# CACHEIT: Application-Agnostic Dynamic Caching for Big Data Analytics

Dat Nguyen*, Muhammad Rafid†, Nathanael Santoso‡, Khanh Nguyen*
*Texas A&M University, USA, {tiendat.ng.cs, khanhtn}@tamu.edu
†Dian Nuswantoro University, Indonesia, 111202012803@mhs.dinus.ac.id
‡Bandung Institute of Technology, Indonesia, 13520129@std.stei.itb.ac.id

2024 IEEE International Conference on Big Data (BigData) | 979-8-3503-6248-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/BigData62323.2024.10825231

***Abstract*—Apache Spark arguably is the most prominent Big Data processing framework tackling the scalability challenge of a wide variety of modern workloads. A key to its success is caching critical data in memory, thereby eliminating wasteful computations of regenerating intermediate results. While critical to performance, caching is not automated. Instead, developers have to manually handle such a data management task via APIs, a process that is error-prone and labor-intensive, yet may still yield sub-optimal performance due to execution complexities. Existing optimizations rely on expensive profiling steps and/or application-specific cost models to enable a postmortem analysis and a manual modification to existing applications.**

**This paper presents CACHEIT, built to take the guesswork off the users while running applications as-is. CACHEIT analyzes the program's workflow, extracting important features such as dependencies and access patterns, using them as an oracle to detect high-value data candidates and guide the caching decisions at run time. CACHEIT liberates users from low-level memory management requirements, allowing them to focus on the business logic instead. CACHEIT is application-agnostic and requires no profiling or a cost model. A thorough evaluation with a broad range of Spark applications on real-world datasets shows that CACHEIT is effective in maintaining satisfactory performance, incurring only marginal slowdown compared to the manually well-tuned counterparts.**

***Index Terms*—caching, memory management, dynamic analysis**

## I. INTRODUCTION

Large-scale data processing frameworks are the backbone of modern computing. Some popular frameworks, to name a few, are Spark [1, 2], MapReduce [3], and Flink [4]. These frameworks use a dataflow programming model where users write applications as a sequence of operations on the input datasets. The processing engines enable a push-button deployment that can scale from a single node to a datacenter-sized cluster. Among these Big Data analytics systems, Apache Spark is the most prominent framework. The ecosystem based on Spark flourishes with support for a wide variety of workloads such as high-level querying (SparkSQL [5]), graph processing (GraphX [6]), and AI/ML (SparkML) in multiple languages such as Python, Scala, Java, and R.

A key feature that makes Spark superior to a framework such as MapReduce is the advocating for keeping data in memory (a.k.a. caching) for reuse and thereby eliminating repeated computations and/or disk I/O to generate the same data. This is highly relevant to modern algorithms and workloads such as graph analytics and machine learning which are often iterative — a same set of functions is invoked on input datasets repeatedly. Despite its importance, caching is not automated. Spark provides APIs for developers to *manually* cache and uncache data. Even though the APIs have rich semantics (e.g., using `persist()`, users can choose between memory, disk, or a combination of both), manual data caching is challenging. To be effective, developers must have a deep understanding of the workflow including the working set size and data dependencies, as well as the internals of Spark. As a simple example, if two consecutive jobs do not share any data dependency, caching data of the earlier job needlessly increases the memory pressure. This is more problematic in cases where the working set size exceeds the memory capacity. Indeed, evidence [7, 8], as well as our experience, shows that mistakes in caching data are detrimental: applications may crash due to out-of-memory errors, or suffer a significant slowdown. The impact is more devastating for long-running or latency-sensitive applications where a slowdown of one worker node can prolong the entire pipeline. These failures are the direct consequences of undue memory pressure caused by keeping useless data in memory, exceeding memory capacity and/or magnifying the overhead of the runtime system. In practice, this is a trial-and-error process that is labor-intensive and error-prone. Developers usually follow a set of best-practice recommendations for picking data to be cached and may miss performance gain opportunities.

There is a body of work on identifying valuable data for caching [9–16]. However, the majority of these works rely on profiling or expert experience to derive an application-specific cost model that can provide hints to users. While commendable, there is much to be desired. Profiling is known to suffer from a plethora of problems such as low accuracy to unseen datasets and being expensive to use. Moreover, these techniques are postmortem. Using the analysis result which might contain false positives, users manually modify the applications for re-execution (i.e., inserting the API `cache()`, which is short-hand for `persist()` using memory — if the target is within Spark libraries, this might not be possible), then observe the performance. The tedious process repeats until users are satisfied. Some other works [8, 17] mitigate this laborious effort by exposing the dynamic workflow to the users, allowing them to interactively reconfigure the run time behavior. Still, involving human in the optimization process makes these tools undesirable.

We argue that the burden on users to manually cache data at development time outweighs the flexibility provided by the framework. Relying on users to enforce caching decisions makes it infeasible to adapt to dynamic execution behaviors. This explains why even after a lengthy experimentation of various combinations, there is still room for performance improvement in most applications. Hence, the overarching goal of this work is 1) to free developers from the labor-intensive task of detecting and manually enforcing caching decisions on a set of data at compile time while 2) providing satisfactory performance at run time. Other attempts for the same goal [18, 19] aim to find an optimum data caching decision using a search algorithm. Although automated, these methods may be costly because they must find the optimum in a large search space that scales with the program duration and complexity. Work that performs caching by dynamically analyzing the data flow [15] operates solely on the global view of the data dependency graph and, hence, falls short in recognizing changes in the usage patterns to timely remove stale data.

To that end, we propose CACHEIT – implemented as an extension to the Spark runtime system. CACHEIT intercepts Spark jobs submission, and automatically extracts and analyzes dataflow graphs of the job. Using a combination of local and global analyses, it tracks data dependencies and access patterns to identify frequently used data and passes such information to the runtime system. During execution, high-value data are automatically kept in memory for future reuse without any user effort. The idea is simple yet — as demonstrated in our evaluation on several benchmarks using real-world workloads — effective. CACHEIT is practical as it does not require any profiling and can be used with any application as-is without any modification to the program's code. The end-to-end execution time of the application with CACHEIT's support is on par with the manually well-tuned versions. Our work prevents users from making caching mistakes, allowing them to focus on high-level business logic instead.

In summary, the contributions of this paper are as follows:

- We provide two analysis algorithms capturing local and global reuse patterns of RDDs, the immutable dataset abstraction, in Spark applications. These algorithms enable timely cache and uncache decisions of the RDDs.
- We thoroughly evaluate the effectiveness of CACHEIT. The results are positive, CACHEIT delivers satisfactory execution time without much user effort.

## II. BACKGROUND & MOTIVATION

In this Section, we briefly introduce the execution model of Spark. We then present an empirical study on the impact of data caching mistakes on performance to motivate our work.

### A. Spark execution model

A Spark cluster consists of one Master node running the driver program, and multiple Worker nodes running executors, as shown in Figure 1. The driver represents the control plane: it accepts user applications and drives the flow of the applications as well as work re-execution in case of errors. The executors perform actual data processing. Both driver and executor are instances of Java Virtual Machines (JVMs). Spark employs a core data abstraction called Resilient Distributed Datasets (RDDs) [20]. An operation on RDD is partitioned into small tasks, each operates on a fragment of the RDD in parallel. For fault-tolerance purposes, RDDs are immutable. When a user submits an application, the driver analyzes the logic and represents the workflow as a directed acyclic graph (DAG) of operations, each node is an RDD. In Spark's terminology, an operation is either a transformation or an action. Transformations produce intermediate RDDs while actions return results to the driver. The execution of an action triggers the execution of all transformations along the actions' dependency graph (or in Spark's lingo, lineage). An action establishes a barrier, delaying transformations until the next action is called. Effectively, actions split a program into multiple jobs, each with a different lineage/DAG.

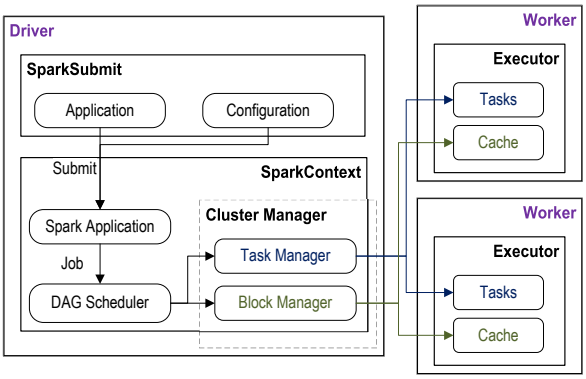


Figure 1: Spark Architecture. The Driver controls the execution flow while Workers perform computation in parallel, each operates on a partition of an RDD.

The result of an operation does not persist. Consequently, an operation referenced in multiple jobs are invoked repeatedly, once for each, to generate *the same* data. To avoid such wasteful computations, Spark programmer has the option to keep the result in memory explicitly[1] by invoking the API `cache()` in the program. Upon executing an action, cached RDDs can be used in lieu of the computations generating them, thereby avoiding performance degradation. Under the hood, these RDDs are marked so that once materialized at run time, they are not garbage-collected by the JVM.

### B. Impact of improper RDD caching to execution time

Caching appropriate RDDs is critical to performance by avoiding wasteful recomputations each time the executor requires them. A naïve solution of caching the result of all operations, assuming infinite memory, is not optimal because it creates undue memory pressure on the runtime system. Evidence [21–23] shows the runtime can spend up to 50%

[1]by default, as this is the most efficient way. Spark does provide other options for caching, which we defer them to future work.

of execution on automatic memory management tasks such as garbage collection (GC).

As a motivating example, we ran PageRank application on a Wikipedia dataset [24] in a cluster using multiple sizes of the heap ranging from 16GB to 128GB, which was very generous given the dataset is of size of 6GB. The code is shown in Listing 1 and graphically visualized in Figure 2. The application was run under three scenarios representing memory consumption ranking from the least to the most:

- **NoCache:** No RDD is persisted, every RDD must be recomputed.
- **ManualIdeal:** Manually applying `cache()` to a selective set of RDDs after an exhaustive search of all combinations. The version with the fastest execution is chosen.
- **CacheAll:** Every RDD is cached in memory, no recomputation is needed.

Listing 1: A Spark PageRank application in Scala.

```
1  def PageRank(edges:RDD, iters:Int):Unit={
2    val graph = edges.groupByKey()
3    var ranks = graph.mapValues(v => 1.0)
4
5    for (i <- 1 to iters) {
6      val contribs = graph.join(ranks).values.
           flatMap{ case (urls, rank) =>urls.map(url
           => (url, rank / urls.size))}
7      ranks = contribs.reduceByKey(_ + _).mapValues
           (0.15 + 0.85 * _)
8    }
9    print(ranks.collect())
10 }
```

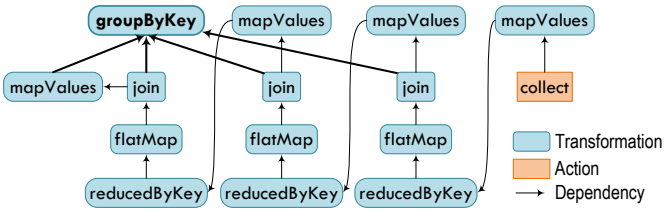


Figure 2: A DAG of PageRank running 3 iterations. The RDD created by the first `groupByKey` transformation is reused in subsequent iterations.

Each heap size configuration is run three times and the average performance is reported in Table I. We normalize the performance to that of **ManualIdeal**. Aside from end-to-end time, we also report the performance of G1 GC [25] - JVM's default garbage collector. For PageRank (PR), the best performance is obtained with only the RDD on Line 2, representing the input graph, cached in memory. This RDD is reused multiple times in the loop (Line 6). Failure to cache this RDD (in **NoCache**) results in, on average, a 30% slowdown due to recomputation in each iteration. Somewhat counter-intuitive, GC cost is also increased in this case (1.41× on average). This is because unpersisted intermediate data are marked as garbage at the end of each job to be collected, forcing the next jobs to recompute these data and allocate additional heap space. Consequently, memory is allocated much faster than that can be collected, thus triggering more frequent GC runs. Naïvely caching more RDDs than necessary in **CacheAll** also degrades the performance. A more occupied heap is constantly under pressure to find space for new allocations, forcing the GC to start more frequently and run for a longer duration. GC overhead in **CacheAll** is more than doubled on average, and is as high as 7.4×. A prolonged iteration holds up the entire pipeline, decreasing throughput, and further prolonging the execution. We observe an average increase of 56% in the execution time of PageRank with **CacheAll**. Similar behavior is observed with other applications such as KMeans (KM), and TransitiveClosure (TC), as shown in Table I.

Table I: Slowdowns (×) incurred by improperly caching, normalized to performance of **ManualIdeal**. For each metric, we report the range and the mean, across all different heap sizes ranging from 16GB to 128GB.

| **Apps** | **NoCache** | | **CacheAll** | |
|---|---|---|---|---|
| | Ex. Time | GC | Ex. Time | GC |
| PR | 1.31 ∼ 1.41 (1.37) | 0.89 ∼ 2.08 (1.41) | 1.46 ∼ 1.86 (1.56) | 1.51 ∼ 7.40 (2.30) |
| KM | 1.13 ∼ 3.28 (2.09) | 0.02 ∼ 0.74 (0.25) | 1.24 ∼ 1.68 (1.41) | 0.99 ∼ 7.51 (5.84) |
| TC | 1.48 ∼ 2.05 (1.60) | 1.39 ∼ 1.42 (1.41) | 1.02 ∼ 1.22 (1.10) | 1.08 ∼ 5.90 (1.40) |

**Discussion** The study confirms that over- and under-estimating the set of candidate RDDs can negatively impact the performance of a Spark program even in the case memory is not a scarce resource, a generous assumption that often fails to hold in practice. Therefore, a Spark program developer is required not only to understand the implemented algorithm but also the internals of Spark to develop a performant application. It is worth noting that even if the programmer understands the implication of Spark's execution model, determining a good set of RDDs and having a good performance is not a trivial problem. The reason is twofold: first, the search space for all combinations of RDDs as caching candidates is huge; second, often users need to use Spark libraries which contain hidden RDDs that are out of users' direct control.

## III. CacheIt's Design

In this Section, we present the design of CacheIt with its two job analyses handling two common RDD reuse patterns. We also describe a special pattern found empirically and how CacheIt handles such a case.

### A. Overview & Design rationale

CacheIt can be implemented as a static analysis that analyzes the whole execution plan of a Spark application before its execution (using a byte code analyzer such as Soot [26]). However, because of the tight coupling between the application and the platform, a static analysis has to exhaustively explore all execution paths including those in libraries, which is the well-known path explosion problem [27], making CacheIt not able to scale to Spark's codebase which has millions of lines

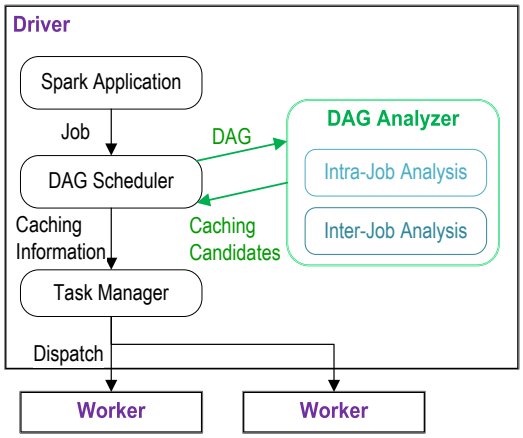


Figure 3: Overview of CACHEIT's design. The *DAGAnalyzer* is new to the Spark engine. It analyzes a job's DAG and returns the caching candidates to the runtime.

of code. Additionally, static analysis is often too conservative, thus lacking the ability to precisely detect caching candidates.

To address these challenges, we implement CACHEIT as a runtime support. Specifically, as shown in Figure 3, a new *DAGAnalyzer* is created in the driver to 1) intercept each job submission at run time, 2) analyze workflow information from the DAGs created by Spark's DAGScheduler to select caching candidates, and 3) return the set of caching candidates to Spark before resuming the job submission.

Because Spark generates DAGs on a per-job basis, each DAG contains the RDDs relating specifically to its job execution. To allow for dynamic behaviors in a Spark program, whether a job is executed depends on the results returned from the previous job(s). Hence, caching decisions made for each job in isolation from the job sequence are ineffective because the intermediate results may be utilized across jobs. An ideal data caching strategy demands the knowledge of future jobs, which is challenging to dynamically derive. However, because there often exists patterns of RDD usage in a job sequence, we can leverage past jobs for future caching decisions. The DAGAnalyzer, therefore, uses a combination of two analyses to have a complete picture of the execution: an intra-job (local) analysis to identify candidates using dependencies in the current dataflow graph; and an inter-job (global) analysis to capture access patterns across jobs.

### B. Intra-job analysis

Algorithm 1 shows the pseudo-code of the DAGAnalyzer's intra-job analysis. The analysis is a BFS-like graph traversal starting from the RDD on which the action is invoked, which we refer to as root RDD ($d_r$ on Line 11). For each RDD in the DAG, the algorithm computes the number of direct downstream RDDs, called $dependentCount$ (Lines 4-10). While simple, $dependentCount$ captures how important an RDD is in the current job. After filtering out all low-value RDDs, i.e., having $dependentCount$ less than a user-defined threshold $T_{DC}$, CACHEIT marks the remaining RDDs as caching candidates at

**Algorithm 1:** Intra-job analysis

```
Input:
integer T_DC: Intra-job dependentCount threshold
RDD d_r: root RDD of the current job
/* BFS on the current job's DAG to count each RDD's number of
   dependents */
1  Map⟨ RDD → integer ⟩ dependenceCounts ← ∅
2  Queue⟨ RDD ⟩ toVisit ← ∅
3  Set⟨ RDD ⟩ visited ← ∅
4  def visit(d: RDD):
5      if d ∉ visited then
6          visited.add(d)
7          dependenceCounts[dep] ← 0
8          foreach RDD dep ∈ d.dependencies() do
9              dependenceCounts[dep]++
10             toVisit.enqueue(dep)
11 toVisit.enqueue(d_r)
12 while toVisit is not empty do
13     visit(toVisit.dequeue())
   /* Candidate RDDs: multiple dependents in the current job */
14 foreach RDD d ∈ dependenceCounts.keySet() do
15     if dependenceCounts[d] ≥ T_DC then
16         d.mark_cache()
```

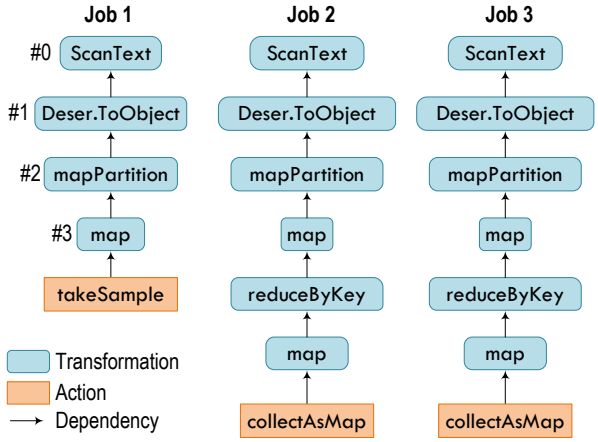


Figure 4: Simplified DAGs of the first three jobs of KMeans. The first 4 RDDs (RDD#0 to RDD#3) appear in all DAGs but only RDD#3 should be cached.

run time (see Section III-E). Because we visit each RDD node in the DAG only once, this analysis is lightweight with a time complexity of $O(D)$ with $D$ being the number of RDDs in the current DAG. Often this $D$ is small, making the analysis overhead negligible. Because each Spark job is reasonably unique, all RDDs cached by CACHEIT's intra-job analysis are automatically uncached at the end of each job unless the inter-job analysis (explained next in Section III-C) detects that they should stay in the memory to be reused in future jobs.

### C. Inter-job analysis

If multiple consecutive jobs share a common set of RDDs, a subset of such RDDs should be cached in memory for faster access in later jobs. In Figure 4, each job shown represents an iteration of the KMeans program. Because all RDDs in each

**Algorithm 2:** Inter-job analysis

```
Input:
integer T_AC: Inter-job appearanceCount threshold
RDD d_r: root RDD of the current job
Set⟨ RDD ⟩ currentJob: RDDs in the current job
Queue⟨ Set⟨ RDD ⟩ ⟩ latest: RDDs in the latest W jobs
/* Counting RDDs appearance */
1  Set⟨ RDD ⟩ nominated ← ∅
2  foreach RDD d ∈ currentJob do
3      foreach Set⟨ RDD ⟩ S ∈ latest do
4          integer appearanceCount := 0
5          if d ∈ S then
6              appearanceCount++
7      if appearanceCount ≥ T_AC & d is not cached then
8          nominated.add(d)
9  Set⟨ RDD ⟩ reached ← ∅
10 def revisit(d: RDD):
11     if d ∉ reached then
12         reached.add(d)
13         if d is not cached & d ∉ nominated then
14             foreach RDD dep ∈ d.dependencies() do
15                 revisit(dep)
16 revisit(d_r)
   /* Candidate RDDs: found to be reused in W jobs */
17 foreach RDD d ∈ currentJob do
18     if d ∈ (nominated ∩ reached) then
19         d.mark_cache()
```

DAG have a low *dependentCount* of no more than 1, they are not cached by the intra-job analysis. This caching decision is incorrect because some RDDs (e.g., #0 to #3) are reused across jobs, and caching them can improve performance. A naïve inter-job solution would cache RDDs that appear in the majority of jobs. However, this is sub-optimal because the appearance of an RDD in a DAG is not always equivalent to it being accessed in an execution. In Figure 4, all jobs share the subset of RDDs#0 - #3. Intuitively, only RDD#3 should be cached because it is the closest RDD to all of the dependents - caching upstream RDDs#0-#2 will not bring additional benefits if RDD#3 is already in memory. Worse, they add space overheads.

Algorithm 2 shows how CACHEIT's inter-job analysis precisely selects RDDs for caching. The analysis keeps a history of $W$ latest DAGs in *latest* queue, updated at the beginning of each job scheduling. Then, the DAGAnalyzer performs two steps. In the first step (Line 1-8), CACHEIT inspects *latest* (Line 3) to derive the number of DAGs each RDD appears in, called *appearanceCount*. RDDs with *appearanceCount* not smaller than a user-defined threshold $T_{AC}$ becomes the *nominated* set — an over-estimation of the caching candidates (RDD#0 to RDD#3 in Figure 4). In the second step (Lines 10 - 16), CACHEIT filters out from *nominated* any RDD that has all of their downstream RDDs are cached or in *nominated*. Specifically, the DAGAnalyzer performs a recursive DFS-like DAG traversal from root RDD $d_r$ that stops as soon as a cached or *nominated* RDD is visited (Lines 10 - 16). All visited RDDs form a *reached* set whose data are already cached or will get materialized in the current job execution. *reached* RDDs that are also in *nominated* are the top caching candidates. (Line 17 - 19). In Figure 4, *nominated* RDD#0-#2 are not cached due to having a common downstream *nominated* RDD#3. The DAGAnalyzer will uncache an RDD if its *appearanceCount* in the latest $W$ DAGs falls below $T_{AC}$.

Because the *appearanceCount* of an RDD is always less than $T_{AC}$ in the first few jobs, CACHEIT delays caching the RDDs by at least one job (i.e., when $T_{AC} = 2$) compared to the ideal, which will incur a penalty. However, given modern workloads are iterative, such a penalty of *delayed caching* can be sufficiently amortized.

### D. A special case of root RDD reuse

We noticed a special RDD usage pattern of the TransitiveClosure (TC) execution as a side-effect of RDD's immutability. TC is a graph analysis program that runs a fixed-point algorithm to compute all paths transitively in a graph. This is represented by an RDD variable `tc` shown in Listing 2. The program iteratively joins graph edges with the already-discovered paths in `tc` to generate new paths. Because RDDs are immutable, each iteration assigns `tc` a new RDD instance representing the up-to-date paths (Line 8). Then, the program calls action `count()` (Line 9) to check for convergence before passing the transformed RDD to the next iteration. Applying `count()` on the RDD instance in `tc` makes it the root RDD of the job. Figure 5 shows a graphical illustration of the situation. An expert will know that the root RDD `tc` (the first instance of RDD#2, #4, #6, and #8) will be used in the next iteration, and will correspondingly cache these RDDs at the first time an action is called upon it. Failing to do so results in the RDD getting recomputed.

Listing 2: TransitiveClosure (TC) application (simplified). Each RDD instance assigned to variable `tc` (Line 8, RDD#2, #4, #6, and #8 in Figure 5) must be cached at their first encounter to be efficiently used in the next iteration (Line 8). Inter-job analysis (Section III-C) fails to timely cache these root RDDs as *delayed caching* occurs to each new RDD instance of `tc`.

```
1  def TC(){
2    var tc = // all paths, i.e., the result
3    var edges = // edges of the graphs
4    var oldCount = 0
5    var nextCount = tc.count()
6    do {
7      oldCount = nextCount
8      tc = tc.union(tc.join(edges)).distinct()
9      nextCount = tc.count()
10   } while (oldCount != nextCount)
11 }
```

This root RDD reuse pattern is a special case of inter-job RDD reuse that must be handled differently. For iterative workloads that operate on the same RDD at each iteration such as KMeans (discussed in Section III-C), delayed caching happens once at the first time this RDD appears. TC is different. Because TC repeatedly reuses last iteration's RDD to to generate a new RDD instance (`tc` on Line 9), delayed

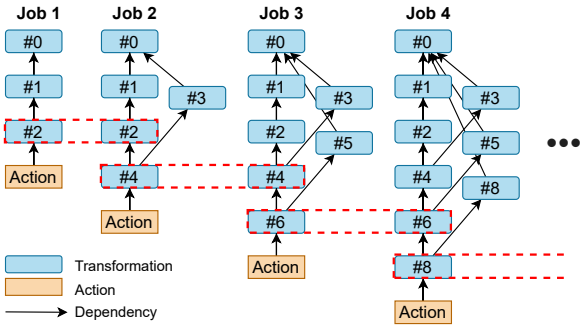


Figure 5: A simplified DAG of the first 4 jobs of TC.

caching happens at every iteration. This inefficiency, although being a corner case that only applies to a subset of programs such as TC, needs to be addressed.

We overcome this inefficiency by adding another analysis step to CACHEIT after inter-job analysis. Each job keeps a counter called $rootReuseCount$, initialized to 0. The DAGAnalyzer inspects each pair of consecutive jobs/DAGs in $W$ latest jobs and increments the current job's $rootReuseCount$ each time a root RDD of a job is used in the next job. The root RDD of the current job is cached if the current $rootReuseCount$ is at least 2. Otherwise, we leave the root RDD of the current job untouched. For example, in Figure 5, at Job 3, CACHEIT detects there have been two reuses (RDD#2 and RDD#4, highlighted in red boxes) and therefore marks RDD#6 as a caching candidate.

### E. Integrating CACHEIT to Spark runtime

To implement CACHEIT, the existing API `cache()` in Spark is modified to be `No_Op` when CACHEIT is being used. The TaskScheduler sends tasks along with RDD caching candidates which are retrieved from CACHEIT to the workers. At run time, if an RDD is a candidate according to CACHEIT, `cache()` is invoked – we did not modify such an operation to demonstrate the impact of our analyses. If all cached RDDs cannot fit in the memory, a standard LRU eviction policy is used. While an alternative eviction policy such as LRC [14] or LPW [28] has been shown to help improve memory utilization, it is not our goal to explore such a policy in this work.

## IV. EVALUATION

**Setup** We implemented CACHEIT in Spark version 3.2.3. The programs are executed using a cluster with one master node and five worker nodes. Each node has 2 Intel®Xeon®Silver 4214R processors, 180GB of memory, and 500GB of SSD, connected via a Mellanox ConnectX-6 card. Hadoop Distributed File System (HDFS) is used for distributed storage.

We evaluated CACHEIT using a collection of programs commonly used to benchmark the performance of Spark in suites such as HiBench [29]. Table II lists these programs along with the input datasets in various scales. The programs are selected to represent various RDD usage patterns and DAG structures, which are important to CACHEIT's caching decisions. For example, WordCount is a simple map-reduce workload that does not require caching because each created RDD is only used once. Other applications that require RDD caching are diverse in their DAG structures. For instance, PageRank is a single-job workload that reuses a subset of RDDs at each execution, while KMeans, TransitiveClosure (TC), SupportVectorMachine (SVM), RandomForest, and LogisticRegression (LR) have RDD reuses across jobs.

Because GC is a major factor of the JVM, and the caching decisions made by CACHEIT directly affect memory usage, we use multiple heap configurations to simulate a spectrum of memory settings, from abundance to scarcity, ranging from 128GB to 1GB. All experiments involving CACHEIT have caching configurations set to the default values ($T_{DC} = 2$; $T_{AC} = 2$; $W = 10$). Each experiment is run three times to avoid noise, and the average is reported. The variation among runs is negligible (normalized standard deviations are no more than $0.5\%$). For each run, we collect the execution time and the GC time.

**Methodology** While there are many similar works [9, 10, 12, 15], unfortunately, they are not open-sourced. We reimplemented **ReSpark** [15], which is closest to CACHEIT to assess its effectiveness. Similarly to the empirical study in Section II-B, we compare CACHEIT to **ManualIdeal**. We also run two extreme cases, **CacheAll** and **NoCache** (also described in Section II-B). They are cases where caching decisions are over- and under-estimated. We implement **CacheAll** at the job scheduler by caching all RDDs used for each job. The implementation of **NoCache** is a modification to the caching APIs to bypass their functionalities, similar to CACHEIT. In this Section, we normalize the performance of all caching strategies to that of **ManualIdeal**.

### A. Overhead of CACHEIT's analyses

CACHEIT pays the overhead of analyzing the job DAGs at run time. Because both analyses inspect the DAGs at each job scheduling, their duration scales linearly with the number of jobs inspected and the size of their DAGs. Table II reports the number of jobs executed, the largest size of the DAG, and the accumulated time of CACHEIT analyses spent when running with Large input datasets. Among the benchmark programs, RandomForest has the largest number of jobs (4250), and the longest analyzing time of 96.14 *seconds* in total. This overhead is insignificant considering that RandomForest takes at least 3.37 *hours*, i.e., this is only $\approx 0.7\%$ the execution time. For other programs, the overhead ranges from $0.008\%$ (PageRank) to $0.28\%$ (TC), showing that the analyses of CACHEIT are lightweight.

### B. Impact of CACHEIT on executions

Due to space constraints, we exclusively report and discuss the evaluation results with *Large* inputs. Nevertheless, CACHEIT's relative performance is consistent across all input sizes listed in Table II.

**Summary** Table III shows the execution time and the GC time of each program running with different heap sizes and caching strategies. The execution time using CACHEIT is

Table II: (a) Benchmark programs used to evaluate CACHEIT. SVM, RandomForest, and LR datasets have a number of data points × feature space dimensions. The inputs of TC and LR are synthetically generated; (b) Number of jobs executed for each input configuration; (c) The maximum number of RDDs visited by CACHEIT's analyses; And (d) the elapsed **ManualIdeal** execution time with the accumulated time of running CACHEIT's analyses; Numbers reported in (c) and (d) are from running the programs with *Large* inputs and the largest heap configuration.

| Programs | (a) Input datasets | | | (b) #Jobs executed | | | (c) Max #RDDs per job | (d) Elapsed time of | |
|---|---|---|---|---|---|---|---|---|---|
| | Small | Medium | Large | S | M | L | | **ManualIdeal** | CACHEIT's analyses |
| WordCount | Amazon reviews (9GB) [30] | Google reviews (20GB) [31] | Flight info (30GB) [32] | 1 | 1 | 1 | 8 | 4.33 $min$ | 28.25 $ms$ |
| PageRank | Wiki links PL (1GB) [24] | Wiki links RS (1.8GB) [24] | Wiki links EN (6GB) [24] | 1 | 1 | 1 | 186 | 10.17 $min$ | 48.89 $ms$ |
| KMeans | Wiki links PL (1GB) [24] | Wiki links RS (1.8GB) [24] | Wiki links EN (6GB) [24] | 9 | 22 | 27 | 52 | 3.26 $min$ | 282.15 $ms$ |
| TC | 1K vertices, 7K edges | 2K vertices, 10K edges | 3K vertices, 20K edges | 8 | 9 | 11 | 88 | 3.95 $min$ | 658.24 $ms$ |
| SVM | SUSY (5M×18) [33] | Higgs (11M×28) [33] | KDD2010 (8.4M×20M) [33] | 83 | 108 | 158 | 16 | 29.30 $min$ | 581.52 $ms$ |
| RandomForest | usps (7.3k×256) [33] | minst (60k×780) [33] | mnist8m (8.1M×784) [33] | 110 | 573 | 4250 | 13 | 202.32 $min$ | 96136.61 $ms$ |
| LR | 500 points×50 features | 20k points×75 features | 50k points×100 features | 100 | 250 | 500 | 2 | 38.53 $min$ | 645.14 $ms$ |

Table III: Execution time and GC time of benchmark programs with four caching strategies. Each measurement is normalized to that of **ManualIdeal** (best). TC performance is with Root Reuse (Section III-D) enabled. For each metric, we report the range and the mean from running with different heap sizes.

| **Apps** | **NoCache** | | **CacheAll** | | **ReSpark [15]** | | CACHEIT | |
|---|---|---|---|---|---|---|---|---|
| | Ex. Time | GC | Ex. Time | GC | Ex. Time | GC | Ex. Time | GC |
| WordCount | 0.95 ∼ 1.01 (1.01) | 0.88 ∼ 1.03 (0.95) | 1.03 ∼ 1.08 (1.06) | 4.30 ∼ 8.43 (5.61) | 1.01 ∼ 1.03 (1.01) | 0.93 ∼ 1.0 (0.96) | 0.99 ∼ 1.03 (1.01) | 0.92 ∼ 1.02 (0.97) |
| PageRank | 1.31 ∼ 1.41 (1.37) | 0.89 ∼ 2.08 (1.41) | 1.46 ∼ 1.86 (1.56) | 1.51 ∼ 7.40 (2.30) | 1.00 ∼ 1.05 (1.02) | 1.04 ∼ 1.30 (1.21) | 1.00 ∼ 1.00 (1.00) | 0.96 ∼ 1.01 (1.00) |
| KMeans | 1.13 ∼ 3.28 (2.09) | 0.02 ∼ 0.74 (0.25) | 1.24 ∼ 1.68 (1.41) | 0.99 ∼ 7.51 (5.84) | 1.05 ∼ 1.57 (1.26) | 0.98 ∼ 7.48 (3.60) | 1.00 ∼ 1.04 (1.03) | 0.94 ∼ 1.09 (1.00) |
| TC | 1.48 ∼ 2.05 (1.60) | 1.39 ∼ 1.42 (1.41) | 1.02 ∼ 1.22 (1.10) | 1.08 ∼ 5.90 (1.40) | 1.09 ∼ 1.14 (1.11) | 0.75 ∼ 1.10 (0.97) | 1.06 ∼ 1.11 (1.08) | 0.98 ∼ 1.01 (1.00) |
| SVM | 34.47 ∼ 60.61 (52.06) | 4.82 ∼ 12.06 (7.58) | 2.00 ∼ 29.74 (9.10) | 2.46 ∼ 50.84 (15.11) | 1.48 ∼ 1.86 (1.68) | 1.67 ∼ 1.95 (1.81) | 1.21 ∼ 1.33 (1.25) | 1.0 ∼ 1.41 (1.21) |
| RandomForest | 1.75 ∼ 1.83 (1.79) | 0.62 ∼ 0.99 (0.76) | 1.86 ∼ 1.94 (1.90) | 1.28 ∼ 1.48 (1.36) | 0.99 ∼ 1.07 (1.02) | 1.01 ∼ 1.58 (1.26) | 0.98 ∼ 1.01 (1.00) | 0.83 ∼ 1.17 (0.98) |
| LR | 0.98 ∼ 1.02 (0.99) | 0.94 ∼ 0.98 (0.96) | 1.03 ∼ 1.17 (1.12) | 1.10 ∼ 2.53 (1.56) | 1.00 ∼ 1.06 (1.03) | 0.93 ∼ 1.09 (1.02) | 0.92 ∼ 1.02 (0.98) | 0.94 ∼ 1.04 (0.99) |

close to, if not matching, that of **ManualIdeal** version. The largest slowdown of CACHEIT is 1.33× for SVM (2GB heap). **NoCache** incurs a slowdown from 1.06× (PageRank, 8GB heap) to up to 60× (SVM, 4GB heap) due to not caching any RDD. Overfilling the heap with RDDs in **CacheAll** increases the duration of GC to up to 50.8× that of **ManualIdeal** (SVM running with a strict 2GB heap), resulting in a slowdown of 29.7×. Overall, using CACHEIT yields effective heap consumption, which reflects in consistent performance across the heap sizes. Next, we discuss the execution of each program with CACHEIT in more detail.

In PageRank and WordCount, CACHEIT mirrors the caching behavior of **ManualIdeal** with negligible slowdown. WordCount is a single-job program that is composed of one `map()` followed by one `reduce()`. Hence, the intermediate data created by these transformations are only used once. Caching these data is not only unbeneficial but also adverse to the performance by introducing GC overhead when the heap size is limited (as seen in **CacheAll**). The simple DAG of WordCount does not satisfy any caching condition of CACHEIT, making its execution with CACHEIT mirror that of **NoCache** and **ManualIdeal**. Meanwhile, PageRank is also a single-job program. Its DAG has one RDD with a high $dependentCount$ as illustrated in Figure 2. CACHEIT's intra-job analysis correctly captures this high-value RDD, resulting in no slowdown compared to **ManualIdeal**.

KMeans, SVM, and RandomForest are composed of multiple jobs, thus they rely more on CACHEIT's inter-job analysis to effectively cache and reuse RDDs across jobs. Because caching decisions are delayed by 1 job (when $T_{AC} = 2$), execution slowdown is more noticeable. In SVM and KMeans, the executions are prolonged respectively to 1.25× and 1.03× on average. The impact of *delayed caching* (cf. Section III-C) is the most severe in SVM due to the high cost of RDD recomputation, confirmed by its **NoCache** version having up to 60× slowdown. In contrast, CACHEIT closely matches **ManualIdeal** ($< 1\%$

difference) in both RandomForest and LR, although each is caused by a different execution effect. In LR, the objects are small and lightweight, hence recomputing them is cheap, as demonstrated by **NoCache** having comparable running time to that of **ManualIdeal**. As such, the penalty of delayed caching is minimal, and thus LR's ideal performance is maintained with CACHEIT. In RandomForest, even though this penalty is significant in its first few jobs, suggested by **NoCache** having 1.79× slowdown on average. However, RandomForest is a long-running program with 4250 jobs (reported in Table II), each of which can fully benefit from the cached RDDs. Therefore, the penalty of delayed caching is efficiently amortized, resulting in the execution with CACHEIT on par with that of **ManualIdeal**.

In TC, CACHEIT applies the analysis for root RDD reuse (will be discussed in Section III-D). This analysis delays the caching decision by at least 2 jobs before the subsequent jobs can cache and reuse the root RDDs. This results in a slowdown of up to 1.11× at 4GB heap, and 1.08× on average.

Similar to CACHEIT, **ReSpark** achieves near-optimal performance in WordCount, PageRank, RandomForest, and LR ($\leq 3\%$ difference). For KMeans, TC, and SVM, CACHEIT is more performant than **ReSpark** thanks to identifying the set of RDDs reused across jobs sooner - at their second appearance.

### C. CACHEIT's threshold sensitivity

We vary $T_{DC}$ and $T_{AC}$ to evaluate CACHEIT's sensitivity. Figure 6 shows the normalized execution time of programs with CACHEIT using different threshold values (annotated as $T_{DC} \times T_{AC}$).

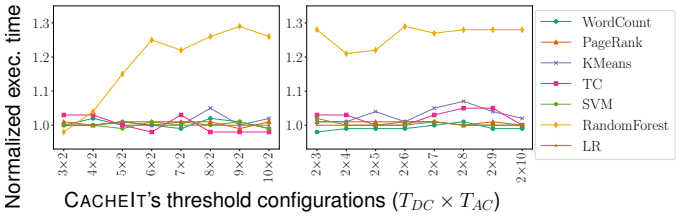


Figure 6: Execution time with different values of $T_{DC} \times T_{AC}$, normalized to that of the default configuration ($2 \times 2$).

Because WordCount and PageRank are single-job programs, varying $T_{AC}$ does not affect their executions. In WordCount, each RDD has at most 1 dependent, which does not exceed any tested $T_{DC}$ value. Hence, incrementing $T_{DC}$ does not affect the execution of WordCount. PageRank has a single RDD that is a common dependency of many other RDDs (shown in Figure 2). This RDD must be cached. In our experiment, the $dependentCount$ of this RDD is 25, exceeding all $T_{DC}$ values shown in Figure 6. When we set $T_{DC}$ larger than 25, CACHEIT does not cache this important RDD, and the PageRank execution mirrors that of **NoCache**, which has an average slowdown of 1.38× compared to CACHEIT and **ManualIdeal** (reported in Section IV-B).

KMeans, SVM, and LR are composed of multiple jobs where each RDD has at most 1 dependent. Therefore, increasing $T_{DC}$ does not affect their performance. These programs are iterative. They repeatedly apply operations on a subset of RDDs which are caching candidates. Consequently, such RDD has $appearanceCount$ equals to the number of jobs invoked (reported in Table II), exceeding $T_{AC}$ values in Figure 6. As we increment $T_{AC}$ to 10, despite the increased penalty caused by *delayed caching*, CACHEIT eventually caches them in subsequent jobs. Therefore, the slowdown is small, up to 1.07× (KMeans with $T_{DC} = 2$ and $T_{AC} = 8$).

RandomForest has a combination of multiple intra-job and inter-job RDD reuses. Incrementing $T_{DC}$ and $T_{AC}$ shrinks the set of RDDs to be cached in the memory. Consequently, we observe a rising trend in execution time with the slowdown as high as 1.29×. Interestingly, the trend stops when $T_{DC} \geq 6$ and $T_{AC} \geq 3$. We found this is because most of the RDDs have $dependentCount < 6$ and $appearanceCount < 3$. Hence, incrementing the thresholds past these points has little effect.

TC is special due to having the root RDD reuse pattern (cf. Section III-D). Therefore, with root reuse, varying $T_{DC}$ or $T_{AC}$ does not affect its execution, as confirmed by the flat line. Because CACHEIT handles this reuse pattern differently, we evaluate it separately in Section IV-D.

### D. Impact of root reuse

Table IV reports the execution time of TC using Large input with two versions of CACHEIT where root RDD caching (cf. §III-D) is disabled and enabled respectively. Across all heap sizes, enabling root RDD caching speeds up the execution of TC by 24% to 30% compared to when disabling this feature. Other programs' performance are unaffected because they do not have this pattern.

Table IV: Execution time of TC and performance gain by enabling CACHEIT's Root Reuse.

| **Heap configs. (GB)** | | 1 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| Exec. time (*sec*) w/ Root Reuse | Dsbl | 320 | 306 | 306 | 317 | 323 | 319 |
| | Enbl | 258 | 244 | 247 | 255 | 250 | 245 |
| ***Speedup*** **(×)** | | 1.24 | 1.25 | 1.24 | 1.24 | 1.29 | 1.30 |

## V. RELATED WORK

**Garbage Collection** Big Data applications often run atop a managed runtime to take advantages of the automatic memory management. GC is one of major sources of the runtime cost. Many modern GCs have been proposed, such as G1 [25], ZGC [34], and Shenandoah [35], offering short pauses and high throughput. Recent years have seen many GC algorithms adapting to modern data-intensive workloads. Panthera [36] and Espresso [37] are GCs that incorporate the characteristics of non-volatile memory. Taurus [38] cordinates GC runs across nodes in distributed systems to hide GC latency. Yak [22] divides the heap memory into two different spaces and manages them differently to adapt to their object characteristics. Semeru [39] and Mako [40] are developed for disaggregated memory, an exciting trend towards datacenter design and resource management. They offload/share GC workload to remote machines with weak compute power. These GCs cannot

solve this data caching problem. In Spark, RDDs have a short, predefined lifetime. As such, RDDs are collected after a task is completed, unless being manually cached.

**Prefetcher and Replacement Policies for Memory Cache** Previous studies are exploring alternative replacement policies in Spark [14, 28, 41]. Instead of the default LRU, these policies consider a combination of data characteristics such as partition size, computation time, or distance between uses to remove data when cache memory is reaching its capacity. These policies are complements and can be adapted into our work as we already extract these metadata. Other works such as MRD [13] and MEMTUNE [9] also provide prefetching solutions on a hierarchy of data storage to hide the re-computation latency of RDDs. Unlike us, these approaches operate only on the user-defined RDDs and they are postmortem. Similar to us, Neutrino [10] is a runtime solution but focuses on moving cached data between different caching levels in Spark. It still relies on users to determine a set of caching RDDs upfront, a task that we aim to eliminate. Orthogonal to us, ATuMm [16] dynamically tunes the JVM configuration to adjust the amount of memory available for execution and caching.

**Spark Program Tuning Supports** There are many works that aim to mitigate manual inconvenience for Spark developers. Among them, Ruya [42] searches for the best cluster configuration (i.e., number of workers, heap sizes, number of cores) within a restricted search space. Blaze [43] collects execution information and uses Integer Linear Programming (ILP) to optimally decide how each RDD should be recovered for reuse (i.e., by recomputing or storing and loading from disk). SparkCAD [8] and MCR [12] are closer to our work. SparkCAD is a profiling-based RDD lineage visualization tool that suggests developers which RDD should be manually cached. MCR automates RDD caching by greedily selecting RDDs using profiling information. Compared to these works, CACHEIT does not require profiling, and is guided by the DAG of RDD dependency which is native to Spark engine.

**Program Analyses to Remove Software Bloat** The term bloat refers to inefficiencies in software such as redundant computations to create the same data. Static [44, 45] and dynamic analyses [46–48] are popular approaches in detecting these inefficiencies. While static analysis is cheap but cannot scale to large codebases, dynamic analyses are precise but have high time and space overheads. Existing works also do not operate on the coarse granularity such as RDD. CACHEIT is a dynamic approach with minimal overheads: intra-job analysis considers static workflows as they are submitted to the scheduler, and inter-job analysis considers dynamic information of past jobs to precisely detect candidates.

## VI. CONCLUSIONS

In this paper, we introduce CACHEIT, a runtime support for Apache Spark framework. CACHEIT releases users from the difficult responsibility of enforcing caching decisions at compile time for all programs. CACHEIT automatically identifies high-value RDDs, leveraging dependencies and access patterns in the workflows. The experimental results are positive, showing that CACHEIT is effective in detecting appropriate data for caching, thereby having only marginal slowdown compared to manually-tuned counterparts while requiring minimal user effort. This shows promises in enabling autonomous data caching not only for Spark but any DAG-based dataflow frameworks.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback in improving this paper. This work is supported by NSF CNS-2107010 and the Indonesian Ministry of Education, Culture, Research and Technology.

## REFERENCES

[1] Apache, "Spark: Unified engine for large-scale data analytics." https://spark.apache.org, 2024.
[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud*, June 2010.
[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, pp. 137–150, 2004.
[4] Apache, "Apache Flink." http://flink.apache.org, 2024.
[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 *SIGMOD*, pp. 1383–1394, 2015.
[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 *OSDI*, p. 599–613, 2014.
[7] H. Li, D. Wang, T. Huang, Y. Gao, W. Dou, L. Xu, W. Wang, J. Wei, and H. Zhong, "Detecting cache-related bugs in Spark applications," in *ISSTA*, p. 363–375, 2020.
[8] H. Al-Sayeh, M. A. Jibril, M. W. B. Saeed, and K.-U. Sattler, "SparkCAD: caching anomalies detector for Spark applications," *Proc. VLDB Endow.*, vol. 15, p. 3694–3697, aug 2022.
[9] 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 *IPDPS*, pp. 383–392, 2016.
[10] E. Xu, M. Saxena, and L. Chiu, "Neutrino: Revisiting memory caching for iterative data analytics," in *HotStorage*, (Denver, CO), June 2016.
[11] S. Park, M. Jeong, and H. Han, "CCA: Cost-capacity-aware caching for in-memory data analytics frameworks," *Sensors*, vol. 21, no. 7, 2021.
[12] A. Nasu, K. Yoneo, M. Okita, and F. Ino, "Transparent in-memory cache management in Apache Spark based on post-mortem analysis," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 3388–3396, 2019.
[13] T. B. G. Perez, X. Zhou, and D. Cheng, "Reference-distance eviction and prefetching for cache management in spark," in *ICPP*, 2018.
[14] Y. Yu, W. Wang, J. Zhang, and K. Ben Letaief, "LRC: Dependency-aware cache management for data analytics

clusters," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9, IEEE, 2017.
[15] M. J. Mior and K. Salem, "ReSpark: Automatic caching for iterative applications in Apache Spark," *2020 IEEE International Conference on Big Data (Big Data)*, pp. 331–340, 2020.
[16] D. Jia, J. Bhimani, S. N. Nguyen, B. Sheng, and N. Mi, "ATuMm: Auto-tuning memory manager in Apache Spark," *IPCCC*, pp. 1–8, 2019.
[17] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie, "Titan: data provenance support in Spark," *Proc. VLDB Endow.*, vol. 9, no. 3, p. 216–227, 2015.
[18] V. M. Gottin, E. Pacheco, J. Dias, A. E. M. Ciarlini, B. Costa, W. Vieira, Y. M. Souto, P. Pires, F. Porto, and J. a. G. Rittmeyer, "Automatic caching decision for scientific dataflow execution in Apache Spark," in *BeyondMR*, 2018.
[19] H. Chao-Qiang, Y. Shu-Qiang, T. Jian-Chao, and Y. Zhou, "RDDShare: Reusing results of Spark RDD," in *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, pp. 370–375, 2016.
[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.
[21] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "FACADE: A compiler and runtime for (almost) object-bounded big data applications," in *ASPLOS*, pp. 675–690, 2015.
[22] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A high-performance big-data-friendly garbage collector," in *OSDI*, pp. 349—365, 2016.
[23] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 936–947, 2016.
[24] The KONECT Project, "KONECT networks datasets," 2013. http://konect.cc/networks/.
[25] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *ISMM*, pp. 37–48, 2004.
[26] "Soot: a Java optimization framework." https://github.com/soot-oss/soot.
[27] V. Raychev, M. Musuvathi, and T. Mytkowicz, "Parallelizing user-defined aggregations using symbolic execution," in *SOSP*, pp. 153–167, 2015.
[28] H. Li, S. Ji, H. Zhong, W. Wang, L. Xu, Z. Tang, J. Wei, and T. Huang, "LPW: an efficient data-aware cache replacement strategy for Apache Spark," *Science China Information Sciences*, 2023.
[29] "HiBench." https://github.com/intel-bigdata/hibench.
[30] J. McAuley, "Amazon product data," 2014. https://cseweb.ucsd.edu/~jmcauley/datasets/amazon/links.html.
[31] J. L. Tianyang Zhang, "Google local data," 2021. https://datarepo.eng.ucsd.edu/mcauley_group/gdrive/googlelocal.
[32] D. Wong, "Flightprices." https://github.com/dilwong/FlightPrices, 2022.
[33] Rong-En Fan, "Libsvm data: Classification, regression, and multi-label," 2011. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.
[34] Oracle, "The Z garbage collector." https://wiki.openjdk.java.net/display/zgc/Main, 2019.
[35] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK," in *PPPJ*, 2016.
[36] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "Panthera: Holistic memory management for big data processing over hybrid memories," in *PLDI*, pp. 347–362, 2019.
[37] M. Wu, Z. Ziming, L. Haoyu, L. Heting, C. Haibo, Z. binyu, and G. Haibing, "Espresso: Brewing Java for more non-volatility," in *ASPLOS*, 2018.
[38] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz, "Taurus: A holistic language runtime system for coordinating distributed managed-language applications," in *ASPLOS*, pp. 457–471, 2016.
[39] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A memory-disaggregated managed runtime," in *OSDI*, pp. 261—280, 2020.
[40] H. Ma, C. Wang, Y. Qiao, and M. Kim, "Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters," in *PLDI*, pp. 92–107, 2022.
[41] Y. Geng, X. Shi, C. Pei, H. Jin, and W. Jiang, "LCS: An efficient data eviction strategy for Spark," *International Journal of Parallel Programming*, vol. 45, pp. 1285–1297, 12 2017.
[42] J. Will, L. Thamsen, J. Bader, D. Scheinert, and O. Kao, "Ruya: Memory-aware iterative optimization of cluster configurations for big data processing," *2022 IEEE International Conference on Big Data (Big Data)*, pp. 161–169, 2022.
[43] W. W. Song, J. Eo, T. Um, M. Jeon, and B.-G. Chun, "Blaze: Holistic caching for iterative data processing," in *EuroSys*, p. 370–386, 2024.
[44] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *ISSTA*, pp. 118–128, 2007.
[45] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," in *FSE*, pp. 59–70, 2008.
[46] K. Nguyen and G. Xu, "Cachetor: detecting cacheable data to remove bloat," in *FSE*, pp. 268–278, 2013.
[47] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky, "Finding low-utility data structures," in *PLDI*, pp. 174–186, 2010.
[48] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *PLDI*, pp. 419–430, 2009.