Received April 29, 2021, accepted June 29, 2021, date of publication July 16, 2021, date of current version July 26, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3097823

Execution Repair for Spark Programs by Active Maintenance of Partition Dependency

XIUPEI MEI, IMRAN ASHRAF, XIAOXUE MA, HAO ZHANG, ZHENGYUAN WEI, HAIPENG WANG, (Graduate Student Member, IEEE), AND W. K. CHAN, (Member, IEEE)
Department of Computer Science, City University of Hong Kong, Hong Kong

Corresponding author: W. K. Chan (wkchan@cityu.edu.hk)

This work was supported in part by the General Research Fund (GRF) of Hong Kong Research Grants Council under Project 11214116 and Project 11200015; in part by the Innovation and Technology Fund of Hong Kong (HKSAR ITF) under Project ITS/378/18; in part by the CityU MF_EXT under Project 9678180; and in part by the Strategic Research Grant of City University of Hong Kong (CityU SRG) under Project 7004882, Project 7005216, and Project 7005122.

ABSTRACT Spark programs typically codify to reuse some of their generated datasets, called partition instances, to make their subsequent computations complete in a reasonable time. At runtime, however, the underlying Spark platform may independently delete such instances or accidentally cause these instances inaccessible to the program executions. Those instances will invalidate the computation assumption made in writing these programs that such depending instances are present, which leads performance bloat and even breaks the executions. In this paper, we present FAR, a novel and effective framework to handle such performance bloat and actively repair the executions by maintaining the instance dependencies in Spark program executions. FAR monitors the partition instance lifecycle activities at all levels, and determines from the execution plan of the current Spark action in the current program execution on whether a partition instance will have a dependency relation with a later one underlying the computation of that action. The experimental results showed that with the active execution repair mechanism of FAR, when some dependency partition instances were inaccessible, programs can achieve 7.3x to 67.0x speedup in re-generating them. The results also interestingly revealed that the program executions actively repaired by FAR can run to successful completion in environments with 1.7x-2.0x fewer available memory.

INDEX TERMS Debugging, execution repair, dataset dependency, big data.

I. INTRODUCTION

Programs running on a cluster of Spark nodes [2] are widely used in practice [36]. They accept inputs containing an arbitrary number of records to compute results. These programs, such as page rank [28] or hot topics [29], generate many sets of intermediate datasets. Each of such sets is called a data partition[2], where the partition instance contains the actual data records for processing. Each data partition is bound to an RDD (*Resilient Distributed Dataset*) [2], which is the most important data structure used in program code. Such a program manipulates RDD instances, thereby using the corresponding data partitions to systematically compute the results from its input through a sequence of Application Programming Interface (API) calls. Nonetheless, keeping all these intermediate partition instances is impractical.

The associate editor coordinating the review of this manuscript and approving it for publication was Hailong Sun.

To address the above problem, there are two levels of strategies: platform and program. At any time, a program execution σ holds a set π of partition instances.

At the platform level, a platform, hereafter denoted by $\text{Spark}_{\text{base}}$, may select and delete an existing instance of a data partition D, say D^1, from π to avail the memory occupied by D^1 for keeping a new partition instance for σ. If the deleted instance D^1 is latter required for generating other partition instances in the remaining part of σ, $\text{Spark}_{\text{base}}$ will check the dependency of partition D and apply an operation sequence based on π to generate a fresh instance of D^1 (denoted as D^2) as its fixing strategy. In generating such an instance D^2, $\text{Spark}_{\text{base}}$ should ensure all the data partition instances directly used to derive D^2 available first, but in some cases, has to delete some other partition instances to create space to keep the former instances. If such cycles of partition creation and deletion are not maintained well, there will be performance bloats, where the execution of a program will be significantly compromised by excessive partition creation and deletion of the same partition instances.

At the program level, with respect to D^1 and its belonging RDD instance R, application developers may add persistence instructions (e.g., *R.persist*() and *R.unpersist*()) in their program code C to retain or delete the corresponding partition instances of R in an all-or-nothing manner. Program-level strategies are unaware of operations done at the platform level. Thus, the internal operations of $\text{Spark}_{\text{base}}$ may invalidate any heuristics written in C that rests on the assumption that D^1 is always persistent for subsequent uses.

In this paper, we present FAR, a novel and effective framework to handle a class of performance bloat that equivalent partition instances of RDDs are excessively generated and deleted. To the best of our knowledge, FAR is the *first* systematic approach to address this problem.

FAR is built atop two insights on programs for Big Data processing. First, in a program execution σ, for each operation to produce a result (rather than an intermediate RDD), each partition D can precisely pair up with a set of outstanding uses (called *budget*) that *must* appear in current round of execution by σ. Second, if the current instance for D is deleted before fulfilling all such uses, $\text{Spark}_{\text{base}}$ will generate a new instance of D.

The basic idea of FAR is to compute the budget of each partition for each such operation in the evaluation phase of σ to extract the dependency relation between partitions relevant to the current operation. Moreover, during the concrete execution phase corresponding to the above evaluation phase, FAR does the following: On handling the request of an instance D^1 of partition D, FAR adjusts the budget of each partition that D depends on. When the budget of a partition is exhausted, FAR instructs $\text{Spark}_{\text{base}}$ to delete the instance of that partition. On contrary, if the budget of a partition D has not been exhausted but no instance of it is found, FAR increases the budget of each partition that D depends on. FAR also instructs $\text{Spark}_{\text{base}}$ to annotate D as reserve so that if a new instance of D is generated, that instance will be kept rather than deleted right after the current use.

FAR is designed with ease of use, high efficiency, and versatility in mind. We have implemented FAR as a Spark component. Application developers can simply insert pairs of FAR's API calls into each code region that leads to a performance bloat in their program code (as illustrated in Section IV.E) or enable it in the configuration file to make $\text{Spark}_{\text{base}}$ use FAR as the default.

We evaluate FAR on six representative applications taken from *GraphX* [6] and GitHub repositories [7] with real-world datasets as the evaluation benchmarks. We have compared FAR to the state-of-the-practice Spark platform (i.e., referred to as $\text{Spark}_{\text{base}}$ in this paper). The main results show that (1) in the scenarios of inaccessible dependency partition instances, FAR can achieve 7.3x to 67.0x speedup in re-generating them compared to $\text{Spark}_{\text{base}}$, and (2) in memory stringent scenarios, FAR can enable program executions to complete successfully in environments with 1.7x to 2.0x fewer available memory than $\text{Spark}_{\text{base}}$. The result also shows that for program executions that can run normally on $\text{Spark}_{\text{base}}$, FAR only incurred no more than 1.39% additional runtime slowdown overhead over $\text{Spark}_{\text{base}}$.

The main contribution of this work is threefold.

- This paper presents the first work, called FAR, to address a class of performance bloat that the equivalent partition instances are excessively produced.
- We show the feasibility of FAR by implementing it as a Spark component and demonstrate its ease of use.
- This paper presents an evaluation of FAR and shows that FAR is effective and efficient. The result also shows that FAR can enable programs to run to completion in situations unable to be handled by $\text{Spark}_{\text{base}}$.

The rest of this paper is organized as follows. We firstly revisit the preliminaries in Section II. Through a motivating example in Section III, we introduce the problem to be solved by FAR, which is presented in Section IV, followed by its evaluation and further discussion in Sections V and VI. We review the related work in Section VII and conclude this work in Section VIII.

II. PRELIMINARIES

A. THE SPARK FRAMEWORK

Spark [2] provides a runtime denoted as $\text{Spark}_{\text{base}}$. It provides ***dataset operations*** for applications to compute data records modeled as *Resilient Distributed Dataset* (***RDD***) [2], where each RDD instance represents ***a partitioned collection of data records***. Each data partition is an array for keeping data records kept in the primary storage and second storage, which we generally refer to as the memory in Spark.

Transformation (e.g., *map*, *filter, join,* and *groupByKey*) and ***action*** (e.g., *reduce*, *collect*, *count*, *first*, and *foreach*) are two kinds of dataset operations for Spark programs. When a transformation T is applied on an RDD instance R_X, a new RDD instance R_Y is created. Each partition in R_Y depends on one or multiple partitions of R_X, which is determined by the function type of T. During a program execution σ, all the created RDD instances and the transformation relations between them construct a *lineage graph G* [2].

Moreover, for each partition P_Y in R_Y, we model its ***dependency*** as a 3-tuple (P_Y, $< P_X >$, F_T), where the second element is a set of partitions in R_X that P_Y depends on and the third element is a transformation function of T. All the partitions and the dependency relations between them construct a ***partition dependency graph*** G_P.

Note that during the program execution, a partition P may be materialized (i.e., populated with data records) zero or multiple time(s) and each such operation produces a partition instance. For ease of reference, we refer to the partition instance of P with occurrence id i as P^i. Note that, for partition P, all its partition instances are *equivalent*.

A typical program execution consists of two alternating phases, the ***lineage graph construction phase*** and the ***concrete execution phase***.

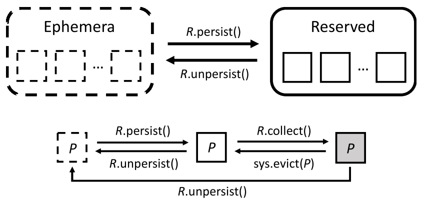


FIGURE 1. State transitions of an RDD instance *R* shown as rounded rectangles, with partitions shown as square rectangles. Partitions are shaded if their instances are kept in memory.

During the *lineage graph construction phase*, by executing transformation operations, σ creates a set of RDD instances and appends them to the lineage graph G_σ. This process continues until an action A on some RDD instance R is invoked, which starts a *concrete execution phase* to compute all the partitions of R. After the completion of A, σ starts the next round of lineage graph construction phase.

The dataflow between partitions is modeled as a dependency. For example, to generate P_Y^1 of P_Y, whose dependency is $(P_\mathrm{Y}, < P_\mathrm{X} >, F_T)$, Spark$_\mathrm{base}$ will ensure all the partition instances of its depending partitions in $< P_\mathrm{X} >$ are accessible followed by applying the function F_T on them to produce P_Y^1. If a partition, say P_X, in $< P_\mathrm{X} >$ is persistent (where the possible states of a partition are defined in the next subsection), Spark$_\mathrm{base}$ creates a dependency from the current instance, say P_X^1, of P_X to P_Y^1. On the other hand, if no partition instance of P_X is found, Spark$_\mathrm{base}$ creates a dependency to P_X to indicate that it should generate an instance of P_X. Therefore, to ensure a partition to have a partition instance, there is a dependency graph to end at either a partition instance or a partition. In the latter case, a partition instance, say P_X^2, is required to be generated so that the dependency now ends at P_X^2. We refer to the above dependency generation as the *establishment procedure* of dependency.

B. STATE TRANSITIONS OF RDD AND PARTITION

Fig. 1 depicts a state-transition diagram modeling the lifecycle of an RDD instance in Spark$_\mathrm{base}$. Spark$_\mathrm{base}$ provides two RDD operations, *persist* [1] and *unpersist* to handle the reuse of generated datasets. These operations, although happen during the lineage graph construction phase and transit the states of RDD instances in G_σ, are not recorded in the lineage graph being constructed by Spark$_\mathrm{base}$.

When an RDD instance R is declared, R is created with the ***ephemera*** state. When σ executes R.*persist*(), R's state transits to ***reserved***. A partition P of R is ***ephemera*** or ***reserved*** (depicted as square rectangles with dashed or solid border) if R is in *ephemera* or *reserved* state, respectively.

Consider the establishment procedure of dependency $(P_{\mathrm{Y}1}, < P_\mathrm{X} >, F_{T1})$. When a partition instance P_X^1 is generated, suppose that P_X is *ephemera*, P_X^1 will be only available within such procedure and is discarded after F_{T1} is completed. Alternatively, suppose that P_X is *reserved*, P_X^1 will be kept in the memory. We refer to the partition as ***persistent*** (depicted as shaded rectangles) if its instance is already kept in memory. As long as P_X is *persisted*, the subsequent creation of other dependency will reuse P_X^1 instead of generating new instances of P_X.

When R's state is *reserved* and σ executes R.*unpersist*(), R's state transits to *ephemera*, and all its *persisted* partition instances are deleted from the memory.

The operation *sys.evict*(P) models Spark$_\mathrm{base}$ to select a *persisted* partition P for reclaiming the memory space from P's occupation. When Spark$_\mathrm{base}$ invokes *sys.evict*(P), its instance P^i is deleted from the memory (and its memory occupation is de-allocated), and P is changed to a *reserved* partition. An intention behind the above design is to hide the deletion's impact on partition instance reuse in σ, which simplifies the handling of such ***missing partition instances*** at the program level. For instance, suppose that there is a *persisted* partition P and P^i is kept for multiple uses in σ. After *sys.evict*(P) is invoked, P^i is removed, and P transits to *reserved*. Upon the next creation of some dependency instance that uses P, P^{i+1} is populated with new data records, and P transits to *persistent* again. The newly generated P^{i+1} is kept for possible use in the subsequent part of σ.

III. MOTIVATING EXAMPLE

In this section, we present a motivating example.

The exemplified program implements the *Floyd-Warshall* algorithm to find the shortest paths in a weighted graph [9]. Let function $f_\mathrm{SP}(i, j, k)$ returns the shortest paths from vertex i to vertex j using the vertices from the set $\{1, 2, \ldots, k\}$. Thus, $f_\mathrm{SP}(i, j, 0)$ returns the weight of edge (i, j). For $k = 1, 2, \ldots, N$, $f_\mathrm{SP}(i, j, k)$ can be computed as follows:

$$f_\mathrm{SP}(i,j,k) = min(f_\mathrm{SP}(i,j,k-1), f_\mathrm{SP}(i,k,k-1) + f_\mathrm{SP}(k,j,k-1))$$

Hence, the shortest paths between all vertices can be obtained by iteratively invoking $f_\mathrm{SP}(i, j, k)$ of every vertex pair (i, j) for $k = 1, 2, \ldots, N$.

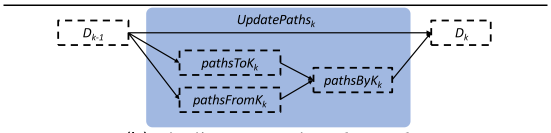
Fig. 2 and Fig. 3 show the implementation of the *Shortest Paths Program* in Spark. In Fig. 2(a), *updatePaths*() is a helper function. It accepts an RDD instance D which models distances of all the paths, and returns a new RDD instance D', which contains the distances that some of which are reduced by passing through vertex k.

The k-th invocation of *updatePaths*() constructs a lineage graph ***updatePaths$_k$***. Fig. 2(b) shows a simplified version of it. The variable D represents a source RDD instance (denoted as D_{k-1}) in *updatePaths$_k$*. At line 8, σ uses D_{k-1} to create another RDD instance *pathsToK* (denoted as *pathsToK$_k$*). The graph *updatePaths$_k$* contains an edge from D_{k-1} to *pathsToK$_k$*. This edge is labeled with *filter*(), which indicates how each partition instance of *pathsToK$_k$* can be computed based on its dependency partitions in D_{k-1}. For brevity, we do not show the label. Other nodes and edges are similarly created.

[1] In Spark, the *cache* operation is a synonym of *persist*. The difference between them is syntactic.

```
 1 class Path(from: Int, to: Int)
 2 class Dist(dist: Int)
 3
 4 val paths = load(...) //each path is in scheme ((from, to), dist)
 5 val graph = sc.makeRDD(paths) // create the graph RDD
 6
 7 function updatePaths(D, k)
 8 │ val pathsToK = D.filter { (p: Path, d: Dist) => p.to == k}
 9 │ val pathsFromK = D.filter { (p: Path, d: Dist) => p.from == k}
10 │ val pathsByK = connect (pathsToK, pathsFromK)
11 │ val Dk = D.join(pathsByK).map{
   │     (p: Path, (d1: Dist, d2: Dist)) => (p, min(d1.dist, d2.dist)) }
12 │ return Dk
13 end function
```

(a) *updatePaths* function



(b) The lineage graph *updatePaths*$_k$

FIGURE 2. Shortest paths program.

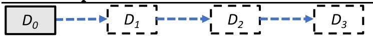
Fig. 3 shows four program versions of the *Floyd-Warshall* algorithm. Each version uses the graph edges as the initial paths, which is assigned to the variable D (at line 16, 23, 31, or 42). We refer such variable D as D_0 and assume each partition of D_0 is *persistent*. We also denote the RDD instance associating with variable D by D_k after the k-th invocation of *updatePaths* These four versions iteratively invoke *updatePaths*() N times. They are different only on when and which RDD instances to be set into which particular RDD states.

In this example, each partition instance is assumed to take $O(1)$ space and each transformation function is assumed to take $O(1)$ time. In this way, we can focus our discussion on the number of dependency instances established when comparing the effects of RDD state operations codified in these versions. Since the actual calculation is at the data partition level, in the sequel, again for simplicity, we suppose that each RDD instance contains one partition. We describe the example at the data partition level and use D_k for $k = 0$ to N to indicate the RDD instance or partition interchangeably.

Version 1 is a straightforward implementation. A high-level lineage graph created at line 18 through the loop at lines 17–19 is depicted under the code in Fig. 3, where all the nodes and edges produced in *updatePaths*() are hidden. In the first iteration, at line 18, *updatePaths*() uses D_0 to compute D_1. Since *version1* has no RDD state operations, the state of D_1 remains as *ephemera*. The lineage graph for the program execution is extended with the graph *UpdatePaths*$_1$ (see Fig. 2), which is depicted as a blue dashed arrow from D_0 to D_1 in Fig. 3. Similarly, D_2 to D_N are returned by *updatePaths*() from the remaining iterations (up to the third iteration).

Since D_1 to D_N are *ephemera*, their instances will not be shared among the creations of dependency instances. As such, *version1* presents a typical problem that suffers from a severe performance bloat problem. For instance, when the *collect* action is performed (line 20) on D_3, *version1* starts to establish the dependency of D_3, which generated three instances

```
15 procedure version1()
16 │ var D = graph
17 │ for each k from 1 to N do:
18 │ │ D = updatePaths(D, k)
19 │ end for
20 │ val result = D.collect()
21 end procedure
```

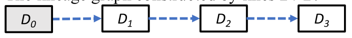


Time and space complexities for calculating D_N: $O(3^N)$ and $O(1)$

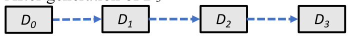
Problem of version1: high runtime slowdown overhead

```
22 procedure version2()
23 │ var D = graph
24 │ for each k from 1 to N do:
25 │ │ D = updatePaths(D, k)
26 │ │ D.persist()
27 │ end for
28 │ val result = D.collect()
29 end procedure
```

The lineage graph constructed by lines 24-27



After generation of D3

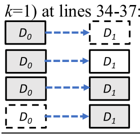


Time and space complexities for calculating D_N: $O(N)$ and $O(N)$

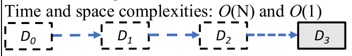
Problem of version2: high memory overhead

```
30 procedure version3()
31 │ var D = graph
32 │ for each k from 1 to N do:
33 │ │ val prevD = D
34 │ │ D = updatePaths(D, k)
35 │ │ D.persist()
36 │ │ D.collect()
37 │ │ prevD.unpersist()
38 │ end for
39 │ val result = D.collect()
40 end procedure
```

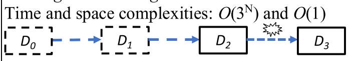
The states of RDDs evolve in the first iteration (when k=1) at lines 34-37:



No instance re-generation:
Time and space complexities: $O(N)$ and $O(1)$



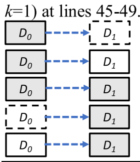
Needing instance re-generation:
Time and space complexities: $O(3^N)$ and $O(1)$



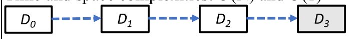
Problem of version3: incurs high runtime slowdowns if some previous partition instance has been lost before collect() runs

```
41 procedure version4()
42 │ var D = graph
43 │ for each k from 1 to N do:
44 │ │ val prevD = D
45 │ │ D = updatePaths(D, k)
46 │ │ D.persist()
47 │ │ D.collect()
48 │ │ prevD.unpersist()
49 │ │ prevD.persist()
50 │ end for
51 │ val result = D.collect()
52 end procedure
```

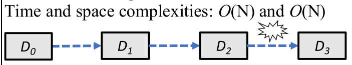
The states of RDDs evolve in the first iteration (when k=1) at lines 45-49.



No instance re-generation:
Time and space complexities: $O(N)$ and $O(1)$



Need instance re-generation:
Time and space complexities: $O(N)$ and $O(N)$



Problem of version4: A problem similar what version2 encountered occurs

FIGURE 3. Four versions of the finding shortest paths program.

of D_2: D_2^1 to compute *pathsToK*$_3$ at line 8, D_2^2 to compute *pathsFromK*$_3$ at line 9, and D_2^3 to compute D_3 at line 11. Similarly, each instance of D_2 also triggers three instances of D_1. Hence, the time complexity of *version1* is $O(3^N)$. Since none of these instances is persisted, the space complexity of persist partitions in *version1* is $O(1)$.

Version 2 marks each D_k for $k = 1$ to N to be reserved (line 26). Therefore, after D^1_{k-1} has been generated, it can be shared among the generations of D^1_k and the nodes in *UpdatePaths*$_\mathrm{k}$. Hence, the time complexity of *version2* is $O(\mathrm{N})$. As D^1_1 to D^1_{N-1} are not deleted by *version2* after their uses, the space complexity of *version2* is $O(\mathrm{N})$. In general, this is an impractical solution due to limited memory capacity in a cluster node. Thus, through the platform-level strategy, Spark$_\mathrm{base}$ will sooner or later delete some partition instances, irrespective to whether they will later be used in generating D^1_N. *version2* thus suffers from another instance of the performance bloat problem in program executions.

Version 3 iteratively deletes D^1_{k-1} from the immediate past iteration after D^1_k has been generated (lines 36–37). We note that this coding style is advocated as a best practice to alleviate performance issues [6]. Consider the loop at lines 32–38. In the k-th iteration, at line 33, D_{k-1} from the previous loop is in the *persistent* state and assigned to variable *prevD*; at line 34 and 35, D_k is returned by *updatePaths*() and is set to *reserved*; at line 36, the *collect*() action generates D^1_k; and at line 37, *version3* deletes D^1_{k-1} and changes D_{k-1} to *ephemera*. The time and space complexities for *version3* are $O(\mathrm{N})$ and $O(1)$, respectively. The corresponding high-level lineage graph for the execution point at the end of the third iteration is depicted under the title of "No instance re-generation".

Nonetheless, the above coding style becomes ineffective when partitions are no longer persistent according to the plan of *version3*. Consider the N-th iteration of loop 32–38 to generate D^1_N where D^1_{N-1} has been generated in the (N-1)-th iteration. Suppose that D^1_{N-1} is deleted by Spark$_\mathrm{base}$ to fulfill the data population requests from the current task or other tasks concurrently running on the same cluster node before generating D^1_N. The state of $D_{\mathrm{N}-1}$ will be changed from *persistent* to *reserved*. In this case, when D^1_N is about to generate, a new partition instance D^2_{N-1} has to be re-generated first. Nonetheless, in the (N-1)-th iteration, after D^1_{N-1} has been populated, D^1_{N-2} as well as all previous instances have been deleted by *version3*. Thus, to re-generate D^2_{N-1}, Spark$_\mathrm{base}$ re-computes every such depending partition based on the current partition dependency graph (start from reloading the input file to generate D^2_0) in an ad hoc manner. Since each such partition has been explicitly deleted by *version3* via the use of *unpersist*() operations and their states are thus *ephemera* (see Fig. 1). Similar to *version1*, the time complexity of *version3* in this scenario becomes $O(3^\mathrm{N})$. The corresponding high-level lineage graph is depicted under the title of "Needing instance re-generation".

Version 4 is revised from *version3* by adding a *persist*() call at line 49 right after the *unpersist*() call at line 48. This can be viewed as a possible patch added by a developer to fix *version3* after realizing the instance re-generation problem illustrated by *version3*. In this case, the *persist*() and *unpersist*() calls issued in the iterations for $k = 1$ to N delete the partition instances of D_0 to $D_{\mathrm{N}-1}$ but transit their states into *reserved*.

When there is no missing partition instance. *version4* creates the same amount of dependency instances as *version3*. The time and space complexities in this scenario are $O(\mathrm{N})$ and $O(1)$, respectively. Consider the scenario of missing partition instances encountered by *version3*. The re-generated instances of D^2_{N-1} to D^2_0 are shared among the dependency establishments as D_0 to $D_{\mathrm{N}-1}$ are marked as *reserved* at line 49. However, although multiple instance re-generations for the same partition can be avoided, each such instance is kept persistently once generated. The space complexity becomes $O(\mathrm{N})$. Similar to *version2*, this is an impractical solution. Spark$_\mathrm{base}$ will eventually delete some partition instances, irrespective to the remaining needs and logics of *version4*. It suffers from the performance bloat problem as well.

IV. OUR PROPOSAL: FAR

In this section, we present FAR. FAR is built on top of Spark$_\mathrm{base}$ and realized as a Spark$_\mathrm{base}$ component.

A. OVERVIEW

During the program execution, the partition states are dynamically changed by the program, Spark$_\mathrm{base}$, and the system environment. Program executes the coded management instructions on RDD instances. When the system is going to be out of memory, Spark$_\mathrm{base}$ selects and deletes some partition instances from the memory to reallocate the memory space [8]. This technique is particularly appealing to enable program executions to run to completion when the memory is stringent. Nonetheless, it is inadequate. As we have presented via a series of programs in the motivating example (Section III), the partition states will directly affect the creation of dependencies. During the creation procedure, failure to establish the dependency relations with other partition instances lead to severe performance bloat problem. A large amount of time and space resources are wasted during such procedure.

FAR is proposed to guard the creation of dependency instances during the execution by creating a period for them to share the equivalent partition instance. Among all possible execution points in σ, FAR strategically chooses the execution point of each action invocation to identify partitions with outstanding uses in each concrete execution phase with respect to all the partitions belonging to the target RDD instance of the dataset action (e.g., the RDD instance R in *R.collect*()). It automatically tracks (i.e., reserves and deletes) these identified partitions not only during the concrete execution phase but also whenever there is any ad hoc platform deletion of any partition, which the latter further identifies partitions for additional outstanding uses with respect to the partition needed to be deleted. A new round of partition generation may trigger a further round of partition deletion, and vice versa. FAR strives for a balance between (lazy) partition retention and (lazy) partition deletion to advance the state of the art to make Spark$_{\text{base}}$ able to serve a wider spectrum of scenarios.

FAR provides two API functions, *enableFAR*() and *disableFAR*(), for application developers to enable and disable FAR in their application code, respectively. It also provides a configuration option in Spark$_{\text{base}}$ so that application developers need not to add FAR API calls to their code.

FAR maintains an analysis state Σ_σ for σ, where Σ_σ is a triple $\langle target, P, R\rangle$, where *target* is the RDD instance associating with the invoked dataset action in the current concrete execution phase, P is the set of all *persistent* partitions kept in Spark$_{\text{base}}$, and R is a map which stores the budget on the outstanding uses (or budget for short) of each partition involved directly or indirectly in computing *target*. FAR consists of two core algorithms, which will be presented in the next two subsections.

B. ALGORITHM 1: PARTITION IDENTIFICATION

On invoking a dataset action A (e.g., *inst*.collect()), ONACTIONINVOKED(*inst*) in Algorithm 1 is called, where *inst* is the RDD instance associating with A. For ease of reference, we denote the set of partitions of *inst* by *inst.partitions* and the set of all the direct dependency partitions of a partition p by *p.dependencies*.

The procedure ONACTIONINVOKED() first keeps *inst* to *target* (line 3). It retrieves from Spark$_{\text{base}}$ the set of all *persistent* partition instances currently maintained by Spark$_{\text{base}}$ and keeps their references in set P via function *getPersistentState*() (line 4). It also recursively computes the number of outstanding uses on each dependency partition starting from *inst.partitions* by calling GETOUTSTANDINGUSAGE() (line 5), which in essence traverses the partition dependency graph, and keeps them in map R. At line 6, it invokes CHECK(), which checks the budges of each such partition, and reserves the partition (line 29) if the partition is *ephemera* and its budget is larger than one (lines 6-7), where *persistPartition*(*part*) marks the *part* as reserved and sets the flag *part.FAR* to true.

The procedure GETOUTSTANDINGUSAGE() accepts a set of partitions that each needs to be generated and returns the additional budget of each dependent partition. It first creates a map C to store the number of times each partition visited and a queue Q to keep the partitions to be visited (lines 13–14). Q is initialized with the partitions in *parts*. Then, the procedure visits each partition d in Q. In each iteration, it checks whether d is currently persisted in Spark$_{\text{base}}$ and whether d has been visited before. If neither is the case, this indicates that d will be generated in the current concrete execution phase and its dependency partitions should be computed first. So, the procedure adds all the dependent partitions of d to Q for traversal (lines 17–19). Then, the procedure updates the map C to keep a budget on d: If d is currently in C, its budget (denoted as C[d]) is incremented by 1; otherwise, C[d] is set to 1 (line 20). As the partitions in *parts* are also budgeted during these node visiting rounds, these initial visits

Algorithm 1: Partition Identification

```
FAR Analysis State: $\Sigma_\sigma = \langle target, P, R\rangle$
target: RDD - RDD instance of current execution phase
P: Set<Partition> - set of persistent partitions
R: Map<Partition, Int> - outstanding usage of each
  partition

1  procedure ONACTIONINVOKED (inst)
2  Input : the RDD instance that the action is invoked on
3      target ← inst
4      P ← getPersistentState()
5      R ← GETOUTSTANDINGUSAGE(inst.partitions)
6      for each part in R,
7          do: CHECK(part) end for
8  end procedure
9
10 procedure GETOUTSTANDINGUSAGE(parts)
11 Input: a list of partitions to generate
12 Output :
     expected usage of other partitions in generating parts
13     C ← Map ()
14     Q ← Queue(parts)
15     while $Q \neq \emptyset$:
16         d ← Q.pop()
17         if d is not in P and d is not in C then:
18             Q.append(d.dependencies)
19         end if
20         C[d] ← C[d] + 1 if d in C else 1
21     end while
22     for each d in parts, do: C[d] ← C[d] − 1 end for
23     return C
24 end procedure
25
26 procedure CHECK(part)
27 Input : a partition to check
28     if R[part] > 1 and part is ephemera then:
29         persistPartition(part)
30     end if
31 end procedure
```

are deducted from C to avoid duplicated budgeting (line 22). The resultant C is returned.

C. ALGORITHM 2: PARTITION TRACKING

FAR monitors the state changes of partitions not only in concrete execution phases but also whenever there is any need of ad hoc deletion of partition instances. More specifically, whenever an instance of a reserved partition has been generated, ONPARTITIONPERSISTED() is invoked (lines 1–12), and whenever a persistent partition instance is deleted, ONPARTITIONEVICTED() is invoked (lines 14–24).

ONPARTITIONPERSISTED: During the generation of *part*, when a reserved partition *part* has been populated with data by Spark$_{\text{base}}$, FAR updates the budget on its persistent dependency partitions and deletes their instances if their budgets have been exhausted: FAR first gets the actual usage information by calling TRAVERSE(*part*) (line 3), which conducts a reachability analysis starting from *part* and returns the number of actual uses of each reachable *persistent* partition instance in P. More specifically, in the procedure TRAVERSE(), a map U and a queue Q are created to store the visit counts of each visited partition and partitions pending to be visited (initialized with *part*'s dependency partitions) (lines 35–36). Then, the procedure iteratively takes a partition from Q to visit until Q is empty (lines 37–43). During each visit on partition d, it increases the actual use of d in U (denoted as $U[d]$) by 1 (line 39). Then, it checks whether d has been persisted. If this is not the case, d's dependency partitions are appended to Q for further visit (lines 40–42). Finally, U is returned to the caller.

After the actual use count on each persistent partition is computed and kept in U, FAR updates the budget on each partition in R (lines 4–10). More specifically, for each partition d in U, the budget $R[d]$ is deducted by the actual use count $U[d]$ (line 5). If $R[d]$ becomes 0, it indicates the budget on d has been exhausted. If this is the case, the physical storage of d is removed from memory via *removePartition*(d)[2] (line 7), where the actual deletion is left by Spark$_{\text{base}}$, and d is also removed from P (line 8). A special consideration is that to avoid interference with the application logic, before the marking for deletion on d, FAR also checks whether d is reserved by itself beforehand. Otherwise, FAR will not mark it for partition deletion. Finally, *part* is included in P (line 11).

ONPARTITIONEVICTED: The procedure firstly removes *part* from P (line 16), and updates the budgets on the outstanding uses of all other partitions if these partitions are directly or indirectly dependency partitions of *part* (lines 17–24). More specifically, FAR checks whether *part*'s budget (denoted as $R[part]$) is positive, which indicates whether *part* is referenced in the later part of the concrete execution phase. If this is the case, FAR computes the budgets of the outstanding uses on other partitions that *part* depends on by calling the procedure GETOUTSTANDINGUSAGE (We note that at line 18, as GETOUTSTANDINGUSAGE() accepts a list of partitions as its input, *part* is packed into a list). After the latest budget on each such partition, say d, is computed and kept in C, FAR increases the budget on d at line 20, and reserves d if d is currently *ephemera* and will be used later via CHECK() (line 21).

FAR also monitors failure exceptions during each concrete execution phase. Once a data loss event is triggered by Spark$_{\text{base}}$, FAR conducts a state refresh by calling the ONFAILUREOCCURS() procedure before the default recovery mechanism of Spark$_{\text{base}}$ is triggered. The purpose is that the data loss invalidates FAR's internal state for further analysis. Hence, in ONFAILUREOCCURS(), P is firstly re-computed (line 27), and R is re-calculated by calling GETOUTSTANDINGUSAGE() with all partitions of the *target* instance as input (line 28). Note that although all the partitions are passed for analysis, the procedure only returns the analysis results on those partitions that have not yet been evaluated, as the partitions evaluated prior to the failure point are skipped. Finally, each ephemera partition in R is marked as reserved if its budget is still greater than one.

[2]Please refer to Section V.A for the details of *removePartition*

D. DISCUSSION ON DESIGN AND NOVELTY OF FAR

FAR has three aspects of novelty in its design.

First, FAR identifies and adjusts the budgets of relevant partitions at hybrid levels: collectively at the lineage graph level when each concrete execution phase starts and individually at the partition level when a persistent partition instance is deleted by Spark$_{\text{base}}$.

Consider a pure partition level strategy, which identifies such a partition when the partition is persistent in memory or marked as reserved. As illustrated in the motivating example, a partition not marked as reserved in the application will not be set into the reserved or persistent state. So, this strategy could not meaningfully start the tracking process at all. Consider another pure partition level strategy, which identifies a partition to be persistent when the partition is ephemera. Since most partitions in a typical program execution should be ephemera (for instance, see *version3* in the motivating example, which reduces the space complexity from $O(\text{N})$ to $O(1)$), this alternative strategy will invoke numerous rounds of traversal on almost every partition instance. Moreover, if a persistent partition has been deleted by Spark$_{\text{base}}$, the state of that partition will **not** be *ephemera* (see Fig. 1). Thus, this alternative strategy cannot correctly handle system deletion scenarios. It should also be noted that Spark$_{\text{base}}$ uses a strategy at the pure partition level, which is shown to be inadequate in Section V.

Consider a pure collective level strategy. A system deletion on a partition of an RDD instance will lead each partition of the RDD instance to trigger a round of graph traversal, which produces redundant computation demands (note that ONPARTITIONEVICTED() in Algorithm 2 uses a finer-granularity, which suffices to serve that purpose). Besides, to the best of our knowledge, FAR is the first technique to propose handling these partitions. Since this strategy is built on top of FAR, without FAR to lay down the groundwork, it would be more difficult to be discovered.

Second, FAR chooses to (i) keep more persistent partition instances (than Spark$_{\text{base}}$) even when free memory locations are in shortage and (ii) applies a partition deletion strategy through Spark$_{\text{base}}$ (e.g., via ONPARTITIONEVICTED() of Algorithm2). As a comparison, Spark$_{\text{base}}$ only chooses to delete partitions. We can view that our strategy forms a kind of two-player game (where FAR and its underlying Spark$_{\text{base}}$ to represent two competing players) to find an equilibrium feasible to both players (if possible), which is novel.

Last, but not the least, FAR simplifies the application code by raising the level of code abstraction on handling the persistence aspect of RDD instances from a procedural programming approach to an annotation approach (e.g., via *enableFAR*() and *disableFAR*() in the application code). It will

Algorithm 2: Partition Tracking

```
FAR Analysis State: $\Sigma_\sigma = \langle target, P, R \rangle$
target: RDD - RDD instance of current execution phase
P: Set<Partition> - set of persistent partitions
R: Map<Partition, Int> - outstanding usage of each
   partition

1  procedure ONPARTITIONPERSISTED($part$)
2  Input : the partition persisted in memory
3      $U \leftarrow$ TRAVERSE($part$)
4      for each $d$ in $U$ do:
5          $R[d] \leftarrow R[d] - U[d]$
6          if $R[d] = 0$ and $d.FAR$ then:
7              removePartition($d$) // delete the partition
8              $P \leftarrow P \setminus \{d\}$
9          end if
10     end for
11     $P \leftarrow P \cup \{part\}$
12 end procedure
13
14 procedure ONPARTITIONEVICTED($part$)
15 Input : a partition deleted by Spark$_{base}$
16     $P \leftarrow P \setminus \{part\}$
17     if $R[part] > 0$ then:
18         $C \leftarrow$ GETOUTSTANDINGUSAGE([$part$])
19         for each $d$ in $C$ do:
20             $R[d] \leftarrow R[d] + C[d]$
21             CHECK($d$) // mark for retention
22         end for
23     end if
24 end procedure
25
26 procedure ONFAILUREOCCURS()
27     $P \leftarrow$ getPersistentState ()
28     $R \leftarrow$ GETOUTSTANDINGUSAGE($target.partitions$)
29     for each $part$ in $R$, do: CHECK($part$) end for
30 end procedure
31
32 procedure TRAVERSE($part$):
33 Input: a partition
34 Output : actual uses of partitions in generating part
35     $U \leftarrow$ Map ()
36     $Q \leftarrow$ Queue($part.dependencies$)
37     while $Q \neq \emptyset$:
38         $d \leftarrow Q$.pop()
39         $U[d] \leftarrow U[d] + 1$ if $d$ in $U$ else 1
40         if $d$ not in $P$ then:
41             $Q$.append($d.dependencies$)
42         end if
43     end while
44     return $U$
45 end procedure
```

be exemplified in Section E and will be used in our evaluation on FAR in Section V. To the best of our knowledge, FAR is the first framework to provide such supports to application developers.

```
1   // revised version 1 with FAR enabled
2   procedure version5()
3       sc.enableFAR()
4       var D = graph
5       for each k from 1 to N do:
6           D = updatePaths(D, k)
7       end for
8       val result = D.collect()
9       sc.disableFAR()
10  end procedure
```

FIGURE 4. Enable FAR for *version1*.

E. EXAMPLES WITH FAR

This section illustrates how FAR works with the motivating example in Section III.

On top of *version1*, application developers can simply revise the code by inserting a pair of *enableFAR*() and *disableFAR*() API calls as illustrated in Fig. 4. The revised program (*version5*) uses FAR to guard the dependency creations during the execution between line 3 and line 9.

Like Section III, suppose that D_0 is *persistent*. When the *collect*() action at line 8 is invoked on D_k, the handler procedure ONACTIONINVOKED() of Algorithm 1 is invoked with D_k to set up the analysis state of FAR: *target* is assigned with the RDD instance D_N, the persistent partition set P is $\{D_0\}$, and the expected uses in R is $\bigcup_{k=1 \text{ to } N} \{D_{k-1} = 3, pathToK_k = 1, pathFromK_k = 1, pathByK_k = 1\}$. FAR checks each partition in R, and invokes *persistPartition*() on D_i for $i = 1$ to N–1 (as D_0 is persisted already) to change these partitions into *reserved* state. Finally, it returns the control back to the concrete execution phase of Spark$_{base}$.

The budget on D_1's outstanding uses in R is 3, which means that D_1 will be used thrice during the computation and can be removed after its third usage.

In the concrete execution phase, Spark$_{base}$ first starts the computation from D_0. Once D_1^1 is generated, the procedure ONPARTITIONPERSISTED() of Algorithm 2 is invoked with D_1. Based on the persistent partitions in P, FAR computes the uses of each partition in U, which is $\{D_0 = 3, pathToK_1 = 1, pathFromK_1 = 1, pathByK_1 = 1\}$ (line 3 of Algorithm 2). After deducting the uses from R, all the budgets in U become 0. As $pathToK_1$, $pathFromK_1$ and $pathByK_1$ are *ephemera*, and D_0 is not reserved by FAR, no partition instance is removed (lines 4 –10 in Algorithm 2). Finally, D_1 is added to P, and P becomes $\{D_0, D_1\}$.

After D_1 becomes persistent, D_2^1 is then generated and the procedure ONPARTITIONPERSISTED () is invoked again with D_2. The computed partition usage U is $\{D_1 = 3, pathToK_2 = 1, pathFromK_2 = 1, pathByK_2 = 1\}$. After deducting the uses from R, D_1's budget becomes zero. As D_1 is reserved by FAR, FAR deletes D_1^1 and changes D_1 to *ephemera*. Finally, P becomes $\{D_0, D_2\}$. The computations on remaining partitions D_3 to D_N follow the same process.

Consider an alternative scenario where D_2^1 is deleted by Spark$_{base}$ before D_3^1 is created. ONPARTITIONEVICTED() is invoked with D_2 as input. It firstly removes D_2 from P. The budget on D_2's outstanding uses is 3, which indicates that D_2 will be re-generated. FAR computes the new budget on D_2 based on the current state kept in P (line 18 in Algorithm 2), and the result is $C = \{D_0 = 3, D_1 = 3, pathToK_1 = 1, pathFromK_1 = 1, pathByK_1 = 1, pathToK_2 = 1, pathFromK_2 = 1, pathByK_2 = 1\}$. These budgets are added to R, and the state of D_1 is changed to *reserved*. In the rest of the execution, D_1 is persisted during D_2^2's re-generation. ONPARTITIONPERSISTED() is invoked with D_2 when D_2^2 is generated. The outstanding uses of other partitions are computed and kept in U, where $U = \{D_1 = 3, pathToK_2 = 1, pathFromK_2 = 1, pathByK_2 = 1\}$. As all D_1's budget has been exhausted, D_1^2 is deleted by FAR.

Therefore, by enabling FAR on top of *version1* (to become *version5*), its time and space complexity are O(N) and $O(1)$, respectively, regardless of whether or not partitions need to be re-populated, which cannot be achieved by Spark$_{\text{base}}$ alone in the four versions shown in Fig. 3, or any combination of them.

V. EVALUATION

In this section, we present the evaluation on FAR by comparing it to the original Spark platform with default configuration (denoted as Spark$_{\text{base}}$) [8]. We aim to answer the following two key questions:

- **RQ1**: Can FAR effectively address the performance bloat problem faced by Spark applications in situations incurring the performance bloat problem?
- **RQ2**: Is FAR efficient compared to Spark$_{\text{base}}$ in the scenario of normal program executions?

For RQ1, we consider two sub-scenarios which to the best of our knowledge, are representative. Recall that a program execution will start its concrete execution phase at each encountered dataset action performed on an RDD instance. During the creation of the related dependency instances, there may be computation errors or system errors to prevent the concrete execution phase to run smoothly. We refer to it as the ***first sub-scenario*** (**Scenario 1**) by failing a program execution when the concrete execution phase is about to start as what Zaharia *et al.* did in their experiment [2].

Alternatively, the concrete execution phase can proceed, but the underlying cluster nodes may have insufficient free memory locations. In such cases, some partitions of a target program execution will change their states from *persistent* to *reserved* due to platform deletion of their instances. We refer to it as the ***second sub-scenario*** (**Scenario 2**) by systematically varying the amount of available memory of executor nodes.

We also note that during program development, a program version may contain functional bugs (e.g., unable to process a particular record with unexpected contents or in an unexpected format), resulting in program execution failure when processing a test dataset. In typical Spark configurations, when a failure occurs, the program version may intentionally re-run for a few times (e.g., 4 times) before execution abortion. When such a failure occurs in the course of establishing a dependency, Spark$_{\text{base}}$ re-generates that instance. Thus, the two sub-scenarios also help evaluate to what extent FAR can assist application developers in testing and debugging their programs incurring critical failures by reducing the runtime of their executions involved.

It is also quite common that a particular program execution may not encounter any performance bloat problem due to dependency establishment failure, hence not triggering the re-generation process of Spark$_{\text{base}}$ of corresponding partition instances. Intuitively, the memory consumption required by programs highly depend on the scale of the input that to be processed. Therefore, although the capacity of underlying infrastructure can be large, the overall resources and the costs could become major issues when computing over larger datasets and/or demanding more accurate results. For instance, when running on a cloud computing platform, multiple big data applications may be parallelly executed on the same cluster. The available resources become more elastic and the platforms perform more interventions than private clouds. Therefore, it is interesting to see how FAR performs under stringent memory resource scenarios.

For RQ2, we consider the normal situation (**Scenario 3**) that neither a system failure nor a memory resource shortage occurs. Intuitively, in such situations, FAR will incur an additional overhead in performing its state initialization and more overheads during concrete execution phases. In RQ2, we aim to evaluate whether FAR is efficient, i.e., whether the overhead introduced by FAR is acceptable. To assess the performance, we evaluate the runtime slowdown of FAR under this scenario and compare it with Spark$_{\text{base}}$.

The whole evaluation procedure taken around 120 hours. All the evaluation results and the source code base of FAR are available at *https://github.com/FAR-Data/.*

A. IMPLEMENTATION

We have prototyped FAR as a module in Spark framework using the Scala language. When an execution started, the Spark driver was launched on the master node and a *FARManager* instance was created in it.

The manager got the schedule information (including jobs, stages and tasks) from Spark's *DAGScheduler* instance and checked whether each partition was persistent by querying the *BlockManagerMaster* instance. To do that, we modified *DAGScheduler* and *Block-ManagerMaster* and so that *FARManager* was notified whenever any concrete execution phase was about to start, any partition was generated or removed. With such information, *FARManager* kept the outstanding uses of each partition and performed its algorithms accordingly.

Whenever *FARManager* needed to persist a partition, it updated the state of such partition on the driver and synchronized the changes to corresponding executors via *DAGScheduler*. When *FARManager* needed to delete a partition (*removePartition*() in Algorithm 2), it requested *BlockManagerMaster* to send *removeBlock* messages to executors, which removed the corresponding dataset.

TABLE 1. Application benchmarks.

Benchmarks (Abbr)	Description
ShortestPaths (SP)	Compute shortest paths to the given set of landmark vertices. [9]
WeaklyConnectedComponents (WCC)	Compute the connected component membership of each vertex and return a graph with the vertex value containing the lowest vertex id in the weakly connected component containing that vertex. [6]
PageRank (PR)	Run PageRank for a fixed number of iterations returning a graph with vertex attributes containing the PageRank and edge attributes the normalized edge weight.[6][28]
SVDPlusPlus (SVD++)	Implement SVD++ based on "*Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model*".[41]
GreedyMaximalMatching (GMM)	An implementation of the greedy matching algorithm. [7]
Breadth-First-Search (BFS)	An implementation of the Breadth-First-Search algorithm. [7]

B. EXPERIMENTAL SETUP

1) ENVIRONMENT

The experiments were conducted on a server with 4 Intel Xeon E7-4850 v3 CPUs and 512GB memory, and with VMware ESXi [10], a bare-metal hypervisor running on it. To approach the real-world environment, we set up a cluster of 21 VM nodes on the server to run the experiments. Each node was a virtual machine instance configured with four vCPUs and 16GB memory, which is a commonly used configuration provided by cloud computing services (such as Amazon AWS[3] and Microsoft Azure[4]) for general purpose computing. This cluster configuration with 21 VM nodes was the maximum number of VMs that we can create in our server while keeping the server's smooth running. Among these nodes, one served as the master node running the Spark driver process, and the other twenty nodes served as worker nodes of Spark executor. Each executor process was configured with 4 cores and 14GB memory.

The JVM version was 1.8.0_231-b11 and the Scala version was 2.11.8. HDFS shipped with Hadoop 2.7.3 was used to keep data. We compiled Spark source code with version 2.4.4 on the above setting as Spark$_{\text{base}}$. We then added the implementation of FAR to this code base as a Spark component and compiled it as FAR in the evaluation.

2) BENCHMARKS

We used six benchmarks in the experiments. All these benchmarks are summarized in Table 1.

The Spark GraphX library is widely used in practice and many of its programs were used in various previous work [25], [26]. We used all the non-trivial benchmarks as our benchmarks: *WeaklyConnectedComponents* (WCC), *PageRank* (PR), *SVDPlusPlus* (SVD++), and *ShortestPath* (SP). We also used implementations of two commonly used algorithms, *GreedyMaximalMatching* (GMM) and *Breadth-FirstSearch* (BFS) from GitHub [7].

The above six benchmarks have also been widely used in other works. In [6], the authors used *PageRank* and *ConnectedComponents* (an implementation to compute weakly connected components in a graph) to evaluate the performance of GraphX, Giraph and GraphLab [38], [39]. In [40], four widely used applications, including *BreadthFristSearch*, *PageRank*, *SingleSourceShortestPath* (which is a special case of *ShortestPaths* in our motivating example) and *WeaklyConnectedComponent* are used to evaluate the efficiency of their work. We included all these benchmarks in our evaluation on FAR. Besides, we added two more benchmarks to seek for a wider generalization. All the benchmarks are using *persist* (or *cache*) and *unpersist* operations to keep and discard the intermediate datasets. In the experiment, we enabled FAR in configuration file so that all the concrete execution phases are protected to make fair comparisons.

3) DATASETS

We used two real-world datasets to evaluate the performance of FAR on these benchmarks. For SP, WCC, PR, GMM and BFS, we used the *uk-2005-host* graph dataset from *WebGraph* [11]. This dataset contained 39,459,925 nodes and 936,364,282 edges, and the graph was stored in a single file with a size of 15.32GB. For SVD++, as it is a widely used algorithm to build a recommendation system, we chose the *Netflix Prize* dataset [12] to evaluate its performance. The benchmark contained 100,480,507 ratings of 17,770 movies made by 480,189 users. The rating file size was 2.43GB. Both datasets were stored on HDFS and the data block size was set to the default value (i.e., 64MB). They have also been widely used in other works [13]–[15].

4) EXPERIMENTAL PROCEDURE

We set 3600 seconds as the timeout threshold for all the experiments, which was one order of magnitude higher than the time spent by these benchmarks to run to completion successfully. Setting a threshold one order of magnitude higher than the runtime needed to complete the baseline execution is a typical setting in software engineering experiments (e.g., [43]). We regarded a program execution as timeout if its total elapsed time of the execution larger than this threshold.

For answering RQ1 in Scenario 1, we applied the following procedure: We firstly ran each benchmark on Spark$_{\text{base}}$ because Spark$_{\text{base}}$ exhibited more constrained capacity in handling the benchmarks. More specifically, at the start of the concrete execution phase of the last dataset action (i.e., the last Spark job of each program execution), we failed one randomly chosen Spark executor, which caused the datasets in that executor inaccessible by other executors. This resulted in the following situation: all the partition instances

[3] https://aws.amazon.com/ec2/instance-types/
[4] https://azure.microsoft.com/en-us/services/virtual-machines/

on the failed executor were cleaned up and Spark$_{\text{base}}$ arranged some other executors to re-generate the required missing partition instances and continued to complete the program execution. This strategy to fail a program execution was also used in [2] and [6].

For each program execution that we failed its executor, if the whole program execution did not finish before the timeout threshold, we terminated the program execution, and ended the experiments on this benchmark. Otherwise, we kept the execution logs, increased the number of iterations (starting from one) by one, and executed the benchmark. In short, we systematically increased the number of iterations until running a benchmark resulted in a timeout. As we are going to present in Section C, Spark$_{\text{base}}$ can handle 12 iterations on average on this set of benchmarks.

We then ran each benchmark on FAR using the same procedure. However, on each benchmark, FAR did not result in a timeout in each of the first 20 runs (i.e., program executions with one, two, ..., 20 iterations). We also observed that the differences in various metrics between Spark$_{\text{base}}$ and FAR have become greater than an order of magnitude. We therefore ended the collection of statistics on FAR after the first 20 experiments on each benchmark. We note that using such a gap in duration to end an experiment is a typical setting in the experiments used in software engineering [44].

To answer RQ1 in Scenario 2, for each benchmark running on each of Spark$_{\text{base}}$ and FAR, we systematically varied the available memory that could be used by each executor. More specifically, for each benchmark, we configured it with 20 iterations and set the available memory for each executor to be 14GB, 13GB,..., 1GB to execute the benchmark. When each executor allocated with more than 8GB memory, the system did not remove any partition instance during the executions as there were sufficient memory locations for persisted partition instances, and when the memory allocation set to 1GB, the executions failed with the *OutOfMemoryError* exception due to insufficient heap memory space. Therefore, in this experiment, we only analyzed the data on these trials with 8GB, 7GB, ..., 2GB as the memory size of each executor. To constrain the memory used by an executor, we configured the *spark.executor.memory* parameter, limiting executors to use no more than a certain amount of memory on the node, and thus the memory locations for persistent partition instances were also limited.

To answer RQ2, for each benchmark, we ran it for a fixed number of iterations. To make all the experiments consistent, we set the iteration number to 20, which was also the same with experiments in RQ1. We ran each benchmark with Spark$_{\text{base}}$ and FAR, respectively, for 20 times.

C. ANSWERING RQ1 THROUGH SCENARIO 1

For each program execution, we measured the total elapsed time to complete the execution. Fig. 5 shows the results. The upper plot is for Spark$_{\text{base}}$ and the lower one is for FAR. In each plot, there are six lines, one line for each benchmark. Each point indicates a program execution where

TABLE 2. Speedup observed before timeout.

Benchmark	Avg. Re-generation Time (s)		Mean Speedup	Max. Re-generation Time (s)		Max. Speedup
	Spark$_{\text{base}}$	FAR		Spark$_{\text{base}}$	FAR	
PR	359.7	31.1	11.6	1946.5	42.8	45.5
WCC	321.0	32.4	9.9	1873.2	51.0	36.7
SP	409.9	31.0	13.2	3211.7	47.9	67.0
SVD++	371.3	101.7	3.7	1258.3	173.4	7.3
GMM	352.6	53.2	6.5	2070.2	89.0	23.2
BFS	316.1	39.6	8.0	2192.2	67.4	32.5

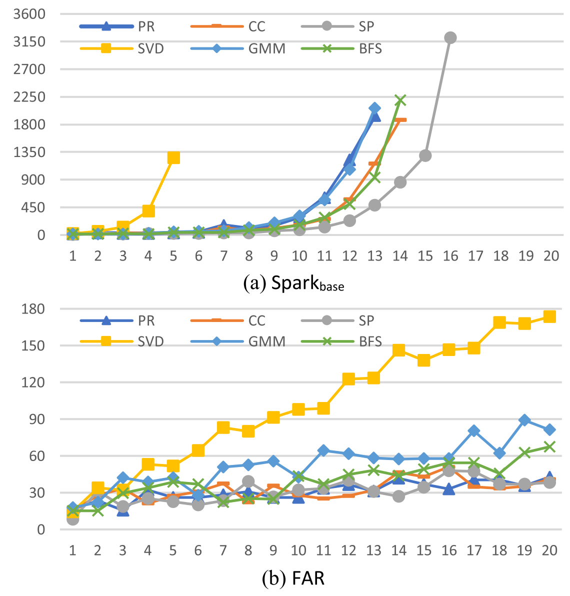
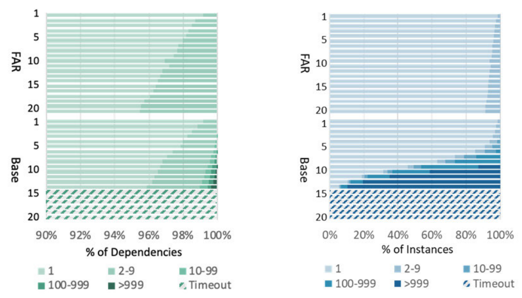


FIGURE 5. Time spent (in seconds) on re-generation of partition instances. The x-axis is the number of configured iterations.

the x-value represents the number of iterations configured in the corresponding benchmark. The y-axis is the time spent on handling inaccessible partition instances. The time spent is calculated by $t_x - t'_x$, where t_x and t'_x are the total elapsed time spent by the program execution with and without triggering re-generation of partition instances. Thus, the y-value of each point represents the overhead to handle re-generations of such inaccessible partition instances incurred by either Spark$_{\text{base}}$ or FAR for these program executions.

We observed that in both plots, the overall trend is that longer time is spent as x increases. However, after some point of x-value, the time spent for Spark$_{\text{base}}$ increases rapidly, and hits the timeout threshold (as indicated by the omissions of points). More specifically, Spark$_{\text{base}}$ hits the timeout limit after $x = 5$, 13, 13, 14, 14, and 16 for benchmarks SVD++, PR, GMM, BFS, WCC, and SP, respectively. While for FAR, the spent time increases gently without omission in the plot. Also, on each benchmark, FAR takes either similar or much less time than Spark$_{\text{base}}$ for the same x-value.

We have also computed the speedup achieved by FAR over Spark$_{\text{base}}$. Table 2 summarizes the results. The first column represents the benchmark. We collected the time



(a) Dependency Level (b) Partition Instance Level

FIGURE 6. **Dependency and partition instance histograms for BFS.**

spent incurred by both Spark$_{base}$ and FAR for these program executions without timeout. The 2nd–4th (5th–6th) columns show the mean (maximum) time spent by Spark$_{base}$, FAR, and the ratio of them in these executions.

Table 2 shows that compared to Spark$_{base}$, enabling FAR can achieve 3.7x to 13.2x mean speedup and 7.3x to 67.0x maximum speedup for the six benchmarks. The average of these mean and maximum speedups is 8.8x and 35.4x. The improvement of FAR over Spark$_{base}$ is large.

To analyze whether there is significant difference between FAR and Spark$_{base}$ in reusing partition instances, we further analyzed individual program executions. We repeated the experiment, but at this time, we inserted logging statements in the source code to measure how many times each dependency had been established and how many partition instances had been generated during each program execution.

We analyze the situations at the *dependency level* and the *partition instance level* as follows.

At the dependency level, for each execution, we collected the dependencies from G_P, and grouped them into five categories by their establishment times (i.e., once, 2 – 9 times, 10 – 99 times, 100 – 999 times, and more than 999 times). We computed a normalized histogram and the total number of dependencies is scaled to 100%.

At the partition instance level, we recorded all the generated partition instances for each execution and grouped them into five categories by the count of equivalent instances. We summed up the total number of partition instances in each category, normalized the histogram on them so that the total number of partition instances is scaled to 100%.

To avoid overloading the readers, for brevity, we only show the results of BFS in Fig. 6. The results of other benchmarks are provided in Appendix I. Fig. 6 shows the normalized histograms at dependency level (left) and at the partition instance level (right) on the BFS benchmark, in which we use different color depths to depict different categories. In each plot, the upper and lower sections show the results from program executions with FAR and Spark$_{base}$, respectively. The program executions resulted in timeout are shown as bars with striped lines. The y-axis is the index of the program execution to obtain the raw data.

In Fig. 6(a), the results show that for all these executions, most dependencies (more than 95%) were established only once. As the execution index increases, more dependencies were involved due to missing partition instances. For the last executions from both FAR and Spark$_{base}$, about 5% of dependencies were established more than once. Only a tiny part (less than 1%) was established more than 999 times in Spark$_{base}$.

However, although the difference is small between FAR and Spark$_{base}$ in Fig. 6(a), failure to share partition instances during the establishment of dependencies resulted in performance bloat. Fig. 6(b) summarizes the findings. From the section on FAR, we can see that most partition instances (more than 90%) were grouped under the first category (i.e., each partition were generated only once), and all other partition instances were grouped under the second category (more specifically, each one had less than three equivalent partition instances). From the section on Spark$_{base}$, as the execution index increases, partitions were generated many times and led to a large percentage of all partition instances. In the last row for Spark$_{base}$ (before timeout), each partition had more than 999 equivalent partition instances, and accounts for 90% of all instances.

TABLE 3. **Time spent (in seconds) with different executor memory allocations.**

Executor Memory (GB)	PR		WCC		SP		SVD++		GMM		BFS	
	Spark$_{base}$	FAR	Spark$_{base}$	FAR	Spark$_{base}$	FAR	Spark$_{base}$	FAR	Spark$_{base}$	FAR	Spark$_{base}$	FAR
8.0	205	205	183	184	159	162	340	345	465	454	257	251
7.0	200	198	183	183	166	168	347	349	480	487	255	252
6.0	199	213	185	182	161	160	353	381	482	486	265	267
5.0	196	221	194	197	166	167	464	401	522	522	273	274
4.0	244	248	214	213	266	266	483	401	-	1868	286	296
3.0	-	1794	-	1133	-	1325	413	384	-	2136	-	1236
2.0	-	2702	-	2217	-	2306	-	440	-	-	-	3027
1.0	/	/	/	/	/	/	/	/	/	/	/	/

* The symbol "-" indicates timeout during the execution.

** The symbol "/" indicates executions failed with OOM exception.

D. ANSWERING RQ1 THROUGH SCENARIO 2

In Scenario 2, following [26], we measured the *Runtime* spent by each execution as presented in Spark's Web UI, which is a web interface to monitor and inspect application executions.

Table 3 shows the time spent in each configuration with different fraction of maximum available memory. From the results, when each executor's memory changed from 8 GB to 5 GB, all program executions for both Spark$_{base}$ and FAR finished without experiencing any timeout. Moreover, the differences in time spent between the two techniques are small, indicating that FAR had a comparable performance as Spark$_{base}$ under less restrictive memory scenarios in the experiment.

However, as the available memory situation became more stringent, Spark$_{base}$ did not continue to scale well and started to perform much poorer than FAR. When the fraction changed from 5 GB to 3 GB, the program executions under Spark$_{base}$ started to result in timeout. In contrast, although the time spent incurred by FAR increased by a large margin, program executions still completed before the timeout threshold. In particular, we observed that in the experiment, program

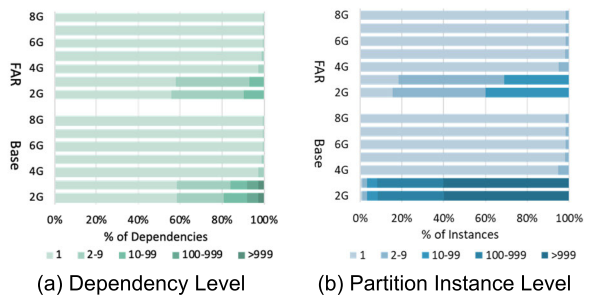


FIGURE 7. Dependency and partition instance histograms for BFS executions with different execution memory allocation.

TABLE 4. Relative runtime (in seconds) overhead of FAR.

Benchmark	Time Spent (s)		Ratio	P-value
	Spark$_{base}$	FAR		
PR	196.2 (4.3)	197.5 (4.8)	1.0066	0.37
WCC	182.9 (5.3)	182.0 (3.7)	0.9954	0.57
SP	164.7 (4.4)	167.0 (3.5)	1.0139	0.08
SVD++	333.7 (22.8)	322.0 (24.3)	0.9649	0.12
GMM	451.5 (11.6)	449.5 (15.4)	0.9955	0.64
BFS	253.0 (6.3)	253.0 (6.7)	1.0003	0.97
Mean	-	-	**0.9961**	-

Note: The standard deviation is shown in the parentheses.

executions on Spark$_{base}$ changed from running normally to resulting in a timeout within a change of 1 GB of available memory, whereas execution performance on FAR degraded more gracefully. Finally, for the executions with only 2 GB as the maximum available memory, all the program executions supported by FAR were completed before timeout except for GMM. The overall results indicate that FAR has the potential to complete program executions much earlier than Spark$_{base}$ in memory stringent situations.

To further analyze the above program executions, we measured the number of established dependencies and partition instances as what we summarized in Fig. 6. But this time, we only varied the amount of memory assigned to executors across executions. For executions resulted in timeouts (where timeout was set to 3600 seconds as well), we calculated the results of the executions right before timeout. We grouped these partition identifiers into five categories using the same scheme that we used in Fig. 6.

Fig. 7 shows the results of BFS, and the results of other benchmarks are provided in Appendix I.

There are two plots in Fig. 7, in each plot, the upper and lower parts are the executions with FAR and Spark$_{base}$, respectively. From Fig. 7, for program executions which completed successfully using either technique (Spark$_{base}$ and FAR), only a small ratio of dependencies had been re-established, and the number of equivalent partition instances were always less than nine.

However, when the available memory became smaller (i.e. less than 4GB), the program executions using Spark$_{base}$ started to result in timeouts. From Fig. 7, the results show that the executions using Spark$_{base}$ generated thousands of equivalent instances of some data partitions and these instances took up a very large proportion of all instances generated. As a comparison, with the same memory allocation, each partition instance in program executions using FAR was always associated less than 99 equivalent partition instances. Take BFS with 3GB memory allocation as an example. Around 60% of all instances using Spark$_{base}$ were generated more than 999 times. More specifically, the maximum and mean counts of equivalent partition instances in the fifth category (i.e., larger than 999) were 2049 and 1281. For its counterpart using FAR, the maximum and mean numbers of equivalent partition instances in its third category (i.e., between 10 and 99) were 20 and 13, respectively. There were two orders of magnitude in difference.

Across all the executions, for both Spark$_{base}$ and FAR, the percentage of partition instances in the first category (i.e., generated only once) dropped from around 90% to a small value (less than 20% for FAR, and less than 5% for Spark$_{base}$), which indicates multi-fold increases in the number of equivalent partition instances. The result indicates that these program executions incurred serious re-generation slowdown overheads when the total available memory locations were insufficient to keep all partition instances that these program executions (with their underlying techniques Spark$_{base}$ and FAR) aimed to keep at the same time. It also reveals that system deletion of persistent instances has a significant impact on execution slowdowns in this scenario.

E. ANSWERING RQ2 THROUGH SCENARIO 3

Similar to the time spent data used in the last subsection, the time spent data in this experiment were extracted from Spark's Web UI. Table 4 summarizes the mean and standard deviation of the time spent (in seconds) of the program executions using Spark$_{base}$ and FAR. The third column shows the ratio of the two columns on its left and the ratio is calculated as *mean time spent of FAR* $\div$ *mean time spent of Spark$_{base}$*. For this ratio, a value over 1 indicates that FAR is slower than Spark$_{base}$; otherwise, FAR has not been observed to have a disadvantage in terms of the slowdown. To analyze the statistical significance, we also ran one-way ANOVA analysis [37], of which the P-values are shown in the rightmost column.

From the Ratio column, FAR-enabled executions caused slight overheads (0.03% – 1.39%) in three of all the six benchmarks, and speedups (0.45 – 3.51%) for the other three. In terms of statistical significance, all the P-values are larger than 0.05, indicating no significant differences between each two groups of program executions has been found at 5% significance level. Overall speaking, FAR and Spark$_{base}$ achieved comparable runtime overheads for all the 240 conducted program executions.

In addition, as Checkpointing is a widely applied fault tolerances strategy to executions, we assessed our approach FAR with checkpointing and found FAR more efficient in handling program executions. More details of the comparison are provided in Appendix II.

F. THREATS TO VALIDITY

The presence of platform bugs remains a major threat of the experiment. FAR needs to calculate accurate dependencies from these activities. Inaccurate results will lead FAR to make wrong decisions on keeping or discarding partition instances. However, as the underlying environment is not guaranteed to be reliable, the completeness of lifecycle activities is not guaranteed either. Although we did not observe that we had encountered this issue in the experiments, it is still possible that FAR may cause memory leak during the execution in other settings.

All the experiments were conducted on the VMware ESXi hypervisor [10]. In general, the virtual machines are isolated and would not interfere with each other. However, when the workload is accidentally high on each and every machine at the same time, there could be preemptions on hardware resources, which yield unstable results. To overcome this threat, we only kept the cluster node virtual machines running during the experiment to avoid the irrelevant workload interfering the experimental results.

The third one lies in the representativeness of the benchmarks. The selected benchmarks used for evaluating FAR have frequent partition operations. Among the six benchmarks, four of them are provided by Spark and widely used in the experiments of prior studies [25], [26]. To obtain greater generalizability, we have searched the open-sourced benchmarks in GitHub, only found two others that incur performance bloat problems are based on iterative algorithms. There were benchmarks but we could not obtain their datasets or these benchmarks could be run on our platforms. However, these benchmarks are all implementations of specific algorithms, different from full-bone industry-strength applications. Readers should interpret our experimental results with care.

The implementation of our technique may contain bugs. We have tested it with small programs and examined the data generated from the benchmarks. We did not observe abnormality.

The experiment was found very time-consuming due to the processing of large datasets and generation of many datasets. Because of the limited resources we can afford, the scale of our experiment could not be scale up further. Within our resource limit, we have evaluated our technique carefully by varying different values for different parameters systematically. Using platforms with other capacity and processing power will certainly give new absolute results. However, we tend to believe that FAR will still outperform Spark$_{\text{base}}$ in scenarios incurring performance bloat due to excessive partition instance generation and deletion.

We only used the memory consumptions and execution time as the metrics. Using other metrics may give different results. However, we tend to believe that the performance bloat problem being solved by FAR will still make our technique have a competitive advantage over Spark$_{\text{base}}$.

VI. FURTHER DISCUSSIONS

In [2], the authors of Spark$_{\text{base}}$ also conducted two experiments to evaluate the performance of Spark$_{\text{base}}$ after a node failure and with insufficient memory (i.e., Scenario 1 and Scenario 2 in this work). More specifically, for Scenario 1, they ran 10 iterations of k-means on a 75-node cluster while killing a node at the start of the 6th iteration. Each iteration cost was about 58 seconds except for the 6th iteration, which took 81 seconds as it reconstructed the lost RDD partition instances. For Scenario 2, they ran logistic regression benchmark on 25 machines with varying amounts of data in memory. The results showed that the iteration time increased from 11.5 seconds to 40.7 seconds and 68.8 seconds when 100%, 50% and 0% memory were configured as the storage space.

However, they didn't encounter severe performance decrease as we had in our experiments. There are two main reasons for this difference. First, both k-means and logistic regression are simple benchmarks that do not create complex lineage graphs. Second, even with complex benchmarks, the issue could be staying unrevealed at an earlier stage of the program execution. In practice, applications are usually complex and long. Enabling FAR under such cases is more necessary and helpful.

In Section V.E, when the available memory is not stringent, the experiment results show that the advantage of FAR is small. However, in practice, there are various use cases that FAR may be applicable. In recent years, deep learning techniques achieved significant advancements. Models with more parameters and more sophisticated structures are proposed and applied to gain insights from massive amounts of data. A lot of work such as *SparkNet, TensorflowOnSpark*, *CaffeOnSpark* that integrate the prevalent deep learning libraries together with big data frameworks to enable distributed deep learning executions on Spark and Hadoop clusters [31]–[33]. In these scenarios, these deep learning programs need plenty of memory resources may run together with Spark and lead to dataset deletion problems. It appears to us that FAR has some potential to alleviate the possible dataset re-generation issues.

FAR also provides opportunities to support debugging and testing techniques on Big Data platforms. *Mutation testing* is an important kind of software testing technique that has been extensively studied in the past decades [30]. During its testing procedure, a set of program variants (called mutants) are generated by seeding small faults in the original program, and then comparisons are made among the original program and these mutants. Given a large number of program mutants and the needs of executing test cases over different program versions, mutation testing is computationally expensive. In addition, execution failures are common in the procedure of mutation testing. In the context of big data programs, these failures may lead to data loss and trigger the retry and recovery mechanism of big data platforms. As we have illustrated in this work, the runtime performance of the re-calculation could be significantly degraded, which makes mutation testing even harder to be applied. We believe that FAR has the potential to alleviate the situation by effectively identifying, persisting and deleting reusable datasets, which could make mutation testing be conducted more efficiently.

Apricot is a debugging technique for deep learning (DL) models by iteratively conducting weight-adaption on them [34]. It generates a set of smaller DL models of the original DL model for the purpose of fixing the latter model and uses them in its iterative process to gradually change and re-train the original DL model. We believe that FAR could be applicable in such a scenario if *Apricot* is run on a Big Data infrastructure, where partitions in FAR are mapped to batches in *Apricot*.

Compared to Spark$_{\text{base}}$, FAR may consume more memory to keep persistent datasets. However, if there is insufficient memory, as in the case of Scenario 2, FAR can still work gracefully. In FAR algorithm, it tracks outstanding uses on each partition in its analysis state, where the algorithm has not been optimized. For instance, deleting a partition with one outstanding use with a smaller ripple effect to re-generate other partitions may be preferable to deleting a partition with more outstanding uses with a larger ripple effect. We leave the investigation of optimizations on FAR as a future work.

VII. RELATED WORK

A. FAULT-TOLERANCE MECHANISMS

There are two major mechanisms for achieving fault-tolerance, lineage graph and checkpointing techniques.

As mentioned in section I, a lineage graph contains RDD instances and the dependency relations between them. If a dataset is missing due to whatever reasons, the information on the graph could be used to recompute the missing dataset to achieve fault-tolerance on data loss [2]. Different from the lineage graph built-in with Spark, FAR takes advantage of dependency relations at both RDD and partition instance levels, which provides finer-grained support. Furthermore, as we analyzed in section III, the hard-coded lineage graph is not always effective to guide the re-computation. By analyzing the execution plan of ongoing action and monitoring the partition instance lifecycle activities, FAR can actively repair the procedure of computations, as well as the whole execution.

There are many checkpointing techniques proposed in the literature [1], [4], [8], [16]–[20]. They periodically back up certain intermediate datasets to secondary storage and can restart the program executions from the saved points. Panda [3] employs the fine-grained checkpointing on the task outputs. It uses the tasks' intrinsic information, such as the size of output data and the distribution of task runtimes, to dynamically identify tasks to be checkpointed rather than recomputed [3]. Xu *et al.* proposed a fault-tolerance mechanism for Apache Flink [21]. Their technique injects checkpointing into each iteration and enables checkpoint to be written along with computing RDD values. As the CPU processing could partially overlap the I/O processing, the whole pipeline execution can archive higher efficiency.

We also conducted an exploratory study to assess FAR with the checkpointing approach. The results showed that FAR is more efficient in the comparison experiment. More details are provided in Appendix II.

B. DATASET MANAGEMENT IN DATA PROCESSING

Many existing works seek to find solutions to persistent dataset management for more efficient memory usage as well as better platform optimization [12], [22]–[26]. A major class of them proposes replacement policies for persisted partitions. If the available memory for a later execution is in shortage, such a policy will select and delete part of persisted datasets to release memory locations [27].

Yu *et al.* proposed a *Least Reference Count* (LRC) policy to replace the default LRU policy of Spark [26]. LRC exploits the lineage graph and deletes persisted partitions where the corresponding RDD instances have the least numbers of unevaluated children RDD instances. Geng *et al.* [22] proposed Least Cost Strategy (LCS) which predicts the future usage and recovery cost information of persisted partition instances from their dependency relationships and selectively deletes them.

Spark uses predefined and fixed parameters to reserve datasets for subsequent usage [8]. MemTune dynamically tunes these parameters by exploiting the task scheduling information at runtime to improve the overall memory resource utilization [25]. As Spark allows developers to choose from a few storage levels for RDD dataset storing, Neutrino provides adaptive storage levels and further optimized the memory use [24]. Their adaptive storage levels are chosen based on the access order of RDDs and runtime information [24]. Gounaris *et al.* investigated the trade-offs between performance and consumed CPU resource of Spark applications. They proposed algorithms to take both execution time and occupied resource into account by dynamically partitioning during the execution [42].

Unlike the techniques in this category, the goal of FAR is not to create a technique to select datasets to be deleted to release memory for other usages. FAR aims to define dataset reuses based on the information of the online lineage graph produced by the Spark platform and the progress of the program execution. Although an important feature of FAR is to delete partitions (if the partition has not been set into some other states by the program execution or the underlying platform) right after all the budgets on the partition instance have been consumed, as discussed in Section VI, FAR has not incorporated strategies to prioritize or select datasets to be deleted in other situations. In fact, unlike strategies like LRC and LCS which do not introduce additional datasets to be persisted, FAR makes retention decisions of partitions and actively changes the states of some partitions to be *persistent* ''for a while''. In this sense, partitions under the management of FAR are still ''transient'' but last longer than a partition in *ephemera* state and shorter than a partition in the *reserved/persistent* state. In the sense of state transition

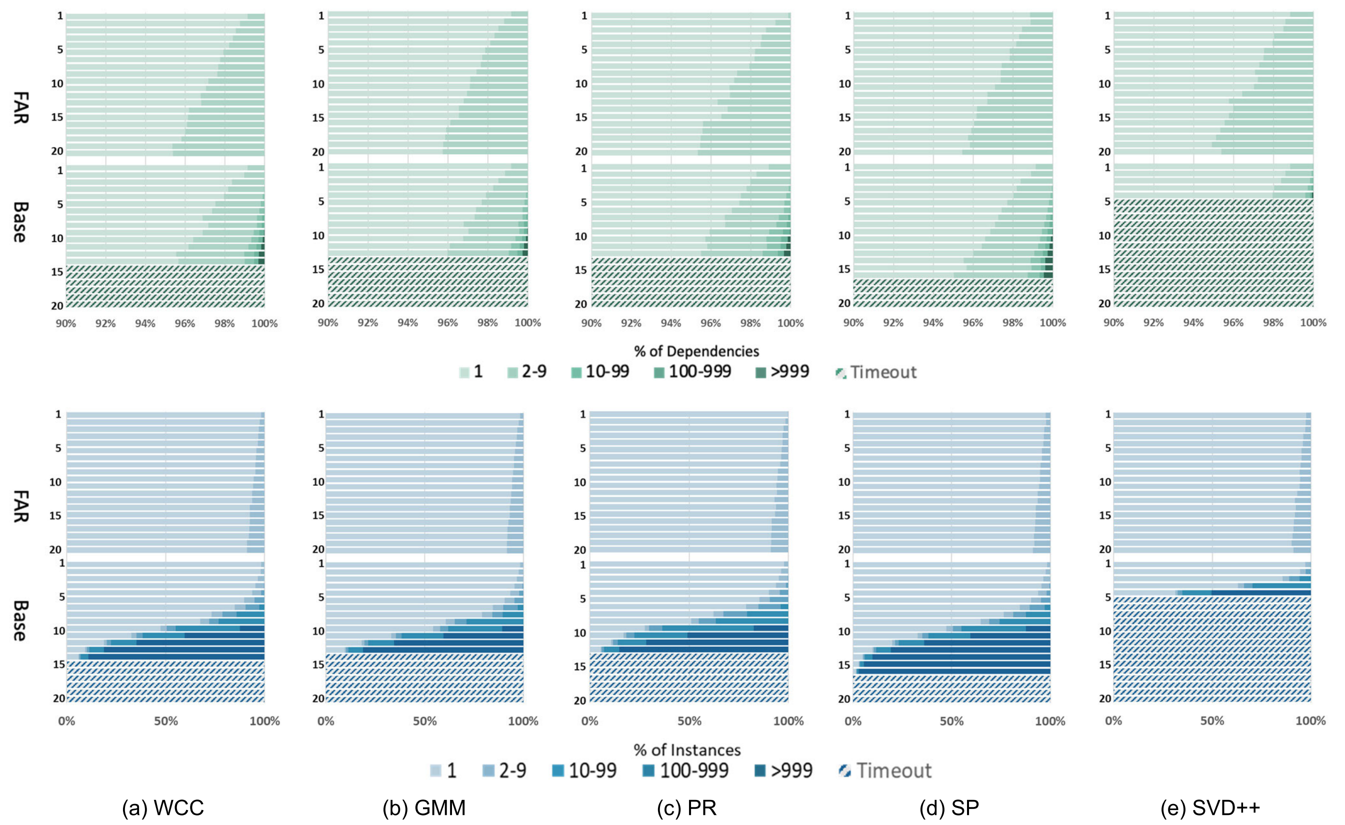


FIGURE 8. Re-generation histograms of executions each with one concrete execution phase failed. In each subfigure, there are four plots for the results of FAR and Spark$_{base}$ at the dependency level (the first subplot and the second subplot respectively) and at the partition instance level (the third subplot and the fourth subplot respectively).

depicted in Fig. 1, FAR introduces a new state and a set of related transitions to the state-transition diagram.

VIII. CONCLUSION

In this paper, we have proposed FAR, a novel execution repair framework to effectively maintain the partition instance dependencies for Spark program executions. To address the performance bloat problem, FAR provides high-level programming abstraction to help application developers to address the performance problem caused by excessive partition instance generation and deletion. We have presented the novel design and the algorithms of FAR. We have shown its feasibility by implementing it as a component in Spark. We have evaluated it using six benchmarks in different scenarios ranging from execution failures needing re-generation of some partition instances to running programs in environments with stringent available memory constraints. We have evaluated FAR in situations where there are sufficient system resources and there is no failure requiring partition instance re-generation. The results have shown that FAR has the potential to effectively and efficiently address a class of performance bloat in Spark applications. FAR transiently incurs higher memory overheads than Spark due to the needs to keep more persistent partitions temporarily, and the experiment shows that this strategy pays off well. The integrations with other fault tolerance or data management strategies are interesting to be further explored. We leave them as future work.

APPENDIX I
RE-GENERATION HISTOGRAMS OF OTHER FIVE BENCHMARKS IN SCENARIO 1 AND SCENARIO 2

Fig. 8 shows the dependency level and partition instance level histograms of five benchmarks in Scenario 1. We can observe that they followed similar trends as the BFS benchmark in Section V.C. To avoid overloading the readers, we do not repeatedly state similar observations.

Fig. 9 shows the re-generation histograms of five benchmarks in Scenario 2. We notice that SVD++ using *Netflix Prize dataset* consumed a small amount of memory. Therefore, in this experiment, all the program executions using FAR can complete without experiencing a large increase in execution time. However, under the extreme case with 2 GB memory, the execution using Spark$_{base}$ still yielded a large percentage of partition instances and failed to complete before timeout. On GMM, the program execution using FAR resulted in timeout when memory allocation was 2 GB. From Table 3 in Section V.D, GMM's executions always took longer time than corresponding executions of other benchmarks under the same memory allocations, indicating that GMM was more complex in processing than other benchmarks (which we have inspected the code and confirmed it). However, from Fig. 9, the numbers of partition instances in

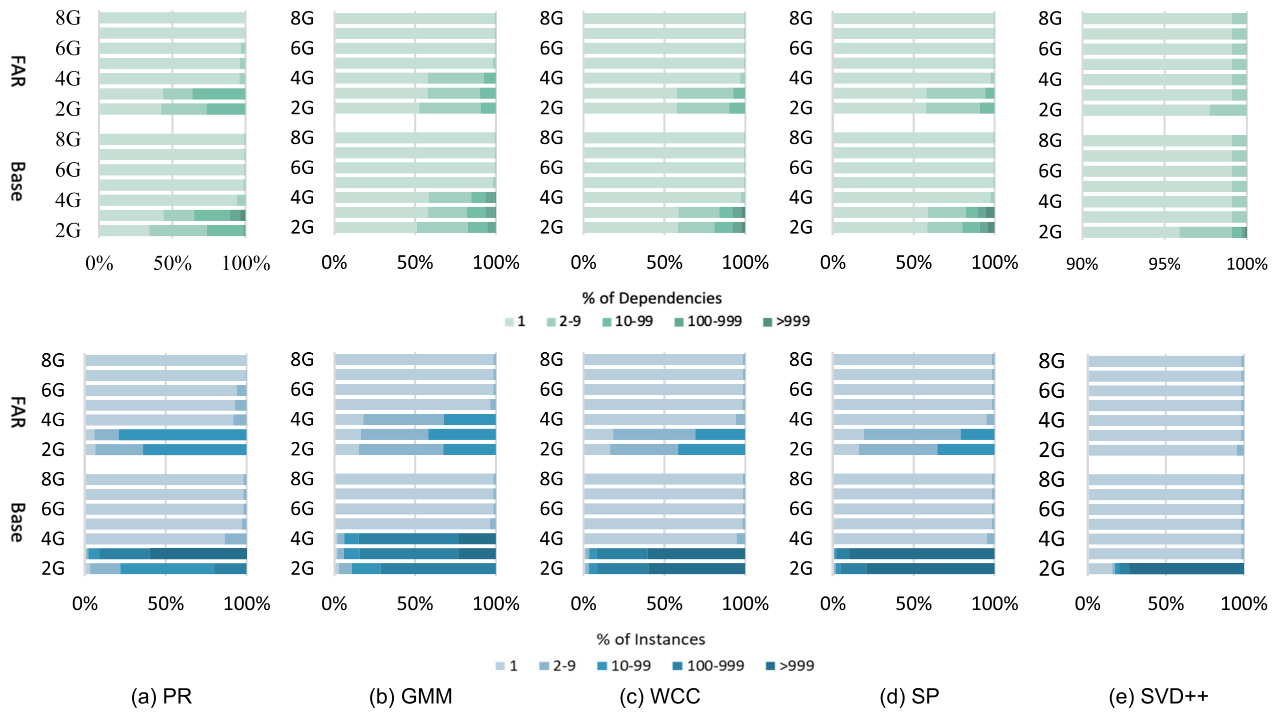


FIGURE 9. Re-generation histograms of executions with different execution memory allocation. In each subfigure, there are four plots for the results of FAR and Spark$_{base}$ at the dependency level (the first subplot and the second subplot respectively) and at the partition instance level (the third subplot and the fourth subplot respectively).

program executions using FAR were small (more specifically, no partition had more than 99 instances). We also executed the GMM benchmark under 2 GB available memory without setting any timeout threshold, it took 4552 seconds to complete. This result was 10.0x slower than the corresponding execution with 8 GB available memory, which is comparable with other benchmarks in this aspect. (We could not complete the execution when running on Spark$_{base}$ after several hours.)

APPENDIX II
FURTHER EXPLORATORY STUDY

Checkpointing is a widely applied strategy to provide fault tolerance to executions. With checkpointing, an intermediate dataset could be periodically shadowed (i.e., making a copy of the partition instances) even if it has not been persisted by the corresponding program execution. Intuitively, if a failure occurs, the missing partition instances can be retrieved from the snapshot captured via checkpointing.

Although checkpointing is inapplicable to Scenario 2 due to its additional memory overhead in shadowing intermediate datasets, intuitively, it can be applied to Scenario 1 and Scenario 3. We thus ask a question on whether FAR can be at least as good as checkpointing in handling program executions in Scenario 1 and Scenario 3?

To seek the answer to the above question, we implemented the above checkpointing procedure on Spark$_{base}$ to conduct a further comparison to FAR (and we are unaware of any available checkpointing implementation for Spark yet). We picked PR as the benchmark in this exploratory study. We chose PR as it is representative in all six benchmarks.

Spark provides the checkpoint interface on RDD. When a checkpoint operation is conducted on an RDD instance, the RDD instance is marked for checkpointing. A concrete checkpointing procedure is triggered once an action on such RDD instance is invoked. During checkpointing, the corresponding datasets are generated and saved to non-volatile storage. If a re-generation of the partition instance is needed in a program execution, Spark$_{base}$ can read the latest snapshot kept in the non-volatile storage during the requested re-generation.

To use checkpointing, we had to modify the source code of the PR benchmark by inserting the checkpoint statements. Specifically, we added code to checkpoint the intermediate RDDs when it finishes its 2nd, 7th, 12th, and 17th iterations. For ease of our discussion, we refer to this implementation as PR$_{CP}$. We compared PR$_{CP}$ with PR using FAR and PR using Spark$_{base}$. We measured their time spent in Scenario 1 where an executor was failed at the 3rd, 8th, 13th, 18th, 19th, and 20th iteration as well as their time spent in Scenario 3 (where timeout was set to 3600 seconds as well). We note that PR using Spark$_{base}$ resulted in timeouts after the 13th iteration.

Fig. 10 summarizes the results of these executions for the three techniques, shown as bars. Along the horizontal axis, the first slot "Start" denotes results of Scenario 3 where these executions run normally. The following slots show the program executions with an executor failed at the 3rd, 8th, 13th, 18th, 19th, and 20th iterations. We note that PR$_{CP}$ can only provide a restoration of the partition instance kept in the snapshot. Thus, PR$_{CP}$ still requires Spark$_{base}$ to re-generate the missing partition instances not in the snapshot.

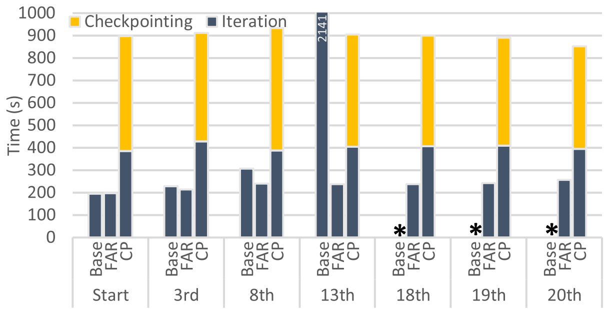


FIGURE 10. Runtime comparison among Spark$_{base}$, FAR and PR$_{CP}$ configurations. The symbol * indicates timeout.

In these program executions for PR$_{CP}$, the snapshots were checkpointed at the end of 2nd, 7th, 12th, and 17th iterations. For the program executions that failures occurred at 3rd, 8th, 13th, and 18th, the datasets to recover the missing partition instances could read directly from the snapshots. On the other hand, for program executions failed at the 19th and 20th iterations, the missing partition instances had to be re-generated by applying transformations on the latest snapshot of the 17th iteration. This setting allows us to evaluate FAR against PR$_{CP}$ in more diverse situations.

Similar to the results presented in the last subsection, the time spent of FAR and that of Spark$_{base}$ were similar. However, the time spent of PR$_{CP}$ was almost 4.2x than FAR. We found that the significant difference was due to the need of I/O operations for checkpointing datasets to the non-volatile storage.

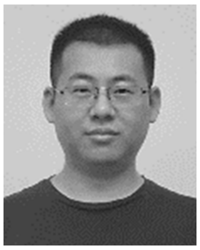
We also observe a side effect of using checkpointing interface. Each invocation of checkpointing will break the lineage graph of current execution into disconnected fragments. Because re-generation of partition instances in Spark$_{base}$ inherently requires the lineage graph to re-generate partition instances, the remaining fragments can only provide limited visibility on some previously generated instances. Therefore, a program execution might load partition instances from the snapshot without knowing that the equivalent instances might have been currently kept in the memory, which further slowed down the program execution and consumed more memory than expected.

Similar to FAR, PR$_{CP}$ did not introduce additional major runtime overheads when a program execution failed in later concrete execution phases, and yet PR$_{CP}$ spent 3.3x to 3.9x more time than FAR. Given that PR$_{CP}$ needs Spark$_{base}$ to support the re-generation of partition instances that are not kept in the snapshot. PR$_{CP}$ can be configured to run with FAR instead of Spark$_{base}$. That is to say, FAR and checkpointing are not competing techniques, rather they are complementary in nature. It also clarifies the nature of FAR that it is not a fault-tolerance technique.

REFERENCES

[1] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 1357–1369.
[2] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Networked Syst. Design Implement.*, 2012, p. 2.
[3] B. Ghit and D. Epema, "Better safe than sorry: Grappling with failures of in-memory data analytics frameworks," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, Jun. 2017, pp. 105–116.
[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, p. 10.
[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 1383–1394.
[6] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Design Implement.*, 2014, pp. 599–613.
[7] Ericpony. *GraphX Playground*. Accessed: May 4, 2020. [Online]. Available: https://github.com/ericpony/graphx-playground
[8] *Apache SparkTM*. Accessed: May 4, 2020. [Online]. Available: https://spark.apache.org/
[9] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Math. Program.*, vol. 73, no. 2, pp. 129–174, May 1996.
[10] *VMware vSphere Hypervisor*. Accessed: May 4, 2020. [Online]. Available: https://www.vmware.com/products/vsphere-hypervisor.html
[11] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Conf. World Wide Web (WWW)*, 2004, pp. 595–602.
[12] J. Bennett and S. Lanning, "The netflix prize," in *Proc. KDD Cup Workshop*, 2007, pp. 3–6.
[13] K. Lee, L. Liu, K. Schwan, C. Pu, Q. Zhang, Y. Zhou, E. Yigitoglu, and P. Yuan, "Scaling iterative graph computations with GraphMap," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2015, p. 57.
[14] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proc. VLDB Endowment*, vol. 7, no. 7, pp. 577–588, Mar. 2014.
[15] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux^2: Distributed graph computation for machine learning," in *Proc. NSDI*, 2017, pp. 669–682.
[16] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, "A fault-tolerant framework for asynchronous iterative computations in cloud environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1678–1692, Aug. 2018.
[17] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "TR-spark: Transient computing for big data analytics," in *Proc. 7th ACM Symp. Cloud Comput.*, Oct. 2016, pp. 484–496.
[18] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.
[19] Apache Software Foundation. (Aug. 2015). *Apache Flink*. Apache.Org. [Online]. Available: https://flink.apache.org/
[20] Apache Software Foundation. (Aug. 2016). *Apache Storm*. [Online]. Available: https://storm.apache.org/
[21] C. Xu, M. Holzemer, M. Kaul, and V. Markl, "Efficient fault-tolerance for iterative graph processing on distributed dataflow systems," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 613–624.
[22] Y. Geng, X. Shi, C. Pei, H. Jin, and W. Jiang, "LCS: An efficient data eviction strategy for spark," *Int. J. Parallel Program.*, vol. 45, no. 6, pp. 1285–1297, Dec. 2017.
[23] T. B. G. Perez, X. Zhou, and D. Cheng, "Reference-distance eviction and prefetching for cache management in spark," in *Proc. 47th Int. Conf. Parallel Process.*, Aug. 2018, p. 88.
[24] E. Xu, M. Saxena, and L. Chiu, "Neutrino: Revisiting memory caching for iterative data analytics," in *Proc. USENIX Workshop Hot Topics Storage File Syst.*, 2016, pp. 16–20.
[25] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE: Dynamic memory management for in-memory data analytic platforms," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 383–392.
[26] Y. Yu, W. Wang, J. Zhang, and K. Ben Letaief, "LRC: Dependency-aware cache management for data analytics clusters," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, May 2017, pp. 1–9.
[27] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982.
[28] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. 7th Int. Conf. World Wide Web*, 1998, pp. 107–117.

[29] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, Jan. 2003, pp. 993–1022.
[30] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Jun. 2011.
[31] *Caffe on Spark*. Accessed: May 4, 2020. [Online]. Available: https://github.com/yahoo/CaffeOnSpark
[32] *Tensorflow on Spark*. Accessed: May 4, 2020. [Online]. Available: https://github.com/yahoo/TensorFlowOnSpark
[33] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training deep networks in spark," 2015, *arXiv:1511.06051*. [Online]. Available: http://arxiv.org/abs/1511.06051
[34] H. Zhang and W. K. Chan, "Apricot: A weight-adaptation approach to fixing deep learning models," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 376–387.
[35] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and its applications," in *25th Int. Symp. Fault-Tolerant Computing. Dig. Papers*, 1995, pp. 22–31.
[36] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. VLDB Endowment*, vol. 8, no. 13, 2015, pp. 2110–2121.
[37] D. C. Howell, *Statistical Methods for Psychology*, 6th ed. Belmont, CA, USA: Thomson Wadsworth, 2007.
[38] *Apache Giraph*. Accessed: May 4, 2020. [Online]. Available: https://giraph.apache.org/literature.html
[39] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.
[40] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 544–557.
[41] Y. Koren, "Factorization meets the neighborhood: A multifaceted collaborative filtering model," in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2008, pp. 426–434.
[42] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1891–1904, Jul. 2017.
[43] E. Pobee and W. K. Chan, "AggrePlay: Efficient record and replay of multi-threaded programs," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 567–577.
[44] Y. Cai and W. K. Chan, "Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs," *IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 266–281, Mar. 2014.



XIUPEI MEI received the B.Eng. degree from the School of Computer Science and Engineering, Beihang University (BUAA), China, in 2012. He is currently pursuing the Ph.D. degree in computer science with the City University of Hong Kong. His main research interests include program debugging and analysis in big data platforms, and concurrency bug detection in multithreaded programs.

IMRAN ASHRAF received the B.S. degree in computer and information sciences from the Pakistan Institute of Engineering and Applied Sciences (PIEAS), in 2011, with a Gold Medal Award. He is currently pursuing the Ph.D. degree in computer science with the City University of Hong Kong. His current research interests include program analysis and security vulnerability detection in decentralized platforms, such as Ethereum blockchains. His primary research interests include security vulnerability detection techniques are fuzz testing, static analysis, and symbolic execution.

XIAOXUE MA received the B.Eng. degree (Hons.) in telecommunication engineering from the College of Physical Science and Technology, Central China Normal University (CCNU), China, in 2017. She is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong. Her current research interest includes dynamic program analysis on concurrency bug detection in multithreaded programs.

HAO ZHANG received the B.Eng. and M.Sc. degrees from Beihang University, China, in 2015 and 2018, respectively. He is currently pursuing the Ph.D. degree in computer science with the City University of Hong Kong. His main research interests include interplay between software engineering and artificial intelligence, as well as program debugging. His work has been reported in ASE and TOSEM.

ZHENGYUAN WEI received the B.Eng. degree (Hons.) in software engineering from the School of Data and Computer Science, Sun Yat-sen University (SYSU), China, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong. His research interests include deep learning robustness and efficiency, and compiler optimization.

HAIPENG WANG (Graduate Student Member, IEEE) received the B.Eng. degree in software engineering from the Department of Computer Science, Beijing Institute of Technology, and the master's degree in data engineering from the Department of Computer Science, City University of Hong Kong, where he is currently pursuing the Ph.D. degree. His current research interest includes software engineering in AI.

W. K. CHAN (Member, IEEE) is currently an Associate Professor with the City University of Hong Kong. His research interests include software engineering in general, program analysis and testing for concurrent software and systems, and software infrastructure for AI-based systems. He is currently a Special Issues Editor of *Journal of Systems and Software*, an Associate Editor of *Software Testing, Verification and Reliability*, a Review Editor of *Array*, a Guest Editor of IEEE TRANSACTIONS ON RELIABILITY, the Program Chair of COMPSAC 2020 and QRS 2020, and a Program Committee Member of ECSE/FSE 2020, ASE 2020, and ICSE 2021. His research results have been reported in more than 100 articles with venues, including but not limited to *ACM Transactions on Software Engineering and Methodology* (TOSEM), IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (TSE), IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS), *ACM Transactions on Social Computing* (TSC), TRel, *Communications of the ACM* (CACM), *Computer*, ICSE, FSE, ISSTA, ASE, WWW, ICWS, and ICDC.