Detecting Cache-Related Bugs in Spark Applications

Hui Li*
Dong Wang*
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{lihui2012,wangdong18}@otcaix.iscas.ac.cn

Tianze Huang*
Beijing University of Posts and Telecommunications
Beijing, China
huangtianze@bupt.edu.cn

Yu Gao
Wensheng Dou†
Lijie Xu
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{gaoyu15,wsdou,xulijie}@otcaix.iscas.ac.cn

Wei Wang
Jun Wei
Hua Zhong
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{wangwei,wj,zhonghua}@otcaix.iscas.ac.cn

ABSTRACT

Apache Spark has been widely used to build big data applications. Spark utilizes the abstraction of Resilient Distributed Dataset (RDD) to store and retrieve large-scale data. To reduce duplicate computation of an RDD, Spark can cache the RDD in memory and then reuse it later, thus improving performance. Spark relies on application developers to enforce caching decisions by using *persist*() and *unpersist*() APIs, e.g., *which* RDD is persisted and *when* the RDD is persisted / unpersisted. Incorrect RDD caching decisions can cause duplicate computations, or waste precious memory resource, thus introducing serious performance degradation in Spark applications. In this paper, we propose *CacheCheck*, to automatically detect cache-related bugs in Spark applications. We summarize six cache-related bug patterns in Spark applications, and then dynamically detect cache-related bugs by analyzing the execution traces of Spark applications. We evaluate CacheCheck on six real-world Spark applications. The experimental result shows that CacheCheck detects 72 previously unknown cache-related bugs, and 28 of them have been fixed by developers.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; *Reliability*; • **Software and its engineering → Software testing and debugging**; *Software maintenance tools.*

KEYWORDS

Spark, cache, bug detection, performance

ACM Reference Format:
Hui Li, Dong Wang, Tianze Huang, Yu Gao, Wensheng Dou, Lijie Xu,Wei Wang, Jun Wei, and Hua Zhong. 2020. Detecting Cache-Related Bugs in Spark Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397353

*Hui Li, Dong Wang and Tianze Huang are equal contributors to the paper.
†Corresponding author

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397353

1 INTRODUCTION

Apache Spark [50, 51] has become one of the most popular big data processing systems, and has been widely used in many big Internet companies, e.g, Facebook [18], Alibaba [2], eBay [19]. As an in-memory big data framework, Apache Spark can provide order of magnitude of performance speedup over disk-based big data frameworks, e.g., Apache Hadoop [1]. Especially, Apache Spark works better on iterative analytics, e.g, machine learning [37], and graph computation [33], which requires data to be shared across different iterations.

A Spark application consists of two kinds of operations: *transformation* (e.g., *map* and *filter*) and *action* (e.g., *count* and *take*). Spark utilizes the abstract of *Resilient Distributed Dataset* (RDD) to store and retrieve large-scale data, which is a read-only collection of objects partitioned across a set of machines. Transformations are used to create RDDs, and actions are used to obtain actual computation results on RDDs. Once an action completes, all its used RDDs are discarded from memory. To reuse an RDD loaded by an action, Spark provides *persist*() API to manually cache[1] the loaded RDD. Thus, the following actions can reuse the cached RDD, and avoid duplicate computations. Note that, an RDD can be persisted in different cache levels, e.g., in memory and on disk. Spark provides *unpersist*() API to manually release cached RDDs when they

[1]We use cache and persist interchangeably in this paper.

will not be used by following actions, thus saving precious memory. Figure 1a shows an example about how to reuse RDD *words* (Section 2.1).

Apache Spark relies on developers to enforce caching decisions through *persist*() and *unpersist*() APIs. Developers need to clearly figure out *which* RDD should be cached, *when* the RDD is persisted and unpersisted. The lazy evaluation of RDDs in Spark (Section 2.3) further complicates cache mechanisms, e.g., a *persist*() operation must be performed before its corresponding *action* operation. Improper caching decisions on RDDs usually cause performance degradation, even worse, out of memory errors. For example, if an RDD used by two actions is not persisted, the RDD will be computed twice, and thus degrading the application performance (e.g., SPARK-1266 [3]). If a cached RDD is not used anymore, it occupies much precious memory, and affects the normal execution (e.g., SPARK-18608 [16]). In our experiment, improper caching decisions can seriously degrade the performance of Spark applications (0.1% − 51.6%), and even cause Out of Memory (OOM). In this paper, we refer to improper caching decisions on RDDs as *cache-related bugs*.

To improve the performance of Spark applications, existing studies mainly focus on improving memory usage in the Spark system, e.g., Yak [39], AstroSpark [45] and Simba [31], and optimizing cache mechanisms in the Spark system, e.g., Neutrino [47], LRC [49] and Lotus [43]. However, cache-related bug patterns and detection approaches in Spark applications have not been studied. Note that, Spark applications have the different execution model from traditional programs (e.g., C/C++ and Java). Thus, the memory / resource leak detection techniques in traditional programs [32, 34, 40] cannot be directly applied on cache-related bug detection in Spark applications.

In this paper, we first analyze the execution semantics of cache mechanisms in Spark, and summarize six bug patterns for cache-related bugs in Spark applications, i.e., *missing persist*, *lagging persist*, *unnecessary persist*, *missing unpersist*, *premature unpersist*, and *lagging unpersist* (Section 3). We then propose *CacheCheck* to automatically detect cache-related bugs in Spark applications. After running a Spark application (without any modification or instrumentation to the application) on our modified Spark system (Section 5), CacheCheck collects the application's execution trace and caching decisions. Then, CacheCheck infers the correct caching decisions for the application based on the execution semantics of cache mechanisms in Spark. Finally, by analyzing the difference between the original caching decisions and inferred correct caching decisions, CacheCheck reports the inconsistencies as cache-related bugs.

We implement CacheCheck, and make it publicly available at https://github.com/Icysandwich/cachecheck. We evaluate its effectiveness on real-world Spark applications, including GraphX [22] and MLlib [23] and Spark SQL [29], etc. Experimental results show that: (1) CacheCheck can precisely detect all 18 known cache-related bugs in Spark applications. (2) CacheCheck detects 72 previously-unknown cache-related bugs on the latest versions of six Spark applications. We have reported these cache-related bugs to the concerned developers. So far, 53 have been confirmed by developers, of which 28 have been fixed. Developers also express their great interests in CacheCheck.

We summarize the main contributions of this paper as follows.

- We summarize six cache-related bug patterns in Spark applications, which can seriously degrade the performance of Spark applications.
- We propose an automated approach, CacheCheck, to detect cache-related bugs in Spark applications.
- We implement CacheCheck and evaluate it on real-world Spark applications. The experimental results show that it can detect cache-related bugs effectively.

The remainder of this paper is organized as follows. Section 2 explains Spark programming model, cache mechanism, and Spark execution semantics briefly. Section 3 presents the six cache-related bug patterns in Spark applications. Section 4 presents our cache-related bug detection approach. Section 5 presents our CacheCheck implementation, and Section 6 evaluates CacheCheck on real-world Spark applications. Section 7 and Section 8 discuss threats and related work, and Section 9 concludes this paper.

2 BACKGROUND

In this section, we briefly introduce the programming model and the cache mechanism in Apache Spark.

2.1 Spark Programming Model

Spark uses the abstract of Resilient Distributed Dataset (RDD) to store and retrieve data in a distributed cluster. An RDD is an immutable data structure that is divided into multiple partitions, and each of them can be stored in memory or on disk across machines. Developers can perform two types of operations on RDDs: *transformation* (e.g., *map*) and *action* (e.g., *count*). Transformations create new RDDs from the existing ones with the specific computation (e.g., user-defined function). Actions return a value (not RDD) by performing a necessary computation (e.g., *count*) on an existing RDD.

A Spark application usually creates RDDs from input data, executes several transformations on RDDs, and performs actions to obtain computation results when necessary [35]. Figure 1a shows a code snippet that contains two jobs: obtaining the total number of words in a text file (Line 4) and taking the first 10 words in the file (Line 5). In this example, we create a new RDD *data* by loading a text file from HDFS (Line 1). Then, we split *data* into *words* by using transformation *flatMap* (Line 2). To get the total number of words, we perform the action *count* on *words* (Line 4). To obtain the first 10 words, we perform the action *take* on *words* (Line 5).

Lineage graph: In Spark, all transformations are lazy. When we perform transformations on RDDs, Spark does not immediately perform the transformation computation. Instead, Spark logs all computation dependencies between RDDs in a DAG (Directed Acyclic Graph), which is referred to as lineage graph. Spark keeps on building this lineage graph until an action operation is applied on the last RDD. After that, Spark generates a job to execute each transformation in the lineage graph.

For example, in Figure 1a, *sc.textFile*() (Line 1) and *data.flatMap*() (Line 2) are not executed by Spark immediately. They will only get executed once we perform an action on RDD *words*, i.e., *words.count*() (Line 4). All the transformations in Figure 1a are included in a lineage graph as shown in Figure 1b. The action operation triggers

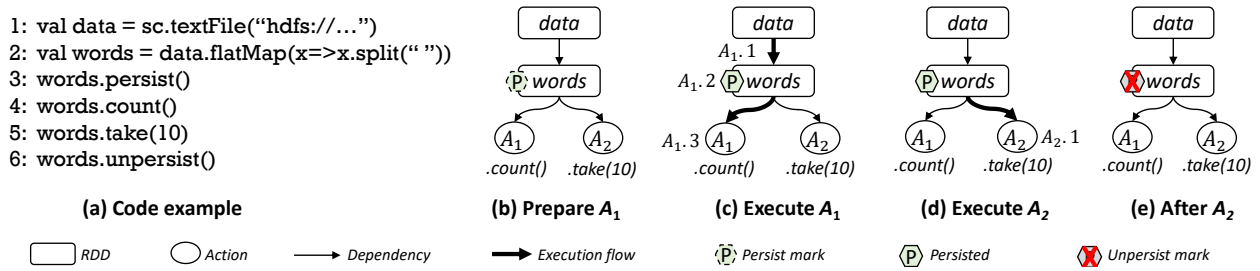

Figure 1: The execution process and cache mechanism in Spark.

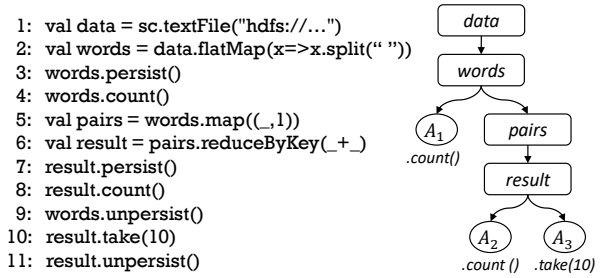

Figure 2: Word count example. The left side shows its code, and the right side shows its lineage graph.

the application execution. In other words, the created DAG will be committed to the cluster and the application begins to execute.

2.2 Cache Mechanism in Spark

In the process of an action execution, the transformations that the action depends on are actually computed and the transformed RDDs are materialized. Once the execution completes, these interim RDDs will be released by the Spark system. Therefore, each transformed RDD should be recomputed each time when an action runs on it. As shown in Figure 1a, RDD *data* and *words* are used twice by two actions: *words.count*() (Line 4) and *words.take*(10) (Line 5). Since each action generates a job, these two RDDs will be computed twice, thus increasing the execution time.

To speed up applications that use certain RDDs multiple times, Spark allows developers to cache data for reuse. By using cache-related APIs, these interim RDDs can be kept in memory (by default) or on disk. So, whenever we invoke another action on the cached RDD, no recomputation takes place. There are two APIs for caching an RDD: *cache*() and $persist(level : StorageLevel)$. The difference between them is that *cache*() persists the RDD into memory, whereas $persist(level)$ can persist the RDD in memory or on disk according to the caching strategy specified by *level*. For easy presentation, we consistently use *persist*() to represent these two APIs in this paper. Freeing up space from the memory is performed by API *unpersist*(). In Figure 1a, *words* is cached by calling $persist()$ on it (Line 3). After the execution of *words.count*() (Line 4), the data of RDD *words* is stored in memory. When executing *words.take*(10) (Line 5), we do not need to recompute *data* and *words* anymore. When *words* is no longer needed, we remove the cached RDD from memory immediately by invoking $unpersist()$ on it (Line 6).

2.3 Spark Execution Semantics

As discussed earlier, when we call a transformation, only the lineage graph is built. When we perform a cache-related API on an reusable RDD, Spark only marks that the RDD should be persisted. The actual caching process takes place when the first action is executed on the RDD. When the next action is executed, Spark finds that the RDD has been cached and thus does not need to recompute it. If this reusable RDD is not cached, recomputation on this RDD takes place and all the RDDs it depends on may be recomputed, too.

Figure 1b shows the dependency relationships between RDDs used by action A_1 (Line 4 in Figure 1a) and action A_2 (Line 5 in Figure 1a). In Figure 1b, *words* is marked as persisted (Line 3 in Figure 1a). When A_1 is triggered, it tries to find all involved RDDs in the lineage graph from bottom to top. Then the actual execution is performed at the sequence of $A_1.1$, $A_1.2$ and $A_1.3$ in Figure 1c. Since *words* is marked as persisted, after *words* is computed, *words* is stored in the memory ($A_1.2$). When A_2 is triggered, it tries to find all involved RDDs first. Since *words* is already persisted, there is no need to find involved RDDs that *words* depends on. The actual execution of action A_2 starts from *words* ($A_2.1$ in Figure 1d). After the execution of action A_2, *words* is freed by Line 6 in Figure 1a, as shown in Figure 1e.

3 CACHE-RELATED BUG PATTERNS

In Spark, which RDD should be persisted and when the RDD is persisted / unpersisted are totally decided by developers. However, developers may misunderstand their Spark applications and the caching mechanisms, and introduce cache-related bugs. To combat cache-related bugs, we thoroughly investigate the Spark programming model and cache mechanism. Further, we enumerate all possible ways to utilize cache-related APIs, and evaluate their performance impacts. All cases that can slowdown Spark applications are considered as cache-related bug patterns. Finally, we summarize six cache-related bug patterns.

In this section, we use the word count example shown in Figure 2 to illustrate cache-related bug patterns. RDD *data* is created by loading a text file from HDFS (Line 1), and further split into *words* (Line 2) by white spaces. We obtain the number of words by performing action *count* on *words* (Line 4). Transformation *map* maps each word in *words* with 1 to create a new key-value RDD *pairs* (Line 5), and transformation *reduceByKey* further shuffles *pairs* based on the key and obtains RDD *result* that contains the number for each particular word (Line 6). We further obtain the total number of unique words in the file by performing action *count* on *result* (Line 8), and

```
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3:+words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: val result = pairs.reduceByKey(_+_)
7: result.persist()
8: result.count()
9:+words.unpersist()
10: result.take(10)
11: result.unpersist()
```

(a) Missing persist

```
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3:+words.persist()
4: words.count()
5:- words.persist()
6: val pairs = words.map((_,1))
7: val result = pairs.reduceByKey(_+_)
8: result.persist()
9: result.count()
10: words.unpersist()
11: result.take(10)
12: result.unpersist()
```

(b) Lagging persist

```
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6:- pairs.persist()
7: val result = pairs.reduceByKey(_+_)
8: result.persist()
9: result.count()
10: words.unpersist()
11: result.take(10)
12:- pairs.unpersist()
13: result.unpersist()
```

(c) Unnecessary persist

```
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: val result = pairs.reduceByKey(_+_)
7: result.persist()
8: result.count()
9:+words.unpersist()
10: result.take(10)
11: result.unpersist()
```

(d) Missing unpersist

```
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6:- words.unpersist()
7: val result = pairs.reduceByKey(_+_)
8: result.persist()
9: result.count()
10:+words.unpersist()
11: result.take(10)
12: result.unpersist()
```

(e) Premature unpersist

```
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: val result = pairs.reduceByKey(_+_)
7: result.persist()
8: result.count()
9:+words.unpersist()
10: result.take(10)
11: result.unpersist()
12: -words.unpersist()
```

(f) Lagging unpersist

Figure 3: Cache-related bug examples and corresponding fix methods for the word count example in Figure 2.

obtain the first 10 records in *result* by performing action *take*(10) on *result* (Line 10). Note that, *words* and *result* are persisted (Line 3 and Line 7). They are correct decisions for us. Then, we use six bug examples in Figure 3 to illustrate why other persist decisions are incorrect and how they can degrade performance.

3.1 Missing Persist

A transformed RDD in an action will be released by default when the action completes. If an RDD is used by the following actions, it will be recomputed multiple times, thus degrading the execution performance. To improve the execution performance, developers should persist these RDDs that can be used multiple times.

Figure 3a introduces a missing persist bug. In Figure 3a, RDD *data* and *words* are used by two actions *words.count*() (Line 4) and *result.count*() (Line 8). If *words* is not persisted, *data* and *words* will be computed twice. We do not need to persist *data* if *words* is persisted, because if *words* is persisted, *data* will not be recomputed. Therefore, the best caching decision is to persist *words* before the first action (i.e., *words.count*()) that depends on *words* is triggered. This bug slows down the example by 21.4%. For another real-world example, the missing persist bug SPARK-16697 [13] in MLlib [23] slows down its application by 51.6% (More details about performance slowdown can be found in Table 1).

3.2 Lagging Persist

As discussed earlier, an RDD is only marked as persisted when the *persist*() API is invoked. The actual caching process takes place when the RDD is materialized by an action. Therefore, for an RDD that needs to be persisted, the *persist*() API must be called on the RDD before the first action that depends on it is triggered. Otherwise, the RDD cannot be persisted as expected.

Figure 3b introduces a lagging persist bug. The *persist*() operation on RDD *words* (Line 5) is invoked after action *words.count*() and before action *result.count*() (Line 9). When action *words.count*() at Line 4 is executed, *words* is not marked as persisted, and will not be persisted. When action *result.count*() is executed, *data* and *words* are computed again, since *words* is not cached yet. In this example, lagging persist of *words* causes recomputation of *data* and *words* in action *result.count*(). This bug slows down the example by 22.9%.

3.3 Unnecessary Persist

Cached RDDs usually occupy large precious memory. Therefore, if an RDD is not reused by multiple actions, it should not be persisted. Unnecessary persist can occupy extra memory, and affects the application execution, thus causing performance degradation.

Figure 3c introduces an unnecessary persist bug on RDD *pairs*, which is persisted at Line 6. Since RDD *result* has been persisted after the execution of action *result.count*() (Line 9). The following action *result.take*() will use cached *result* rather than *pairs*. Thus, it is unnecessary to persist *pairs* in this example. This bug slows down the example by 4.5%. The real-world unnecessary persist bug SPARK-18608 [16] slows down its application by 23.3%.

3.4 Missing Unpersist

If an RDD is persisted but not unpersisted when it will not be used anymore, the RDD will be kept in memory until the application

completes or it may be replaced by other cached RDDs. Thus, the unpersisted RDD can occupy precious memory, and affects the application execution, thus causing performance degradation.

Figure 3d introduces a missing unpersist bug. RDD *words* stays in memory until the application completes. In fact, after action *result.count*() at Line 8, *words* will not be used anymore. Thus, it should be unpersisted at Line 9. This bug slows down the example by 0.9%. The real-world missing unpersist bug SPARK-3918 [6] in MLlib [23] causes an OOM in its application.

3.5 Premature Unpersist

A cached RDD should be unpersisted until it will not be used anymore. Premature unpersist of a cached RDD will invalidate the cached RDD, and cause its recomputation, thus slowing down the application.

Figure 3e introduces a premature unpersist bug. RDD *words* is unpersisted after action *words.count*() and before action *result.count*() at Line 6. Thus, action *result.count*() at Line 9 has to recompute *words*. Only after executing action *result.count*(), RDD *words* will not be used anymore and can be unpersisted (Line 10). This bug slows down the example by 35.7%.

3.6 Lagging Unpersist

Once a cached RDD will not be used anymore, it should be unpersisted right away. Otherwise, the cached RDD still occupies precious memory, and affects the application execution, thus causing performance degradation.

Figure 3 introduces a lagging unpersist bug. After executing action *result.count*(), RDD *result* is persisted and RDD *words* will not be used anymore. So, RDD *words* should be unpersisted right after action *result.count*() (Line 9). However, RDD *words* is kept in memory until it is unpersisted at Line 12. This bug slows down the example by 0.3%.

4 DETECTING CACHE-RELATED BUGS

Given a Spark application, CacheCheck works in three steps. First, after running the application, CacheCheck collects its actual execution trace about its lineage graph, actions and caching decisions (Section 4.1). Second, from collected lineage graph and actions, CacheCheck infers the correct caching decisions based on the execution semantics of Spark (Section 4.2). Third, CacheCheck analyzes the actual and correct caching decisions, and identifies the difference, which usually indicates cache-related bugs (Section 4.3).

4.1 Execution Trace

The execution of a Spark application consists of a number of actions. In order to detect cache-related bugs, we need to record which RDDs are persisted, and when they are persisted and unpersisted. Thus, we only consider three kinds of operations in the execution trace of a Spark application. They are listed as follows.

- *action*: We use *rdd.action()* to denote that an *action* is performed on *rrd*.
- *persist*: We use *rdd.persist*() to denote that a *persist* operation is performed on *rdd*.
- *unpersist*: We use *rdd.unpersist*() to denote that an *unpersist* operation is performed on *rdd*.

An execution trace of a Spark execution can be represented as follows.

$$\tau = (action|persist|unpersist)*$$

In the execution trace, each action contains its partial lineage graph that it directly depends on. For example, in Figure 2, action A_1 contains its partial lineage graph $G_1 = data \rightarrow words \rightarrow A_1$, and action A_2 contains its partial lineage graph $G_2 = data \rightarrow words \rightarrow pairs \rightarrow result \rightarrow A_2$, and action A_3 contains its partial lineage graph $G_3 = data \rightarrow words \rightarrow pairs \rightarrow result \rightarrow A_3$, Note that, Spark generate a unique ID for each RDD. In the execution trace, the recorded lineage graph is built on these unique IDs. For easy presentation and understanding, we use variable names instead of these unique IDs in the lineage graphs.

We can further combine the partial lineage graphs of all actions to generate the whole lineage graph G. The whole lineage graph generation algorithm is straightforward. For each action A_i, we inspect each edge e in its lineage graph G_i. If e does not exist in G, we merge edge e and its corresponding nodes into G. For example, we can combine G_1, G_2, and G_3, and generate the lineage graph in the right side of Figure 2.

Note that, the execution trace precisely reflects the caching decisions for the Spark application, e.g., which RDDs are persisted, and when they are persisted and unpersisted. Take Figure 2 as an example. After running this application, we can obtain an execution trace as follows: {*words.persist*(), *words.count*(), *result.persist*(), *result.count*(), *words.unpersist*(), *result.take*(10), *result.unpersist*()}. From this trace, we can see that *words* is persisted before action A_1 (i.e., *words.count*()), and *words* is unpersisted after action A_2 (i.e., *result.count*()).

We collect the execution trace using dynamic method by slightly instrumenting Spark's code. The details are shown in Section 5. Thus, complex code structures, e.g., loops and recursions, will be unfolded as a sequential trace in this step.

4.2 Correct Caching Decision Generation

To judge whether the caching decisions in a given actual execution trace τ_a are correct or not, we construct the correct execution trace τ_c, which reflects the correct caching decisions. In order to do so, we extract all actions in the actual execution trace τ_a, and ignore all caching decisions in τ_a. Then, we construct the correct execution trace τ_c in three steps. First, we identify all RDDs that need to be persisted (Section 4.2.1). Second, we identify the correct locations for persisting these RDDs (Section 4.2.2). Third, we identify the correct locations for unpersisting these RDDs (Section 4.2.3).

4.2.1 Identify RDDs That Need to be Persisted. As discussed in Section 3.1, if an RDD is used by multiple actions, the RDD needs to be persisted. In the lineage graph, the reused RDD is depended by multiple actions. For example, in Figure 2, *words* is used by action A_1 and A_2. Thus, *words* should be persisted. However, not all RDDs that are depended by multiple actions should be persisted. In Figure 2, *pairs* is depended by action A_2 and A_3. If we persist *pairs* and do not persist *result*, RRD *result* will be computed twice by A_2 and A_3, respectively. If we persist *pairs* and *result*, the cached *pairs* will not be reused, and cause an unnecessary persist (Section 3.3). Therefore, CacheCheck needs to identify optimal decisions about what RDDs should be persisted.

Algorithm 1: Identify RDDs that need to be persisted.

```
   Input: actions (Executed action list)
   Output: cachedRDDs
1  cachedRDDs ← ∅;
2  foreach a1, a2 ∈ actions do
3  |  identifyCachedRDDs(a1.rdd, a2);
4  end
5
6  Function identifyCachedRDDs(rdd, a2) do
7  |  if a2.dependsOn(rdd) then
8  |  |  cachedRDDs.add(rdd);
9  |  else
10 |  |  foreach pRDD ∈ rdd.parentRDDs do
11 |  |  |  identifyCachedRDDs(pRDD, a2);
12 |  |  end
13 end
```

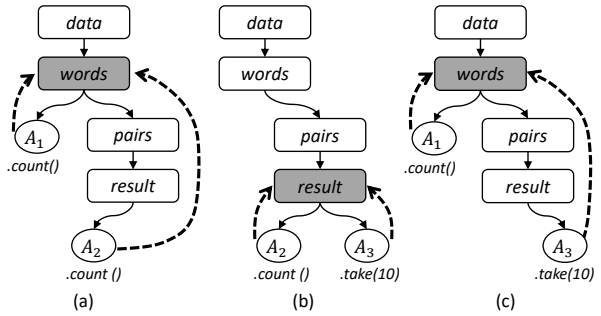

Figure 4: Identify cached RDDs for the example in Figure 2.

Algorithm 1 shows how to identify RDDs that need to be persisted for an execution trace. We aim to find the minimal set of RDDs, if they are persisted, all RDD recomputation will be eliminated. For every two actions a_1 and a_2 in the executed action list *actions*, CacheCheck finds the closest RDDs that are depended by both a_1 and a_2 (Line 2–4). CacheCheck starts from the RDD that a_1 directly depends on ($a_1.rdd$ at Line 3), and checks whether the RDD is used by action a_2. If a_2 also depends on *rdd*, then *rdd* is one of the closest RDDs that are used by two actions, and *rdd* needs to be persisted (Line 7–8). If a_2 does not depend on *rdd*, CacheCheck further checks whether the parent RDDs of *rdd* are depended by a_2 (Line 10–12).

Take Figure 2 as an example. This example contains three actions: *words.count*(), *result.count*() and *result.take*(10). The cached RDD identification process is shown in Figure 4. For action *words.count*() and *result.count*(), the closest RDD shared by them is *words* as shown in Figure 4a. For action *result.count*() and *result.take*(10), the closest RDD shared by them is *result* as shown in Figure 4b. For action *words.count*() and *result.take*(10), the closest RDD shared by them is *words* as shown in Figure 4c. Therefore, *words* and *result* need to be persisted.

4.2.2 Identify Persist Locations. After finding all the RDDs that should be persisted in Algorithm 1, we need to identify when to persist these RDDs. Algorithm 2 presents how to identify when to persist an RDD. First, we construct an initial execution trace by adding all actions into it (Line 2). For the example shown in Figure 2, the initial execution trace is {*words.count*(), *result.count*(), *result.take*(10)}.

For each RDD *rdd* in *cachedRDDs*, the operation *rdd.persist*() should be inserted before the first action that depends on *rdd* (Line

Algorithm 2: Generate correct caching decisions.

```
   Input: actions (Executed action list), cachedRDDs
   Output: τc (Correct trace)
1  τc ← ∅;
2  τc.addAll(actions);
3  /* Identify persist locations */
4  foreach rdd ∈ cachedRDDs do
5  |  for i ← 1; i ≤ actions.length; i++ do
6  |  |  if actions[i].dependsOn(rdd) then
7  |  |  |  τc.insertBefore(actions[i], rdd.persist());
8  |  |  |  break;
9  |  end
10 end
11 /* Identify unpersist locations */
12 foreach rdd ∈ cachedRDDs do
13 |  lastAction ← NULL;
14 |  for i ← 1; i ≤ actions.length; i++ do
15 |  |  if use(action[i], actions[i].rdd, rdd) then
16 |  |  |  lastAction ← actions[i];
17 |  end
18 |  τc.insertAfter(action[i], rdd.unpersist());
19 end
20
21 /* Does curRDD use tarRDD? */
22 Function use(action, curRDD, tarRDD) do
23 |  if curRdd = tarRDD then
24 |  |  return True;
25 |  else if isCached(action, curRDD) = False then
26 |  |  foreach pRDD ∈ curRDD.parentRDDs do
27 |  |  |  if use(action, pRDD, tarRDD) then
28 |  |  |  |  return True;
29 |  |  end
30 |  return False;
31 end
32
33 /* Has rdd been cached before executing action? */
34 Function isCached(action, rdd) do
35 |  if cachedRDDs.contain(rdd) = False then
36 |  |  return False
37 |  preAction ← τc.previousAction(action);
38 |  if preAction ≠ NULL then
39 |  |  if τc.isBefore(rdd.persist(), preAction) then
40 |  |  |  return True;
41 |  return False;
42 end
```

Algorithm 3: Detect cache-related bugs.

```
   Input: τ_c (Correct trace), τ_a (Actual trace)
1  for i ← 1; i ≤ τ_a.length; i++ do
2      op ← τ_c[i];
3      if op ∉ τ_a then
4          if op.isPersist then
5              Report 'missing persist on op.rdd';
6          if op.isUnpersist then
7              if !op.isAfter(τ_c.lastAction) ∧
                τ_a.havePersist(op.rdd) then
8                  Report 'missing unpersist on op.rdd';
9      else
10         if op.isPersist then
11             cAction ← τ_c.nextAction(op);
12             aAction ← τ_a.nextAction(op);
13             if cAction.isBefore(aAction) then
14                 Report 'lagging persist on op.rdd';
15         else if op.isUnpersist then
16             cAction ← τ_c.previousAction(op);
17             aAction ← τ_a.previousAction(op);
18             if cAction.isBefore(aAction) then
19                 Report 'lagging unpersist on op.rdd';
20             else if aAction.isBefore(cAction) then
21                 Report 'premature unpersist on op.rdd';
22 end
23 for i ← 1; i ≤ τ_a.length; i++ do
24     op ← τ_a[i];
25     if op ∉ τ_c then
26         if op.isPersist then
27             Report 'unnecessary persist on op.rdd';
28 end
```

4–10). For the example shown in Figure 2, *words* and *result* should be cached. For RDD *words*, *words.count*() is the first action that depends on it. For RDD *result*, *result.count*() is the first action that depends on it. Therefore the new trace that contains the RDD persist locations is: {*words.persist*(), *words.count*(), *result.persist*(), *result.count*(), *result.take*(10)}.

4.2.3 Identify Unpersist Locations. If a cached RDD is not used any more by following actions, it should be unpersisted as early as possible to save memory. That is to say, once the last action that uses a cached RDD has been executed, the RDD should be discarded. First, we find the last action which uses a cached RDD (Line 13–17). Then, we insert *rdd.unpersist*() right after the last action (Line 18). To decide if an *action* uses an RDD *rdd*, CacheCheck starts to check if the RDD *curRDD* that *action* directly depends on uses *rdd* (Line 15). If *curRDD* is not cached before, then CacheCheck checks whether its parent RDDs use *tarRDD* (Line 25–29). If *curRDD* has been cached (Line 34–42), then CacheCheck will stop search, since *action* will use cached *curRDD*, and does not use its parent RDDs. Take Figure 2 as an example. the last action that uses *words* is *result.count*() and the last action that uses *result* is *result.take*(10). Note that, once *result* is persisted by action A_2, action A_3 will use the cached *result*, and will not use *words* any more. Therefore, the new correct trace is: {*words.persist*(), *words.count*(), *result.persist*(), *result.count*(), *words.unpersist*(), *result.take*(10), *result.unpersist*()}.

4.3 Bug Detection

CacheCheck detects cache-related bugs by analyzing the difference of caching decisions in the collected actual execution trace (τ_a) and the generated correct execution trace (τ_c) in Section 4.2. For all the buggy examples in Figure 3, CacheCheck can generate the same correct execution trace: {*words.persist*(), *words.count*(), *result.persist*(), *result.count*(), *words.unpersist*(), *result.take*(10), *result.unpersist*()}. However, their actual execution traces are different, and illustrate different cache-related bugs. Algorithm 3 shows our cache-related bug detection. We illustrate it as follows.

(1) Missing persist. If an operation *rdd.persist*() belongs to the correct execution trace but does not belong to the actual execution trace, CacheCheck reports a missing persist bug (Line 3–5).

(2) Lagging persist. Both the correct execution trace and the actual execution trace contains an operation *rdd.persist*(). However, in the actual execution trace, *rdd.persist*() is not performed before the first action that uses *rdd* (Line 10–14). Take Figure 3b as an example. We can get the actual execution trace as {*words.count*(), *words.persist*(), *result.persist*(), *result.count*(), *words.unpersist*(), *result.take*(10), *result.unpersist*()}. We can see that *words.persist*() is located before *words.count*() in the correct trace. However, it is located after *words.count*() in the actual execution trace. So, CacheCheck reports a lagging persist bug on RDD *words*.

(3) Unnecessary persist. If an operation *rdd.persist*() belongs to the actual execution trace but does not belong to the correct execution trace, CacheCheck reports an unnecessary persist bug (Line 25–27).

(4) Missing unpersist. If an RDD *rdd* that should be cached has been persisted in the actual execution trace, but the operation *rdd.unpersist*() belongs to the correct execution trace but not belong to the actual execution trace, and the operation is not located after the last action in the trace, CacheCheck reports a missing unpersist bug (Line 6–8).

(5) Premature unpersist. Both the correct execution trace and the actual execution trace contain an operation *rdd.unpersist*(). However, in the actual execution trace, *rdd.unpersist*() is performed before the last action that uses *rdd* (Line 20–21). Take Figure 3e as an example. The actual execution trace of this example is {*words.persist*(), *words.count*(), *words.unpersist*(), *result.persist*(), *result.count*(), *result.take*(10), *result.unpersist*()}. *words.unpersist*() is located after *result.count*() in the correct execution trace, while it is located before *result.count*() in the actual trace. Thus, CacheCheck reports a premature unpersist bug on RDD *words*.

(6) Lagging unpersist. Both the correct execution trace and the actual execution trace contain an operation *rdd.unpersist*(). However, in the actual execution trace, *rdd.unpersist*() is not performed right after the last action that uses *rdd* (Line 15–19).

5 IMPLEMENTATION

CacheCheck first collects an execution trace by modifying Spark implementation, and then performs cache-related bug detection offline

on the execution trace. Here, we mainly introduce the implementation details which are not described in Section 4, including code instrumentation for trace collection, and bug reports for helping inspect bugs and bug deduplication.

Execution trace collection. Spark invokes *SparkContext.runJob*() to execute each action in a Spark application. We modify *SparkContext.runJob*(), and record the information about each action, including the action ID, the RDD that the action works on, and the lineage graph of the action. To obtain each cache-related operation, we modify *RDD.persist*() and *RDD.unpersist*(), and record the RDD ID for each *persist*() and *unpersist*() operation. Our modification to the Spark system is quite lightweight, and involves only 338 lines of code. The overhead for obtaining execution traces is usually negligible (less than 5% in our experiments).

Note that, we modify the Spark system to collect execution traces of Spark applications. Therefore, developers only need to run their Spark applications on our modified Spark system, without instrumenting or modifying their applications in any way. CacheCheck can automatically collect their execution traces and perform offline analysis for cache-related bug detection.

Bug reports. CacheCheck detects cache-related bugs based on the execution trace. If a cache-related bug at the same code location is executed multiple times, CacheCheck can report the cache-related bug multiple times. In order to eliminate duplicate bug reports, we consider two bug reports are the same if they satisfy the following two conditions: (1) Their bug-related RDDs are generated in the same code locations. (2) They have the same bug information, e.g., bug pattern and code locations of cache-related API invocations.

To facilitate bug inspection for developers, we provide the following information for each reported bug: (1) The code positions and call stacks for each operation that relates to the bug report, e.g, where a related RDD is generated, where a cache-related API is called on an RDD, and invocation position for each action. (2) The lineage graph for each action that relates to the bug report. For example, for a missing persist on *rdd*, CacheCheck provides where two actions use *rdd*, and which action firstly uses *rdd*. For a premature unpersist on *rdd*, CacheCheck provides where the *unpersist*() is invoked on *rdd*, and actions that use *rdd* after the *unpersist*() operation.

6 EVALUATION

We evaluate CacheCheck and study the following three research questions:

RQ1: *Can cache-related bugs seriously affect the performance of Spark applications?*

RQ2: *Can CacheCheck effectively detect known cache-related bugs in Spark applications?*

RQ3: *Can CacheCheck detect unknown cache-related bugs in real-world Spark applications?*

To answer RQ1 and RQ2, we first build a cache-related bug benchmark with 18 bugs (Section 6.1). To answer RQ1, we compare the performance difference between the buggy versions and their corresponding fixed versions (Section 6.2). To answer RQ2, we evaluate CacheCheck on our bug benchmark (Section 6.3). To answer RQ3, we evaluate CacheCheck on six real-world Spark applications to validate if CacheCheck can detect new bugs (Section 6.4).

6.1 Cache-Related Bug Benchmark

We select the six bug examples in Figure 3 as our micro-benchmark (P1–P6 in Table 1). We further manually inspect issue reports in open source Spark applications, to collect real-world cache-related bugs. We select three representative Spark applications, i.e., GraphX [22], MLlib [23] and Spark SQL [29], as our study applications. These applications are representative official Spark applications, well-maintained, and widely used in practice. All issue reports in these applications are publicly available in Spark JIRA [28].

We use cache-related keywords, i.e., cache and persist, to search all issue reports in GraphX, MLlib and Spark SQL. Two authors and two master students carefully analyze each returned issue report. If an issue report satisfies the following conditions, we select it into our benchmark. (1) We can identify its root cause, and confirm that it is a cache-related bug. (2) We can find its fixing patch, and obtain its buggy version and fixed version. (3) We can reproduce it based on an existing test case in its application.

Through the above process, we finally obtain 12 cache-related bugs (B1–B12 in Table 1). Note that, these 12 cache-related bugs only cover three bug patterns discussed in Section 3, i.e., 3 missing persist bugs, 3 unnecessary persist bugs, and 6 missing unpersist bugs. We do not observe cache-related bugs for other three bug patterns, i.e., lagging persist, premature unpersist and lagging unpersist. However, as discussed in Section 6.4, we detect 15 bugs for these three bug patterns. Among them, 11 have been confirmed by developers, and 7 have been fixed. This indicates that developers may not have realized these three bug patterns before our study.

6.2 Performance Slowdown Caused by Cache-Related Bugs

Experimental design. We use the 18 cache-related bugs in our bug benchmark in Table 1 as our experimental subjects. We run this experiment in the local mode with 8GB memory. For the six cache-related bugs (P1–P6), we use a 660MB text file as input. For the 12 real-world cache-related bugs (B1–B12), we run their corresponding test cases with inputs of 23.4–708MB. For each cache-related bug, we run its buggy version and fixed version 10 times, and then use the average execution time to compare the performance slowdown.

Experimental result. The columns 5–7 in Table 1 (Execution time) show the absolute execution time of the buggy / fixed versions and the performance slowdown. We can see that cache-related bugs can cause non-negligible (0.1% – 51.6%) performance slowdown. Eight cache-related bugs introduce more than 5% performance slowdown: P1, P2, P5, B1, B2, B3, B5, B8. Specially, in bug B2, all the cached RDDs in a loop are not unpersisted, and waste huge memory, thus causing Out of Memory (OOM). After correctly unpersisting all RDDs in the loop, the application terminates normally.

Generally, missing persist (e.g., P1, B5), lagging persist (e.g., P2), and premature unpersist (e.g., P5) can usually cause much performance slowdown, since they can cause recomputation of related RDDs. However, not all missing persist bugs can cause serious performance slowdown, e.g., B6 and B7. This is because recomputing the unpersisted RDDs in these two cases is not time-consuming. Missing unpersist (e.g., B2, B3) and unnecessary persist (e.g., B8) can cause much performance slowdown sometimes, if they occupy too much memory.

Table 1: Experimental Results on Known Bugs

ID	Issue ID	App	Pattern	Execution time			Detected	New bugs
				Buggy(s)	Fixed(s)	Slowdown		
P1	Pattern-MP (Figure 3a)	Word count	Missing persist	31.7	26.1	21.4%	√	0
P2	Pattern-LP (Figure 3b)	Word count	Lagging persist	32.1	26.1	22.9%	√	0
P3	Pattern-UP (Figure 3c)	Word count	Unnecessary persist	27.3	26.1	4.5%	√	0
P4	Pattern-MUP (Figure 3d)	Word count	Missing unpersist	26.3	26.1	0.9%	√	0
P5	Pattern-PUP (Figure 3e)	Word count	Premature unpersist	35.4	26.1	35.7%	√	0
P6	Pattern-LUP (Figure 3f)	Word count	Lagging unpersist	26.2	26.1	0.3%	√	0
B1	SPARK-1266 [3]	MLlib	Unnecessary persist	24.7	23.4	5.2%	√	4
B2	SPARK-3918 [6]	MLlib	Missing unpersist	OOM	27.2	-	√	0
B3	SPARK-7100 [10]	MLlib	Missing unpersist	27.6	26.1	5.9%	√	0
B4	SPARK-10182 [9]	MLlib	Missing unpersist	34.2	33.6	1.8%	√	2
B5	SPARK-16697 [13]	MLlib	Missing persist	67.8	44.7	51.6%	√	1
B6	SPARK-16880 [14]	MLlib	Missing persist	262.2	256.3	2.3%	√	1
B7	SPARK-18356 [15]	MLlib	Missing persist	20.9	20.7	0.7%	√	2
B8	SPARK-18608 [16]	MLlib	Unnecessary persist	244.9	198.7	23.3%	√	3
B9	SPARK-26006 [20]	MLlib	Missing unpersist	56.0	54.8	2.0%	√	2
B10	SPARK-2661 [4]	GraphX	Missing unpersist	13.7	13.1	4.6%	√	0
B11	SPARK-3290 [5]	GraphX	Missing unpersist	11.9	11.7	1.8%	√	8
B12	SPARK-7116 [11]	Spark SQL	Unnecessary persist	20.4	20.2	0.1%	√	0

6.3 Detecting Known Cache-Related Bugs

We use the 18 cache-related bugs in Table 1 to evaluate the cache-related bug detection ability of CacheCheck. For each cache-related bug in Table 1, we run its test case, and use CacheCheck to detect cache-related bugs on the collected execution trace.

The last two columns in Table 1 show the detection result. We can see that CacheCheck can detect all 18 cache-related bugs (Detected). CacheCheck further detects 23 new cache-related bugs (New bugs) in these test cases. Two authors and two master students manually inspect these new detected cache-related bugs to validate whether they are real bugs. After manual validation, we find that all these newly detected cache-related bugs are true. Since these new cache-related bugs are detected on old versions of GraphX, MLLib and Spark SQL, we do not submit them to the developers for further confirmation.

6.4 Detecting Unknown Cache-Related Bugs

In this section, we evaluate CacheCheck on six real-world Spark applications, and obtain developers' feedbacks about our detected cache-related bugs.

6.4.1 Experimental Design. We collect six Spark applications for our experiments from two aspects. First, we use GraphX [22], MLlib [23], Spark SQL [29], which are the three official Spark applications, and well maintained by the Spark community. Second, we collect another three Spark applications from GitHub by the following steps. (1) We used the keyword "Apache Spark" to search GitHub repositories written in Scala, and obtained around 1600 repositories. (2) We sorted these repositories by their stargazers in descending order, and manually checked them one by one. (3) We aimed to select representative data process applications. Thus, If a repository is used for examples, learning, and extensions of Spark framework, we ignored it. (4) If a repository contains test cases, and we can run them on Spark 2.4.3, then we selected it. By following the above steps, we obtained three applications and evaluated CacheCheck on them. The first four columns in Table 2 show the basic information about all these six Spark applications.

We perform our experiments as follows. First, we run the test cases included in these applications on our modified Spark implementation (Section 5) and collect their execution traces. The fifth and sixth columns in Table 2 show the numbers of collected actions and RDDs in the execution traces. Second, we use CacheCheck to detect cache-related bugs based on these execution traces.

In our experiment, a cache-related bug may be triggered multiple times by different test cases. We adopt the approach discussed in Section 5 to remove duplicated bug reports. When counting bugs, we only count unique bugs, i.e., remove duplicated bug reports from different test cases. MCL [8] and t-SNE [7] are built on MLlib. For these two applications, if we detect a bug occurring in MLlib, we only count it once in MLlib, and do not count it in these applications.

6.4.2 Detection Result. Table 2 shows the unique cache-related bugs detected by CacheCheck. In total, CacheCheck detects 72 cache-related bugs, covering the six cache-related bug patterns in Section 3. In detail, CacheCheck detects 34 missing persist bugs (MP), 1 lagging unpersist bug (LP), 18 unnecessary persist bugs (UP), 5 missing unpersist bugs (MUP), 12 lagging unpersist bugs (LUP), and 2 premature unpersist bugs (PUP). We can see that cache-related bugs are common in real-world Spark applications.

We have submitted all these 72 bugs to developers through JIRA and GitHub. So far, 58 bugs have been discussed by developers, and 53 of them have been confirmed as real bugs. The remaining 5 bugs are not decided yet. Developers have fixed 28 bugs. Among the 14 bugs that have not received responses, 9 occur in third-party

Table 2: Detecting Cache-Related Bugs on Real-World Spark Applications

App	Description	Star	LOC	Action	RDD	MP	LP	UP	MUP	LUP	PUP
GraphX [22]	Graph computation	-	32,708	11	236	1(1)	0	6(5)	1(1)	0	0
MLlib [23]	Machine learning	-	109,786	888	4,017	24(21*)	1(1)	10(10)	2(1)	12(8)	2(2)
Spark SQL [29]	SQL	-	395,122	69	263	4(3)	0	0	0	0	0
MCL [8]	Markov clustering	32	1,058	48	290	4	0	0	0	0	0
Betweenness [12]	k betweenness centrality	40	488	6	83	0	0	1	0	0	0
t-SNE [7]	Distributed t-SNE	129	798	39	97	1	0	1	2	0	0
Total				1,061	4,986	**34(25)**	**1(1)**	**18(15)**	**5(2)**	**12(8)**	**2(2)**

* The numbers in the parentheses denote the bugs confirmed by developers.

```
1:  userInput.persist();
2:  itemInput.persist();
3:  itemFactor = computeFacotrs(userFactor, itemInput).persist()
4:  itemFactor.count()
5:  userFactor = computeFactors(itemFactor, userInput)
5:  val userOutput = userInput.join(userFactor)
6:  val itemOutput = itemInput.join(itemFactor)
7:  userOutput.count()
8:  itemFactor.unpersist()  // Premature unpersist
9:  itemOutput.count()
10: userInput.unpersist()  // Lagging unpersist
11: itemInput.unpersist()
```

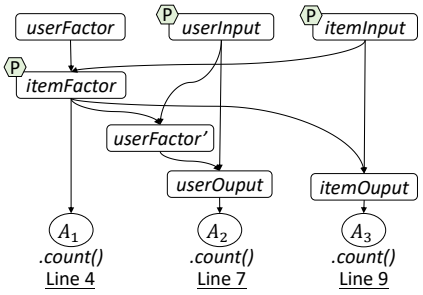

Figure 5: Premature unpersist and lagging unpersist bugs in simplified ALS algorithm in SPARK-29844 [24]. *userFactor'* **denotes a new RDD created in Line 5, which is different from the original RDD** *userFactor* **in Line 3.**

applications on GitHub, e.g., MCL [21], mainly due to the inactive maintenance of these applications.

Note that, we have detected 1 lagging persist bugs, 12 lagging unpersist bugs, and 2 premature unpersist bugs in real-world Spark applications. Among them, 11 have been confirmed by developers, and 7 have been fixed. Although we did not observe these bug patterns when we collecting known cache-related bugs, developers confirm that these bug patterns are harmful.

6.4.3 Feedbacks from Developers. So far, developers have given feedbacks on 58 cache-related bugs. In the process of bug submission, developers have shown great interests in CacheCheck. For example, after they confirmed and resolved one bug [24], they gave the following comments to encourage us to submit more cache-related bugs: "*Nice, I wonder what else CacheCheck will turn up.*"

Besides, we also received some interesting comments from developers, which indicates that even experienced developers may not be easy to figure out complicated caching decisions. For example, Figure 5 shows the simplified code of our bug report SPARK-29844 [24] in MLlib. There are two cache-related bugs in this report. (1) *itemFactor* (Line 8) should be unpersisted after *itemOutput.count*() (Line 9), instead of after *userOutput.count*(Line 7). This causes a premature unpersist bug. (2) *userInput* (Line 10) should be unpersisted before *itemOutput.count*() (Line 9), instead of after *itemOutput.count*() (Line 9). This causes a lagging unpersist bug. In this bug, a developer left the following comment in its PR [25]: "*I don't see the problem here immediately. Doesn't this (itemOutput) depend on some of the cached user RDDs (userInput)?*" The developer thought

```
1: baggedInput.persist()
2: while(treeNode.nonEmpty) {
3:   // findBestSplits will invoke an action
4:   RandomForest.findBestSplits(baggedInput)
5: }
6: baggedInput.unpersist()
```

Figure 6: An unnecessary persist bug when Line 4 in the *while* **statement is executed only once in MLlib [26].**

itemOutput depends on *userInput*, so *userInput* should be unpersisted after all actions completed. After we explained the fact that *itemOutput* only depends on the cached *itemFactor*, he agreed that moving *userInput.unpersist*() before *itemOutput.count*() is a kind of optimization.

Why developers do not fix some cache-related bugs? Among the 53 confirmed cache-related bugs, developers do not fix 25 of them for now. We summarize the reasons why they do not fix them as follows.

(1) Developers thought that the performance slowdown caused by cache-related bugs should be negligible. 17 bugs belong to this case. First, the should-be-persisted RDD is small, and the overhead of recomputation is negligible, e.g., SPARK-29872 [27]. Second, the bugs can only occur in rare cases. For example, Figure 6 shows an unnecessary persist bug in MLLib [26]. If the iteration (Line 2–5) is only executed once, the persist on RDD *baggedInput* is unnecessary. Developers thought this rarely occurs in the production environment. Third, the bugs occur in illustrative code (i.e., code

```
1: def apply(vertices, edges): Graph[V, E] = {
2:   vertices.persist()
3:   val newEdges = edges.map(part => part.withoutVertex).persist()
4:   new Graph(vertices, newEdges)
5: }
```

Figure 7: An API related to cache decisions in GraphX [17].

examples), and developers thought fixing the bugs can make code understanding difficult, e.g., SPARK-29872 [27].

(2) Developers thought that fixing cache-related bugs makes APIs difficult to utilize. 8 bugs belong to this case. Spark applications, e.g., GraphX, MLLib and Spark SQL, can be used as APIs to build other applications. The caching decisions of these APIs may not consider all their usage scenarios. Figure 7 shows an unnecessary persist bug in GraphX [17]. In API *apply*(), RDD *vertices* and *newEdges* are persisted. This API can be invoked by many graph processing algorithms in GraphX, e.g., SVDPlusPlus and PageRank. Both *vertices* and *newEdges* are used by only one action in SVDPlusPlus, so CacheCheck reports two unnecessary persist bugs on them. However, in PageRank, they are used by multiple actions in an iterative computation, so the persist decisions are correct. Developers wanted to simplify the usage of these APIs, and would not like to fix them for different applications.

6.4.4 Slowdowns of Detected Unknown Bugs. The slowdowns of cache-related bugs are usually highly related to the size of input data in the Spark applications. We detected unknown cache-related bugs by running test cases included in the applications. Note that, these test cases usually run on small amount of data (e.g., a string with 100 characters, a graph with 10 nodes), and complete quickly (e.g., in less than 1 second). In these scenarios, the slowdowns can be ignored. Therefore, we did not report the slowdowns for these test cases. However, for real scenarios, these test cases will be run on large amount of data (e.g., 100G) for a long time (e.g., 30 minutes). In these scenarios, the slowdowns will be greatly important. Thus, for the newly reported bugs that have been fixed, we run their test cases with large amount of data like what we did in Section 6.2, and analyzed their slowdowns. Our experiment shows that, the slowdowns range from 0.2% to 35.4%, and 3 bugs cause Out of Memory (OOM).

7 DISCUSSION

While our experiments show that CacheCheck is promising in detecting cache-related bugs in Spark applications, we discuss potential threats and limitations of our work.

Representativeness of our studied Spark applications. We select a number of cache-related bugs and Spark applications in our experiments. Our selected cache-related bugs come from real-world Spark applications. Our selected Spark applications, e.g., GraphX [22], MLlib [23] and Spark SQL [29], are well maintained and actively developed by the Spark community. Thus, we believe our studied cache-related bugs and Spark applications can represent the real-world cases.

Trade-off between memory and performance. Whether an RDD should be persisted or not is usually a trade-off between execution performance and the memory requirement of the persisted RDD. Our approach does not take the memory requirement into consideration. However, we believe that CacheCheck can still give developers useful suggestions about how to enforce proper caching decisions. Additionally, Spark provides various cache storage levels, e.g., off-heap, on-disk, in-memory. CacheCheck cannot provide useful suggestions for cache levels.

Generalization to other systems. CacheCheck is currently designed for Spark. It may be generalized to other big data systems that have the same or similar cache mechanisms as Spark. For example, in order to reduce the cost of recomputation and reloading files, Hadoop and Flink have utilized or planned to utilize similar cache mechanisms. CacheCheck may be adapted into these systems.

8 RELATED WORK

To the best of our knowledge, no previous work can detect cache-related bugs in Spark applications. In this section, we discuss related work that is close to ours.

Cache optimization in Spark. Existing work mainly focuses on cache optimization techniques in Spark implementation. Neutrino [47] employs fine-grained memory caching of RDD partitions and uses different in-memory cache levels based on runtime characteristics. Yang et al. [48] developed some adaptive caching algorithms to make online caching decisions with optimality guarantees. Zhang et al. [52] studied the influence of disk use in the Spark cache mechanism, and proposed a method of combined use of memory and disk caching. These works may eliminate the use of *persist*() and *unpersist*() operations and cache levels. To meet this all-or-nothing property in data access, PACMan [30] coordinates access to the distributed caches and designs new cache replacement mechanisms. LRC (Least Reference Count) [49] exploits the application-specific DAG information to optimize the cache replacement mechanism. However, it is still a common practice to enforce caching decisions by developers causing severe performance slowdown. CacheCheck automatically detects improper *persist*() and *unpersist*() usage in Spark applications. Thus, our CacheCheck is orthogonal to these pieces of existing work.

Memory management optimization in Spark. Researchers have proposed some approaches to improve memory management in big data systems. An experimental study [46] has shown that garbage collectors can affect big data processing performance. Yak [39] divides the managed heap in JVM into a control space and a data space, and optimizes garbage collection for big data systems. Skyway [38] can directly connect managed heaps of different (local or remote) JVM processes, and optimizes object serialization / deserialization. Stark [36] optimizes in-memory computing on dynamic collections. Panthera [42] analyzes user programs running on Apache Spark to infer their coarse-grained access patterns, and then provides fully automated memory management technique over hybrid memories. Different from these approaches, CacheCheck focuses on the caching decisions in Spark applications.

Resource leak detection in traditional programs. Resource leak in programs can cause severe problems, e.g, performance degradation and system crash [44]. Researchers have made lots of efforts to detect resource leak. Sigmund et al. [32] and Yulei et al. [40] utilized value-flow analysis to detecting memory leaks in C programs. Emina et al. [41] developed Tracker to detect resource leak in Java

programs. Relda [34] and Relda2 [44] detect resource leak in Android applications. However, all these works cannot handle with the execution model of Spark, e.g., lazy execution. Thus, they cannot be applied to detect cache-related bugs in Spark applications.

9 CONCLUSION

Apache Spark uses caching mechanisms to reduce the RDD recomputation in Spark applications, and relies on developers to enforce correct caching decisions. In this paper, we summarize six cache-related bug patterns in Spark applications. Our experiments on real-world cache-related bugs show that cache-related bugs can seriously degrade the performance of Spark applications. We further propose CacheCheck, which can automatically detect cache-related bugs by analyzing the execution traces of Spark applications. Our evaluation on real-world Spark applications shows that CacheCheck can effectively detect cache-related bugs. In the future, we plan to further improve the detection capability of CacheCheck using new techniques, e.g., static analysis, and performance estimation of cache-related bugs.

ACKNOWLEDGEMENTS

We thank Kai Zhang, Yange Fang, Kai Kang, and Xingtong Ye for their contributions in validating detected unknown bugs. This work was partially supported by National Key Research and Development Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 61802377, 61702490), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

REFERENCES

[1] 2004. *Apache Hadoop.* Retrieved January 10, 2020 from http://hadoop.apache.org/
[2] 2014. *Mining ecommerce graph data with Apache Spark at Alibaba Taobao.* Retrieved January 6, 2020 from https://databricks.com/blog/2014/08/14/mining-graph-data-with-spark-at-alibaba-taobao.html
[3] 2014. *SPARK-1266: Persist factors in implicit ALS.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-1266
[4] 2014. *SPARK-2661: Unpersist last RDD in bagel iteration.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-2661
[5] 2014. *SPARK-3290: No unpersist callls in SVDPlusPlus.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-3290
[6] 2014. *SPARK-3918: Forget Unpersist in RandomForest.scala(train Method).* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-3918
[7] 2015. *Distributed t-SNE via Apache Spark.* Retrieved January 10, 2020 from https://github.com/saurfang/spark-tsne/
[8] 2015. *MCL spark.* Retrieved January 10, 2020 from https://github.com/joandre/MCL_spark/
[9] 2015. *SPARK-10182: GeneralizedLinearModel doesn't unpersist cached data.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-10182
[10] 2015. *SPARK-7100: GradientBoostTrees leaks a persisted RDD.* Retrieved January 23, 2020 from https://issues.apache.org/jira/browse/SPARK-7100
[11] 2015. *SPARK-7116: Intermediate RDD cached but never unpersisted.* Retrieved January 23, 2020 from https://issues.apache.org/jira/browse/SPARK-7116
[12] 2015. *Spark betweenness.* Retrieved January 10, 2020 from https://github.com/dmarcous/spark-betweenness/
[13] 2016. *SPARK-16697: Redundant RDD computation in LDAOptimizer.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-16697
[14] 2016. *SPARK-16880: Improve ANN training, add training data persist if needed.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-16880
[15] 2016. *SPARK-18356: KMeans should cache RDD before training.* Retrieved January 23, 2020 from https://issues.apache.org/jira/browse/SPARK-18356
[16] 2016. *SPARK-18608: Spark ML algorithms that check RDD cache level for internal caching double-cache data.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-18608
[17] 2016. *SPARK-29878: Improper cache strategies in GraphX.* Retrieved January 19, 2020 from https://issues.apache.org/jira/browse/SPARK-29878
[18] 2017. *Tuning Apache Spark for Large-Scale Workloads.* Retrieved January 10, 2020 from https://databricks.com/session/tuning-apache-spark-for-large-scale-workloads
[19] 2018. *Moving eBay's Data Warehouse Over to Apache Spark – Spark as Core ETL Platform at eBay.* Retrieved January 10, 2020 from https://databricks.com/session/moving-ebays-data-warehouse-over-to-apache-spark-spark-as-core-etl-platform-at-ebay
[20] 2018. *SPARK-26006: mllib Prefixspan.* Retrieved January 23, 2020 from https://issues.apache.org/jira/browse/SPARK-26006
[21] 2019. *Cache missing in MCL.scala.* Retrieved January 25, 2020 from https://github.com/joandre/MCL_spark/issues/20
[22] 2019. *GraphX | Apache Spark.* Retrieved January 10, 2020 from http://spark.apache.org/graphx/
[23] 2019. *MLlib | Apache Spark.* Retrieved January 10, 2020 from http://spark.apache.org/mllib/
[24] 2019. *SPARK-29844: Improper unpersist strategy in ml.recommendation.ASL.train.* Retrieved January 10, 2020 from https://issues.apache.org/jira/browse/SPARK-29844
[25] 2019. *SPARK-29844 pull request: Improper unpersist strategy in ml.recommendation.ASL.train.* Retrieved January 10, 2020 from https://github.com/apache/spark/pull/26469
[26] 2019. *SPARK-29856: Conditional unnecessary persist on RDDs in ML algorithms.* Retrieved January 25, 2020 from https://issues.apache.org/jira/browse/SPARK-29856
[27] 2019. *SPARK-29872 pull request: Improper cache strategy in examples.* Retrieved January 10, 2020 from https://github.com/apache/spark/pull/26498
[28] 2019. *Spark JIRA.* Retrieved January 10, 2020 from https://issues.apache.org/jira/projects/SPARK
[29] 2019. *Spark SQL | Apache Spark.* Retrieved January 10, 2020 from http://spark.apache.org/sql/
[30] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 267–280.
[31] Mariem Brahem, Stephane Lopes, Laurent Yeh, and Karine Zeitouni. 2016. AstroSpark: Towards a distributed data server for big data in astronomy. In *Proceedings of SIGSPATIAL PhD Symposium*. 3:1–3:4.
[32] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 480–491.
[33] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 599–613.
[34] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 389–398.
[35] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark: Lightning-fast big data analysis.* O'Reilly Media, Inc.
[36] Shen Li, Md Tanvir Amin, Raghu Ganti, Mudhakar Srivatsa, Shanhao Hu, Yiran Zhao, and Tarek Abdelzaher. 2017. Stark: Optimizing in-memory computing for dynamic dataset collections. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*. 103–114.
[37] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.
[38] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 56–69.
[39] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 349–365.
[40] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 254–264.
[41] Emina Torlak and Satish Chandra. 2010. Effective interprocedural resource leak detection. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*. 535–544.
[42] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 347–362.

[43] Kun Wang, Ke Zhang, and Chengxue Gao. 2015. A new scheme for cache optimization based on cluster computing framework Spark. In *Proceddings of International Symposium on Computational Intelligence and Design (ISCID)*. 114–117.

[44] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Transactions on Software Engineering (TSE)* 42, 11 (2016), 1054–1076.

[45] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 1071–1085.

[46] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An experimental evaluation of garbage collectors on big data applications. *Proceedings of the VLDB Endowment (VLDB)* 12, 5 (2019), 570–583.

[47] Mohit Xu, Erci andSaxena and Lawrence Chiu. 2016. Neutrino: Revisiting memory caching for iterative data analytics. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 16–20.

[48] Zhengyu Yang, Danlin Jia, Stratis Ioannidis, Ningfang Mi, and Bo Sheng. 2018. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In *Proceedings of International Conference on Cloud Computing (CLOUD)*. 277–284.

[49] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*. 1–9.

[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2:1–2:14.

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 10:1–10:7.

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