}_c + \text{computation time}_p}{\text{size}_c} \quad (3)$$

Also, the benefit of c reduces if p is cached again. Consider three datasets: A is a parent of B, B is a parent of C, and B and C are cached. The benefit of B is updated if A is cached/unpersisted but the benefit of C is not because it will be computed starting from B, whether A is cached or not. Hence, whenever a dataset is cached/unpersisted, the benefits of *only* its direct cached child datasets are updated.

6.1.2 Step 2: Traverse the Current Job's DAG. We analyze 130 real-world applications [9] and group them into 6 classes based on their common execution patterns (see Fig. 6). We use Algorithm 1 and the example in Fig. 7 and Table 2 to explain a job's DAG traversal. For each dataset, the Auto-cacher profiles: (i) Facts, which are meta

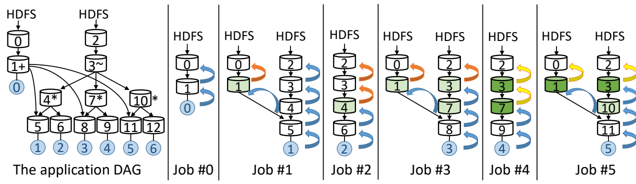

Figure 7: DAG traversal by the Auto-cacher.

Algorithm 1: Traversal of Job's DAG

```
   Input :Current dataset D, Traversal type T
1  set D.current_job to true
2  if T is Type C then
3  |   foreach dep ∈ D.parent_dependencies do
4  |   |   Traverse to dep.parent with Type C
5  |   return
6  D.last_usage_job_id = id of current job
7  if D is cached then
8  |   foreach dep ∈ D.parent_dependencies do
9  |   |   Traverse to dep.parent with Type C
10 |   return
11 D.num_computations++
12 if T is Type B then
13 |   foreach dep ∈ D.parent_dependencies do
14 |   |   if dep is wide && dep is computed then
15 |   |   |   Traverse to dep.parent with Type C
16 |   |   else
17 |   |   |   Traverse to dep.parent with Type B
18 |   return
19 condition 1 ←− D.call-site is beneficial
20 condition 2 ←− D.call-site is not marked as non-beneficial &&
     D is not marked as non-beneficial && D.num_computations > 1
21 if condition_1 || condition_2 then
22 |   cache(D)
23 foreach dep ∈ D.parent_dependencies do
24 |   if dep is wide && dep is computed then
25 |   |   Traverse to dep.parent with Type C
26 |   else if condition_2 then
27 |   |   Traverse to dep.parent with Type B
28 |   else
29 |   |   Traverse to dep.parent with Type A
```

data e.g. id and call-site. (ii) DAG data, which is updated during the DAG traversal e.g. status (cached, unpersisted or N/A), number of computations (#C), last usage job id (LUJI) and whether the dataset is used in the current job (CJ). (iii) Metrics, which are collected from the Cluster-cache Manager e.g. computation time and size.

As each dataset points back to its parent (§2.0.1), the Auto-cacher traverses the job's DAG recursively using these dependencies, starting from the action. At the beginning i.e. **before caching any dataset**, the traversal Type A (denoted by ⊃ in Fig. 7) caches any dataset that either belongs to a beneficial call-site (Line 19 of Algo. 1, e.g. D_7 in job_3) or satisfies the following conditions (Line 20 of Algo. 1): (1) It is computed more than once (e.g. D_1 in job_1). (2) It does not belong to the list of non-beneficial datasets. (3) Its call-site does not belong to the list of non-beneficial call-sites. The Auto-cacher selects a dataset for caching without knowing its size and computation time. Thus, evaluating whether it is beneficial or not is not possible at this point. It is only after the job execution, i.e. when the dataset is fully computed, that the Auto-cacher conducts a final evaluation to keep the dataset cached if it is beneficial, or unpersist it otherwise (§6.1.1). The Auto-cacher adds the newly cached dataset to the list of recently cached datasets i.e. cached datasets that still need evaluation. As an exception, if the dataset has been cached previously (i.e. already evaluated as beneficial) but

Table 2: DAG data. Green-colored and red-colored cells refer to cache and unpersist instructions respectively.

	D_0			D_1			D_2			D_3			D_4		
	#C	CJ	LUJI	#C	CJ	LUJI	#C	CJ	LUJI	#C	CJ	LUJI	#C	CJ	LUJI
Job_0	1	T	0	1	T	0	-	-	-	-	-	-	-	-	-
Job_1	2	T	1	2	T	1	1	T	1	1	T	1	1	T	1
Job_2	2	F	1	2	F	1	2	T	2	2	T	2	2	T	2
Job_3	3	T	3	3	T	3	3	T	3	3	T	3	2	F	2
Job_4	3	F	3	3	F	3	3	T	3	3	T	3	2	F	2
Job_5	3	T	3	3	T	5	3	T	3	3	T	5	2	F	2

unpersisted afterward, the Auto-cacher does not add it to the list. This happens when a beneficial dataset is not used in a few jobs after being cached and used again in later jobs (e.g. D_1 in job_3). To avoid cache-unpersist cycles in such cases, the dataset's *unpersistence-distance* (1 by default) is calculated (i.e. current job id - LUJI) and associated with its call-site if it is greater than the call-site's current unpersistence-distance. For example, the unpersistence-distance of D_1 in job_3 is set to 2 (job_3 ID - job_1 ID) and is associated with its call-site denoted by "+" and therefore D_1 is not unpersisted in job_4 despite its absence in job_4's DAG (see Fig. 7 and Table 2).

After the Auto-cacher caches a dataset (denoted by the light green color), the Type B traversal (denoted by ⊃ in Fig. 7) continues towards the dataset's parents (Line 27 of Algo. 1) and increments their #C because the child has to be computed first, starting from them, before being cached. The Auto-cacher does not cache any parent during this traversal to avoid caching consecutive datasets. Hence, the Auto-cacher prunes candidates for caching e.g. parents of a cached dataset are no more candidates. So D_0 and D_1 are not cached together as D_0 will not be recomputed after caching D_1.

If the Auto-cacher **traverses a cached dataset** (i.e. already cached in a previous job, denoted by the dark green color), the Auto-cacher continues traversing with the Type C traversal (denoted by ⊃ in Fig. 7; Line 9 of Algo. 1) to mark that the dataset's parent is part of the current DAG (Line 1 of Algo. 1) and, thus, avoid unpersisting it in the next step (§6.1.3). The Auto-cacher does not cache the parent or increment the parent's #C because the parent will not be computed since the child is already cached. Similarly, as shuffle blocks are persisted (§2.0.4), by reaching a wide transformation that has been computed in a previous job, the Auto-cacher considers it as already cached data (Lines 15 and 25 of Algo. 1).

Since the DAGs of subsequent jobs are not yet available (§2.0.1), the Auto-cacher might cache a dataset that will not be used later on. This would decrease the efficiency of AGILE-ANT and all caching solutions [10, 38, 41, 53]. However, Agile-Ant unpersists cached datasets that are not used anymore (§6.1.3). Note that AGILE-ANT is not limited to iterative applications where jobs (i.e. iterations) have similar DAG topology because it caches/unpersists datasets based on their reuse pattern whether the jobs are similar or not.

6.1.3 Step 3: Unpersist Unused Cached Datasets. After completing the DAG traversal, the Auto-cacher unpersists any cached dataset that does not belong to the DAG of the current job (i.e. not traversed via any of the three traversal types) and has not been used for up to the unpersistence-distance of its call-site. For example, the Auto-cacher unpersists D_1 in job_2 because the DAG of job_2 does not include D_1 and the difference between job_2 ID and the LUJI of D_1 (i.e. 1) has reached the unpersistence-distance of D_1's call-site ("+"), i.e. 1. Contrarily, in job_4, the Auto-cacher keeps D_1 cached because the unpersistence-distance of D_1's call-site becomes 2 (see above).

6.1.4 Step 4: Notify Executors. Lastly, the Auto-cacher notifies the Cluster-cache Manager, which in turn notifies each Executor-cache Manager about the target number of blocks (#Blocks) from each recently cached dataset that the respective executor is expected to cache. For a dataset D_i with ID i, $\#\text{Blocks}_i = \lceil \frac{\text{number of blocks of } D_i}{\#\text{executors}} \rceil$.

7 CLUSTER-CACHE MANAGER

7.1 During Job Execution

Tasks perform the same computation on different blocks, in the form of an execution lineage, starting from the last cached block (if any) in the lineage. Fig. 8 shows how $task_0$ in job_4 of the LIR example in Fig. 1 computes and caches a block from each of D_5 and D_{41}. If D_{41_0} (a block of D_{41} with index 0) is cached, $task_0$ fetches it and executes $action_4$ starting from it. Otherwise, $task_0$ computes D_{41_0} from D_{5_0}. D_{5_0} is fetched if cached or computed otherwise, and the process goes on. The Executor-cache Manager profiles the computation start and end timestamps of D_{41_0} (i.e. Ts_0 and Ts_3 respectively) as well as D_{5_0} (i.e. Ts_1 and Ts_2 respectively), from which it obtains the computation time CT of each block as follows:

$$CT_{D5_0} = Ts_2 - Ts_1 \qquad CT_{D41_0} = (Ts_3 - Ts_0) - CT_{D5_0}$$

If D_{5_0} is already cached, CT_{D5_0} drops to zero. Note that this timestamp-based profiling method is not applicable for non-blocking transformations (§2.0.6). This is why collecting metrics is limited to only cached datasets because caching is a blocking operation (§10.4).

7.1.1 Early Evaluation. After an executor computes and caches a certain configurable number of blocks ($2 \times$ #cores per executor by default), the Executor-cache Manager calculates the size of the target blocks Target Size_{D_i} of each recently cached dataset D_i as:

$$\text{Target Size}_{D_i} = \frac{\text{total size of } D_i \text{ computed blocks} \times \#Blocks_i}{\#\text{computed blocks of } D_i}$$

The higher the configurable number above, the more the local early evaluation is delayed and the higher the accuracy due to more computed blocks. Each Executor-cache Manager calculates the *required cache size* as the size sum of blocks that are already cached from previous jobs and the predicted sizes of target blocks of all recently cached datasets. If the required cache size is larger than the actual caching capacity (§2.0.3), the Executor-cache Manager sends a cache-limitation vote to the Cluster-cache Manager including the *blocks metrics list*. Each item in the list comprises (i) the block id, (ii) whether the block is cached, evicted or failed to be cached (§7.1.2), (iii) the id of the task that computes the block - to get the task execution time afterward, (iv) the block size and computation time. If the number of votes across all Executor-cache Managers exceeds a predefined configurable threshold ($\lceil \frac{\#executors}{3} \rceil$ by default), the Cluster-cache Manager predicts the computation time CT and size of each dataset D_i based on its blocks D_{ij} metrics collected from each *voter* i.e. Executor-cache Manager j as follows:

$$\text{Predicted size of } D_i = \frac{\sum_{j=1}^{\text{voter}} \text{total size of } D_{ij}}{\sum_{j=1}^{\text{voter}} \#D_{ij}} \times \#D_i \text{ blocks}$$

$$\text{Predicted CT of } D_i = \frac{\sum_{j=1}^{\text{voter}} \text{total CT of } D_{ij}}{\sum_{j=1}^{\text{voter}} \#D_{ij}} \times \#D_i \text{ blocks}$$

Note that increasing the value 3 mentioned above results in the global early evaluation being conducted with less delay and on a fewer number of blocks. The CT of a dataset represents the cumulative time across all cores to compute it. Therefore, unlike *response time*, CT is not divided by #cores. The Cluster-cache Manager then sends the predicted size and computation time of each recently cached dataset to the Auto-cacher, which evaluates if the dataset is beneficial or not (Eq. (1) and Eq. (2)). If so, the Cluster-cache Manager calculates the number of required executors as follows:

$$\#\text{Req. executors} = \lceil \frac{\sum_{i=1}^{\text{evaluated datasets}} \text{size of } D_i}{\text{executor actual cache capacity}} \rceil - \#\text{executors}$$

where the evaluated datasets are the datasets cached in previous jobs and those evaluated as beneficial in the early evaluation. The Cluster-cache Manager then sends a *scale out request* to the Scale-out Manager (§8) and notifies all Executor-cache Managers of the early evaluation results to maintain their eviction policy (§7.1.2).

Discussion. The early evaluation avoids delaying adding executors, which would result in cache limitation and lead to recomputations in the next job. Note that the early evaluation is not decisive because it relies on prediction using a few blocks. In the case of data skews, the size and computation time of blocks might vary and, therefore, the final evaluation (§6.1.1) is required when the metrics of all blocks are available after the job execution (§7.2).

7.1.2 Executor Cache Management. The Executor-cache Manager enforces a cache eviction policy when the executor cache is fully utilized to the extent that some blocks are evicted or not successfully cached. This situation occurs when: (i) Tasks require more execution memory (§2.0.3), (ii) Blocks are not uniformly distributed among executors. (iii) A scale out request has not been made yet because #voters is not enough. (iv) There are delays while adding new executors (§8.1). The eviction policy assigns the highest priority to beneficial datasets (i.e. from a previous job), followed by the recently cached datasets evaluated as beneficial in the early evaluation, then the recently cached datasets that have not yet been evaluated, and finally the recently cached datasets evaluated as non-beneficial. Within blocks of the same priority, the policy prioritizes parent datasets over their children since the evicted child blocks can be recomputed later from their parent blocks. As a result of this policy, each block that the executor is supposed to cache either: (a) is cached and remains cached, (b) is cached but evicted later, (c) fails to be cached. A block's computation time and size can be profiled in all three cases and the first two cases, respectively. If a block fails to be cached, its size cannot be obtained since it is not a complete block in memory. Thus, the Executor-cache Manager predicts the complete block size by linear scaling based on the number of tuples in the incomplete block and the size of the incomplete block.

7.2 After Job Execution

After a job is executed, each Executor-cache Manager sends its blocks metrics list (§7.1.1) to the Cluster-cache Manager, which calculates the metrics of each recently cached dataset and sends them to the Auto-cacher for the final evaluation (§6.1.1). Then if there is no pending data migration or scale out round, the Cluster-cache

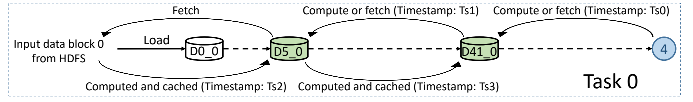

Figure 8: Computing and caching multiple blocks in a task.

Manager balances the caching load among executors depending on whether there are *overloaded executors* (§7.2.1) or not (§7.2.2).

7.2.1 Borrow Remote Cache. *Overloaded executors* are executors that have blocks evicted or failing to be cached. The Cluster-cache Manager uses the blocks metrics list of each executor to search for overloaded and less-loaded executors, from which to *borrow* space. It then notifies each overloaded executor to migrate blocks to a less-loaded executor (§8.2). The notification consists of the ID of the less-loaded executor and the size of the blocks to migrate, which equals half of the difference of the size of cached blocks in both executors. The cluster cache is enough to cache any block of a beneficial dataset: in the scale-out mode, more executors can be added, and in the resource-constrained mode, the less beneficial datasets are already unpersisted (§6.1.1). Agile-ant initially considers the actual cache capacity of each executor to be the whole unified memory (M §2.0.3). However, on an overloaded executor (where caching capacity reaches the limits between R and M), Agile-ant sets the actual cache capacity to be equal to the total size of cached blocks in the overloaded executor and, correspondingly, recalculates the cluster cache size and scales out if needed. This is how Agile-ant is adaptive towards changes in the execution memory utilization and how it covers various applications with different memory footprints.

7.2.2 Balance Executors' Cache. Even without any overloaded executors, some executors might cache more blocks than others (i.e. in terms of the total size of cached blocks). And even without cache evictions, their tasks will take longer execution time than those of other executors. These straggler executors then become the performance bottleneck since a job execution finishes only when the last executor executes all its tasks. Thus, the Cluster-cache Manager iteratively migrates blocks from the executor with the maximum size of cached blocks to the one with the minimum size if the former caches 10% (experimentally selected) more than the average size of cached blocks across executors and the latter caches 10% less.

8 SCALE-OUT MANAGER

The Scale-out Manager receives requests from the Cluster-cache Manager and, if Agile-ant is in the scale-out mode, adds executors without exceeding the maximum number of executors which is given by administrators or cluster managers e.g. YARN [47]. If the number of executors after the scale out reaches the maximum number, Agile-ant switches over to the resource-constrained mode.

8.1 Scale Out

There are factors to consider when requesting new executors. (1) Cluster managers take time before allocating new executors due to resource negotiation. (2) Requested executors arrive at different times. (3) The delays in the arrival of executors are unpredictable. The Scale-out Manager initiates a scale out round only when all requested executors in the previous round have arrived. Upon the arrival of a new executor (N), the Scale-out Manager notifies the Cluster-cache Manager, which in turn notifies the Executor-cache Manager of each executor (O) that is not among the newly added executors to migrate part of its cached blocks to N as follows:

$$migration\text{-}ratio = \frac{1}{\#executors_{bs} + \#executors_{new}} \qquad (4)$$

$$S(blocks_{ON}) = S(all\ cached\ blocks\ in\ O) \times migration\text{-}ratio$$

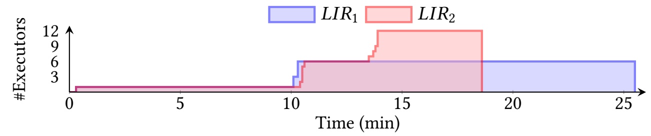

Figure 9: Execution costs of LIR.

where $S(blocks_{ON})$ is the size of blocks to migrate from O to N and $\#executors_{bs}$ and $\#executors_{new}$ are the number of executors before scaling out and the newly added executors respectively. For example, if 3 executors are already allocated and then 2 new executors are added, each of the initial 3 executors will migrate 1/5 of its cached blocks to each new executor. By the end of the scale out round, each executor will have around 3/5 of the cached blocks.

8.2 Blocks Migration

The Cluster-cache Manager notifies the Executor-cache Manager of a *sender* executor to migrate some of its cached blocks to a *receiver* executor. If the migration is triggered by a scale out request (§8.1), the notification includes the id of the receiver executor and the *migration-ratio*, from which the sender's Executor-cache Manager calculates the size of blocks to migrate. Whereas if the migration is triggered by borrow remote cache or balance executors cache (§7.2), the notification already includes the size of blocks to migrate.

8.2.1 Selected Blocks for Migration. The sender may cache blocks of different datasets. Consider blocks migration starting in LIR (§5.1.1) while D_5 and D_{41} are cached. The Executor-cache Manager defines a partition index counter starting from zero and searches for the blocks lineage, i.e. all blocks having this partition index (D_{5_0} and D_{41_0}). If no block is found, it searches for the next (D_{5_1} and D_{41_1}). For two blocks of the same lineage where the parent block is cached while the child block is not (e.g. D_{5_0} is cached and D_{41_0} is not), the parent is migrated. This means the child is either evicted, not successfully cached or not yet computed. (§7.1.2). But if both are cached, both are migrated. This approach avoids having the majority of D_5 blocks in a set of executors while those of D_{41} are in another set. Without this, problems would arise in the next job when the tasks dispatcher selects the executors of the latter set to execute the tasks since they have the blocks of the child D_{41}, and leave the executors of the former set not utilized. The selection of blocks continues until the target migration blocks size is reached.

8.2.2 Migrating Selected Blocks. The sender's Execution-cache Manager migrates selected blocks in two batches using the blocks transfer API (§2.0.2). In the first batch, it migrates the blocks of the first blocks lineage only to the receiver and evicts them from the sender's cache. It then compares the time required to transfer 1 MB with the time required to compute 1 MB of the migrated blocks. If the former is larger, it unpersists all the selected blocks without migrating them i.e. it cancels the second batch. In this case, they will be recomputed later, which is less costly than transferring them. Otherwise, it migrates the remaining selected blocks in the second batch and unpersists them afterwards. Finally, it sends an *end-of-migration* notification to the Cluster-cache Manager. This notification is important so that load balancing of blocks (§7.2) is not conducted if there is a pending data migration. This two-batch approach of blocks migration makes Agile-ant adaptive to changes in the cluster utilization by multiple tenants during the actual run.

Table 3: Details of evaluated applications.

Application	Input data (size)	#jobs	Library	Class	Properties
Analyzing Co-Occurrence Networks (ACON)	MEDLINE [1] (67.9 MB)	44	Advanced Analytics	C&F	Many small cached datasets
Anomaly Detection in Network Traffic (ADNT)	KDD Cup 1999 [1] (708.2 MB)	1349	Advanced Analytics	D&E	Many small cached datasets
Predicting Forest Cover (PFC)	Covtype [1] (71.7 MB)	734	Advanced Analytics	C&D&E	Many small cached datasets
Recommending Music (RM)	Audioscrobbler [1] (463.2 MB)	474	Advanced Analytics	D&E&F	Many small cached datasets
Connected Components (CC)	road-road-usa.mtx [43] (469.7 MB)	58	GraphFrames	C&E	Many small cached datasets
Label Propagation (LP)	road-road-usa.mtx [43] (469.7 MB)	8	GraphFrames	C&F	Many large cached datasets
Page Rank (PR)	road-road-usa.mtx [43] (469.7 MB)	26	GraphFrames	F	Many large cached datasets
Strongly Connected Components (SCC)	road-road-usa.mtx [43] (469.7 MB)	126	GraphFrames	C&F	Many large cached datasets
Nweight (NW)	HiBench datasets [6] (849.1 MB)	1	HiBench	B	Few large cached datasets
Scala Page Rank (SPR)	HiBench datasets [6] (2.8 GB)	1	HiBench	B	One large cached dataset
Scala Sort (Sort)	HiBench datasets [6] (9.5 GB)	1	HiBench	A	No cached datasets
Word Count (WC)	HiBench datasets [6] (30.3 GB)	1	HiBench	A	No cached datasets
Alternating Least Squares (ALS)	HiBench datasets [6] (9.3 MB)	35	Spark MLlib	C&F	Many small cached datasets
Dense K-means (DKM)	HiBench datasets [6] (18.5 GB)	103	Spark MLlib	D	Few large cached datasets
Gradient Boosted Trees (GBT)	HiBench datasets [6] (91.7 MB)	202	Spark MLlib	E&F	Many small cached datasets
Latent Dirichlet Allocation (LDA)	HiBench datasets [6] (246.6 MB)	47	Spark MLlib	C	Many small non-beneficial datasets
Linear Regression (LIR)	HiBench datasets [6] (44.7 GB)	156	Spark MLlib	D	Few large cached datasets
Logistic Regression (LOR)	HiBench datasets [6] (7.5 GB)	68	Spark MLlib	D	Few large cached datasets
Principal Components Analysis (PCA)	HiBench datasets [6] (16.8 GB)	647	Spark MLlib	C	No cached datasets
Random Forest Classifier (RFC)	HiBench datasets [6] (7.5 GB)	18	Spark MLlib	D	Few large cached datasets
Support Vector Machine (SVM)	HiBench datasets [6] (37.3 GB)	106	Spark MLlib	D	One large cached dataset

9 DISCUSSION

9.1 Distributed In-Memory Processing

Frameworks on clouds e.g. Spark[55], Snowflake[18], Databricks[33], Flink[16], Storm[46], etc., are conceptually similar, although different in implementation. They run applications as dataflows on a driver and executors to process distributed datasets in parallel. They also utilize executor memory for caching as per eviction policies and support elasticity by adding/releasing executors on demand.

Although Agile-ant is implemented on Spark, its main concepts are applicable with minor modifications to any framework that supports (1) caching datasets in memory based on eviction policies and (2) elasticity with data migration between executors. A minor modification, for example, is if a framework does not cache shuffled data blocks (§2.0.4), the Auto-cacher shall not consider a computed wide transformation as cached during the traversal. Also, features of Agile-ant do not need to be re-implemented if a framework already supports them. For example, the Catalyst optimizer of Spark-SQL [14] provides the application DAG and the size of datasets before the execution. Hence, Agile-ant determines the reusability of datasets and the required cluster size to cache them before the actual run.

9.2 Shared Computing Clusters

Agile-ant optimizes the execution cost of a single application run. And since the optimal cluster size of each application run is not known in advance (§1), scheduling resources on demand between multiple concurrent application runs might result in some of them waiting for resources occupied by others. This could be overcome using Agile-ant by the cluster manager (e.g. Yarn) asynchronously notifying the Scale-out Manager of each run with the new available number of executors whenever executors are occupied or released. For example, consider two application runs A and B on our 12-node cluster (§10.1), with the maximum number of executors set to 12 for both runs. Initially, each run starts with one executor. Then A requests 10 executors and, later on, B requests 10 executors. As a result, the scale-out round of B will not be completed until A releases the executors that B requires, which increases the execution time and cost of B significantly. In this case, when A occupies 10 executors, B is notified of the available number of executors and, accordingly, switches to the resource-constrained mode and when A releases executors, B switches back to the scale-out mode.

10 EVALUATION

10.1 Workloads and Experimental Setup

We conduct all experiments on our 12-node Spark cluster. Each node is equipped with an Intel Core i5 CPU running at 4x 2.90 GHz, 16 GB RAM, 1 TB disk, 1 GBit/s LAN, HDFS and runs Hadoop MapReduce 2.7, Spark 3.1.2, Java 8u102 and Apache YARN. We allocate 12 GB of memory and 4 cores to each executor in all experiments.

Applications. Table 3 shows the 21 applications we study from various libraries and the class of Fig. 6 for each application.

Execution Costs. We calculate the execution cost of a run as the sum of the allocation times of all executors. For example, the execution cost of LIR_2 and LIR_1 (§5.1.1) with 100 iterations (represented by the shaded areas in Fig. 9) is 86.1 and 101.5 executor minutes respectively. LIR_2 is faster and cheaper than LIR_1 due to the re-computation of D_5 in job_{20} of LIR_1. Note that cloud providers use a pay-as-you-go pricing model. Thus, our way of calculating the execution costs represents the monetary costs on public clouds. However, as they vary in price rates and offers [39], the cost ratios in our comparisons might not accurately reflect the monetary cost ratios on public clouds, but the conclusions are similar. For example, the monetary costs of LIR_2 will still be less than those of LIR_1.

10.2 Agile-Ant vs Alternatives

We compare Agile-ant with three alternatives: (1) Our baseline i.e. *default runs* with the caching decisions of developers and the LRU policy. (2) An alternative i.e. *LRC runs* with the developers' caching decisions but with the LRC policy, in order to quantify how much cache eviction policies reduce execution costs. As these policies apply when more than one datasets are cached and the cluster cache is not enough to cache them all, we exclude applications whose default caching decisions cache one or no dataset and applications that cache multiple small datasets that fit in a single executor cache. (3) Another alternative i.e. *ReSpark runs* with ReSpark's caching

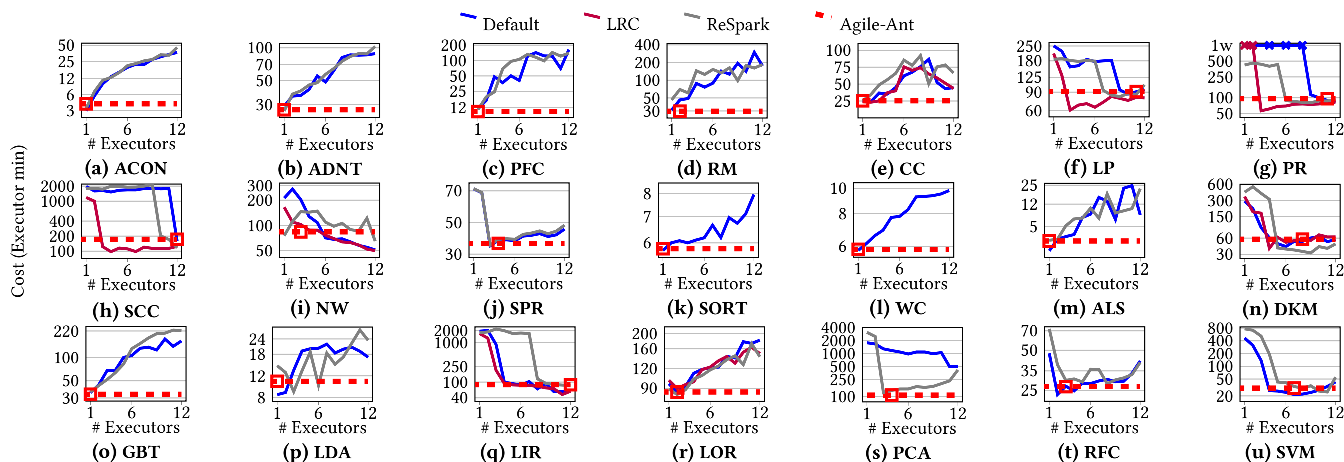

Figure 10: Default vs LRC vs ReSpark vs Agile-Ant runs.

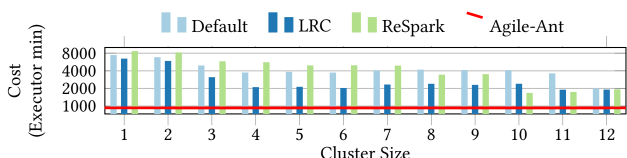

Figure 11: Dynamic vs Static cluster size.

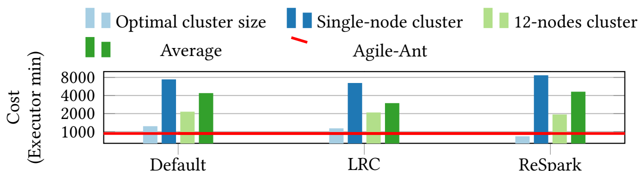

Figure 12: Agile-Ant vs Alternatives: Different cluster sizes.

decisions and the default LRU policy, so as to quantify how much autonomous caching reduces execution costs. In addition, we run each application using Agile-Ant (*Agile-Ant runs*). Since the three alternatives run on fixed-size clusters, we run each 12 times on all cluster sizes (1 to 12 executors), and as Agile-Ant is orthogonal to any fixed cluster size, we run each Agile-Ant run a single time starting from a single executor and set the maximum number of executors to 12. For each application, we show in Fig. 10 the execution cost of the default runs ( ), the LRC runs ( ) and the ReSpark runs ( ). We represent Agile-Ant runs as horizontal lines ( ) because they are orthogonal to any fixed cluster size. Moreover, since Agile-Ant runs start with a single executor and might end with more executors, we mark the number of executors when the Agile-Ant runs finish using a square ( ) on the horizontal lines.

From Fig. 10, we see that for Class A applications, the optimal and worst cluster sizes are 1 and 12 respectively. This is because adding more executors increases cost regarding the serial part of the application and data shuffling between more executors [13, 48]. The same applies to applications whose cached datasets fit in a single executor cache (ACON, ADNT, CC, PFC, RM, ALS, and GBT), only that their performance can be improved by ReSpark and Agile-Ant on a single-executor cluster by modifying the developers' caching decisions. For Class B applications, where the whole application DAG is visible before the application starts, ReSpark and Agile-Ant cache all reusable datasets. In the case of SPR, one dataset is reused and the developers cached it. Therefore, the default, ReSpark and Agile-Ant runs have the same performance. In NW, there are multiple reused datasets. Increasing the cluster size is thus beneficial to avoid evicting the beneficial datasets. On small cluster sizes, LRC improves the performance as it resolves the cyclic eviction of LRU (§2.0.5). Since ReSpark caches non-beneficial large datasets, large cluster sizes do not completely eliminate evicting beneficial datasets. The same applies to Class C applications (PCA and LDA) with the difference that Agile-Ant and ReSpark delay the caching decisions such that a single recomputation of datasets is required. As to Class D and Class E applications (DKM, LIR, LOR, RFC, and SVM), a limited cluster cache leads to evictions in default and ReSpark runs. Thus, a single-executor cluster is the worst for many applications. LOR is an exception because it is a compute-intensive application and recomputing reused datasets is not severe (similar to LDA §3.1). LRC improves the performance of these applications on small cluster sizes only slightly due to cache evictions that it cannot avoid completely. But Agile-Ant caches beneficial datasets and scales out while running the applications. In Class F applications (LP, PR, SCC), ReSpark somewhat improves the performance when there are cache limitations because it unpersists datasets based on their call-sites. However, as it keeps root datasets always cached (§10.5), on small cluster sizes, it cannot avoid cache evictions. LRC performs even better than ReSpark on small cluster sizes due to evicting blocks of parent datasets that are not needed anymore (see Fig. 6). Thus, it requires relatively less memory for caching and outperforms the others (including Agile-Ant) when the optimal cluster size is selected. However, if the child datasets do not fit in the cluster cache, the performance of LRC drops significantly. To avoid such a problem, Agile-Ant unpersists datasets based on their unpersistence distance (§6.1.1) and scales out proactively (§7.1).

Our findings are as follows. (1) The default runs outperform the others if developers make correct caching decisions and administrators select the optimal cluster sizes (e.g. SMV - 7 executors). (2) In Class F applications, LRC outperforms the others if administrators select the optimal cluster sizes (even if developers do not unpersist datasets) (e.g. PR - 3 executors). (3) ReSpark outperforms the others if developers do not cache beneficial datasets that fit in the cluster cache (e.g. DKM - 9 executors). (4) Agile-Ant achieves the optimal

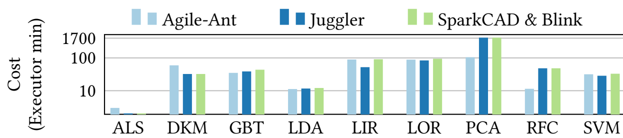

Figure 13: Agile-Ant vs Competitors.

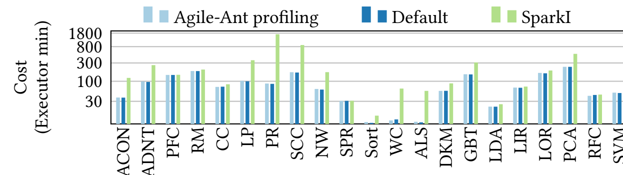

Figure 14: Overhead of profiling runtime metrics.

costs regardless of the initial caching decisions and cluster sizes. However, it has learning overhead (§10.7). (5) The optimal cluster size varies between applications. For example, 12 executors are the optimal for SCC but the worst for LOR while 1 executor is the optimal for GBT but the worst for DKM. Thus, fixing the cluster size for all runs leads to high costs. In Fig. 11, we sum the costs of all 21 runs of each alternative on each cluster size and compare them with the total sum of the 21 runs of Agile-Ant. On all cluster sizes, all the alternatives cost more than Agile-Ant. The alternative having the minimal cost, i.e. ReSpark on 10 executors, costs 1.8x compared with Agile-Ant. We also evaluate the costs if the administrators selected the optimal cluster sizes. We sum the costs of the alternative runs on their optimal cluster sizes and compare with the cost of Agile-Ant runs. Fig. 12 shows that Agile-Ant still outperform the default and LRC runs while ReSpark costs 88.1% compared to Agile-Ant. Note that selecting the optimal cluster size is difficult (§3.2). The average costs of the default runs, the LRC runs and the ReSpark runs are 4.6X, 3.1X, and 4.9X compared to the Agile-Ant runs respectively. (6) The reason for the poor performance of PR in the default runs on 8 executors is as follows. As more execution memory is required, some blocks that would be needed later are evicted, until reaching R (§2.0.3). Being a local minima problem, more execution memory is then required to recompute them, which leads to more evictions and GC rounds. But, for LRC runs on 3 executors, the R of each executor is enough to cache the blocks of needed child datasets since LRC evicts parent datasets. Thus, LRC has the best performance because when more execution memory is required, the needed blocks are not evicted. As Agile-Ant detects overloaded executors (§7.2.1), it recalculates the actual cache capacity, continuously, and scales out accordingly to increase the cluster memory, which is used for caching and execution as well.

10.3 Agile-Ant vs Competitors

Juggler is limited to ML applications (§4). Thus, we compare it with Agile-Ant only on Spark MLlib applications. We run each *schedule* [10] on its recommended cluster size and compare the schedule having the minimal cost with Agile-Ant. We also use SparkCAD [9] to recommend a schedule and Blink [8] to recommend the optimal cluster size. Fig. 13 shows the cost of application runs using Agile-Ant, Juggler and SparkCAD & Blink. Juggler and SparkCAD rely on previous runs which do not always reflect the actual application behaviour (§3). Nevertheless, both outperform Agile-Ant in applications like LIR and DKM because they recommend the same

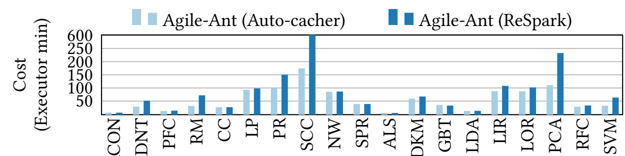

Figure 15: Auto-cacher vs ReSpark runs.

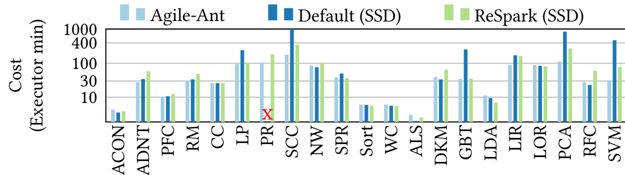

Figure 16: Scale out vs scale up.

caching decisions and cluster sizes as Agile-Ant but without learning overheads (§10.7). SparkCAD caches all reused datasets, leading to over-provisioning of executors. The total cost of Agile-Ant is 15.9% and 16.3% of those of Juggler and SparkCAD & Blink respectively. Juggler performs well on only 7 applications out of 21. And even for those 7 applications, any change in application parameters other than #examples and #features may drop its performance.

10.4 Agile-Ant Profiling Overhead vs SparkI

The runtime profiling of SparkI [10] is based on adding a profiling transformation in-between each pair of transformations. The added transformations are blocking ones (§2.0.6) and thus cause significant overhead. In contrast, Agile-Ant does not add any transformation (§7.1). To quantify the overhead of both, we run each application with its default caching decisions on 12 executors using the default implementation of Spark, SparkI and a modified version of Agile-Ant (Agile-Ant profiling), in which the Auto-cacher and the Scale-out Manager are disabled. As Fig. 14 shows, the additional cost of Agile-Ant profiling and SparkI is 1.4% and 198.7% respectively.

10.5 Auto-cacher vs ReSpark

Similarly to the Auto-cacher, ReSpark [38] caches and unpersists datasets on the fly based on their call-sites, making it suitable for Class E and Class F applications (see Fig. 6). We implement ReSpark in Agile-Ant to replace the Auto-cacher so as to compare the Auto-cacher (Agile-Ant runs) with ReSpark using its default LRU (Agile-Ant ReSpark runs) as shown in Fig. 15. We exclude applications that have no reused datasets. As ReSpark does not consider whether a dataset is beneficial for caching, it caches a lot of datasets. Also, it does not unpersist the first cached dataset of each call-site to have a measure of the number of times the datasets of the same call-site are reused. As a result, it keeps non-reused datasets cached. On average, *Agile-Ant* runs cost 57.6% of *Agile-Ant ReSpark* runs.

10.6 Scale out vs scale up

Instead of scaling out, the caching capacity can be extended by using storage e.g. SSDs. Therefore, we run each application on a single executor equipped with a 500 GB, SATA 3.3, 6.0 Gb/s SSD (enough to cache all datasets) using the developers' caching decisions (*Default SSD runs*) and ReSpark (*ReSpark SSD runs*) and compare them with the *Agile-Ant runs* (§10.2). On average, the cost of Default SSD runs and ReSpark SSD runs is 3.2X and 1.6X respectively compared with

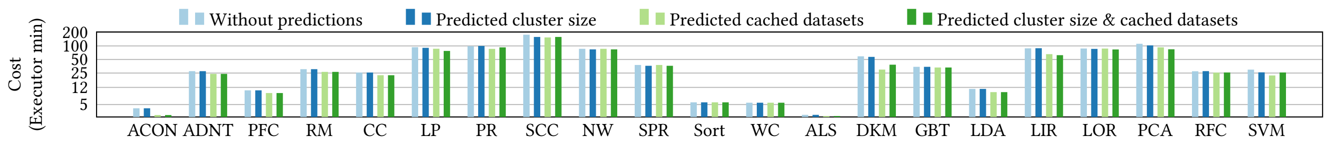

Figure 17: Agile-Ant runs with predictions.

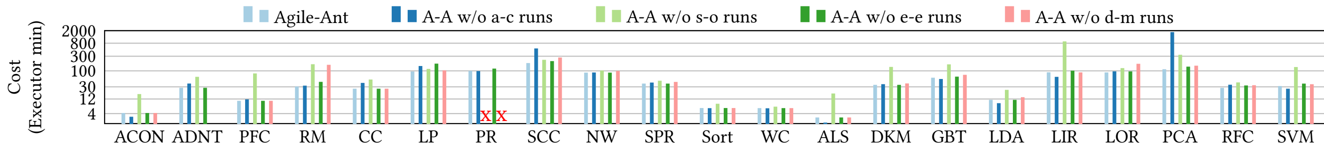

Figure 18: Evaluating the performance benefit of each feature of Agile-Ant.

the cost of Agile-Ant runs (see Fig. 16). In addition to the high memory bandwidth compared to the SSD bandwidth, the main reasons for having the high SSD runs costs are, firstly, persisting blocks on disks requires serializing them and reading them from disks requires deserializing them each time, which adds significant overhead [56]. Secondly, the execution memory of a single executor in SSD runs is limited in many runs and results in huge GC overhead (which forces us to terminate the execution of PR in Default SSD runs) while Agile-ant reacts towards changes in the execution memory utilization by scaling out (§7.2.1). The two problems above remain even with SSDs as fast as memory. This experiment shows that ReSpark still improves the performance with SSDs.

10.7 Learning Overhead of Agile-ant

The LIR example in §5.1.1 shows that there are unavoidable *learning overheads* in Agile-ant runs. The first one results from caching datasets in their second usage. For example, a recomputation of D_5 will be saved if it was cached in job_0 instead of job_1. The second one is due to delaying adding executors, which leads to evictions and blocks migration overheads. For example, if 12 executors are allocated in advance in LIR_2, the blocks of D_5 and D_{41} will be cached in all executors at the end of job_1 and job_5 respectively.

To quantify these overheads, we collect the caching decisions and the number of executors of Agile-ant runs (§10.2) and then execute each application using (1) Agile-ant with the number of executors selected right from the beginning (e.g. LIR on 12 executors), (2) Agile-ant with the caching decisions of Agile-ant (e.g. cache D_5 and D_{41} immediately upon their first usage and do not cache D_{35} in LIR), and (3) Agile-ant with both the caching decisions of Agile-ant and the number of executors selected right from the beginning. We compare their execution costs in Fig. 17.

Firstly, selecting the number of executors right from the beginning only saves 3.5% of the costs of Agile-ant runs because (i) A single executor is the optimal cluster size for some applications. (ii) The early evaluation proactively triggers adding executors before having cache limitations. (iii) Many applications execute in phases such that at the beginning, no dataset is cached and, thus, a single executor is the optimal choice for this phase. Secondly, caching suitable datasets in advance saves 13.3% of the costs of Agile-ant runs. In Class B applications, the application DAG is accessible right from the beginning. Agile-ant thus makes caching decisions without delays. However, caching the datasets in advance saves 49.2% and 25.8% of the costs of Agile-ant runs in DKM and LIR respectively because, as they are data-intensive applications, a single re-computation of a dataset adds a significant overhead (§10.1). Finally, caching suitable datasets and selecting the number of executors in advance save 14.3% of the costs of Agile-ant runs. Thus, using prediction frameworks reduces the cost of Agile-ant runs and when they have errors, Agile-ant corrects them.

10.8 Zoom In

To measure the execution cost reduction that each feature of Agile-ant brings, we conduct the following runs. In each run, we disable one feature and keep the remaining ones. (1) Agile-ant without auto-caching (*A-A w/o a-c runs*): disable the Auto-cacher and use the developers' caching decisions. (2) Agile-ant without scaling out (*A-A w/o s-o runs*): keep the cluster size selected right from the beginning (we run each experiment on 12 different cluster sizes and take the average), (3) Agile-ant without the early evaluation (*A-A w/o e-e runs*): disable the early evaluation and rely on the final evaluation only for scaling out. (4) Agile-ant without data migration (*A-A w/o d-m runs*): disable all data migration operations whether during the scale-out (§8.1) or after job execution (§7.2.1 and §7.2.2). As Fig. 18 shows, on average, disabling the Auto-cacher increases the cost by 2.97X, disabling the scaling out increases the cost by 2.86X, disabling the early evaluation increases the cost by 1.18X and disabling the data migration increases the cost by 1.4X. Note that disabling the Auto-cacher and relying on the developers' caching decisions benefits performance in some cases like LIR and SVM because the developers make correct caching decisions as the Auto-cacher, but the latter with a learning overhead (§10.7).

11 CONCLUSION

In this paper, we have presented Agile-ant, a self-managing framework that minimizes the execution costs of big data applications by making correct caching decisions and selecting the optimal cluster size on the fly. Overall, the evaluation of Agile-ant shows very good results, in comparison with the baseline and related work.

ACKNOWLEDGMENTS

This work was partially funded by the Thuringian Ministry for Economy, Science and Digital Society under the project "thurAI" and the German Research Foundation (DFG) in the context of the project "Processing-In-Memory Primitives for Data Management (PIMPMe)" (SA 782/31) as part of the priority program "Disruptive Memory Technologies" (SPP 2377).

REFERENCES

[1] [n.d.]. Advanced Analytics with Spark Source Code. https://github.com/sryza/aas. Accessed: 2024-07-30.

[2] [n.d.]. Apache Spark official website. https://spark.apache.org/. Accessed: 2024-07-30.

[3] [n.d.]. Dynamic resource allocation in Spark. https://spark.apache.org/docs/3.1.2/job-scheduling.html#dynamic-resource-allocation. Accessed: 2024-07-30.

[4] [n.d.]. MemoryStore in Spark. https://github.com/apache/spark. Accessed: 2024-07-30.

[5] [n.d.]. Netty network application framework. https://netty.io. Accessed: 2024-07-30.

[6] [n.d.]. The HiBench Suite. https://github.com/Intel-bigdata/HiBench. Accessed: 2024-07-30.

[7] 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 (2020), 819–839.

[8] Hani Al-Sayeh, Muhammad Attahir Jibril, Bunjamin Memishi, and Kai-Uwe Sattler. 2022. Blink: lightweight sample runs for cost optimization of big data applications. In *New Trends in Database and Information Systems: ADBIS 2022 Short Papers, Doctoral Consortium and Workshops: DOING, K-GALS, MADEISD, MegaData, SWODCH, Turin, Italy, September 5–8, 2022, Proceedings*. Springer, 144–154.

[9] Hani Al-Sayeh, Muhammad Attahir Jibril, Muhammad Waleed Bin Saeed, and Kai-Uwe Sattler. 2022. SparkCAD: caching anomalies detector for spark applications. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3694–3697.

[10] 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 *ACM SIGMOD*.

[11] 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)*.

[12] 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 *NSDI*, Vol. 2. 4–2.

[13] 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.

[14] 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.

[15] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[17] Jen-Yao Chwng, Donald Ferguson, George Wang, Christos Nikolaou, and Jim Teng. 1995. Goal-oriented dynamic buffer pool management for data base systems. In *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95*. IEEE, 191–198.

[18] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[19] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.

[20] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 560–595.

[21] 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.

[22] Hazar Harmouch and Felix Naumann. 2017. Cardinality estimation: An experimental survey. *Proceedings of the VLDB Endowment* 11, 4 (2017), 499–512.

[23] 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.

[24] Kaixin Huang, Shengan Zheng, Yanyan Shen, Yanmin Zhu, and Linpeng Huang. 2018. An adaptive eviction framework for anti-caching based in-memory databases. In *Database Systems for Advanced Applications: 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part II 23*. Springer, 247–263.

[25] 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).

[26] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics.. In *OSDI*. 427–444.

[27] 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.

[28] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.

[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[30] 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.

[31] Bing Liu, Fang Liu, Nong Xiao, and Zhiguang Chen. 2018. Accelerating spark shuffle with RDMA. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–7.

[32] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. 53–59.

[33] Ron L'Esteve. 2022. Databricks. In *The Azure Data Lakehouse Toolkit: Building and Scaling Data Lakehouses on Azure with Delta Lake, Apache Spark, Databricks, Synapse Analytics, and Snowflake*. Springer, 83–139.

[34] Ashraf Mahgoub, Alexander Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 189–204.

[35] 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.

[36] Patrick Martin, Hoi-Ying Li, Min Zheng, Keri Romanufa, and Wendy Powley. 2000. Dynamic reconfiguration algorithm: Dynamically tuning multiple buffer pools. In *Database and Expert Systems Applications: 11th International Conference, DEXA 2000 London, UK, September 4–8, 2000 Proceedings 11*. Springer, 92–101.

[37] 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.

[38] Michael J Mior and Kenneth Salem. 2020. ReSpark: Automatic Caching for Iterative Applications in Apache Spark. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 331–340.

[39] 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*.

[40] Mark Needham and Amy E Hodler. 2019. *Graph algorithms: practical examples in Apache Spark and Neo4j*. O'Reilly Media.

[41] 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.

[42] 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, 14 (2013), 1678–1689.

[43] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[44] Yijie Shen, Jin Xiong, and Dejun Jiang. 2021. Using vectorized execution to improve SQL query performance on spark. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–11.

[45] Wenhu Tian, Pat Martin, and Wendy Powley. 2003. Techniques for automatically sizing multiple buffer pools in DB2. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. 294–302.

[46] Jan Sipke Van Der Veen, Bram Van Der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J Meijer. 2015. Dynamically scaling apache storm for the analysis of streaming data. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*. IEEE, 154–161.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management*. 1–8.

[52] 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.

[53] 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.

[54] 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 *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[56] Kaihui Zhang, Yusuke Tanimura, Hidemoto Nakada, and Hirotaka Ogawa. 2017. Understanding and improving disk-based intermediate data caching in Spark. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2508–2517.

[57] 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.