ReSpark: Automatic Caching for Iterative Applications in Apache Spark

Michael J. Mior
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
mmior@cs.rit.edu

Kenneth Salem
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
kmsalem@uwaterloo.ca

***Abstract*—Apache Spark is a distributed computing framework used for big data processing. A common pattern in many Spark applications is to iteratively evolve a dataset until reaching some user-specified convergence condition. Unfortunately, some aspects of Spark's execution model make it difficult for developers who are not familiar with the implementation-level details of Spark to write efficient iterative programs.**

Since results are constructed iteratively and results from previous iterations may be used multiple times, effective use of caching is necessary to avoid recomputing intermediate results. Currently, developers of Spark applications must manually indicate which intermediate results should be cached. We present a method for using metadata already captured by Spark to automate caching decisions for many Spark programs. We show how this allows Spark applications to benefit from caching without the need for manual caching annotations.

***Index Terms*—Spark, caching, iteration**

I. Motivation

Apache Spark [1] is a widely used framework for big data analytics, machine learning, and many other domains. Spark has an advantage over previous solutions like MapReduce [2] in that intermediate results can be held in memory without the need to write results to disk at each stage. This makes Spark well-suited for iterative applications. In these applications, it is common for intermediate results to be reused in several iterations. However, Spark does not automatically cache intermediate results. To guide caching, application programmers must include explicit annotations in the their programs to tell Spark which intermediate results to add to or remove from its cache. Unfortunately, Spark's caching annotations can be surprisingly difficult to use effectively. Although the annotations are simple, they can sometimes interact with Spark's lazy job execution semantics in unexpected ways. We illustrate some of these problems in Section III.

In this paper, we present ReSpark. The goal of ReSpark is to to eliminate the need for explicit caching annotations in Spark applications. Instead, ReSpark automatically determines what should be added to and removed from the cache on behalf of the program. Our focus is specifically on iterative applications since iteration is the primary motivator for caching in Spark.

We take a learning approach to automating caching decisions. As an application runs, ReSpark observes how intermediate results are used by the application. It then uses these observations to predict how subsequent intermediate results will be used, and bases its caching recommendations on those predictions. Unlike prior work, these observations and adjustments are made during a single run of the application, without requiring prior training data. We tested the performance of ReSpark using a suite of Spark benchmarks and found that the performance of many applications, using our enhancements and with no manual caching annotations, was comparable to that of manually annotated versions.

Our work makes the following contributions:

- In Section IV, we introduce *ReSpark*, a series of modifications to the Spark runtime which automate the selection of intermediate results to persist.
- Section IV-C presents *lazy unpersist*, an alternative to the default mechanism of unpersisting cache entries in Spark which ensures that cached results are fully utilized before they are removed from the cache.
- Finally, in Section V we analyze the performance of our techniques and demonstrate that explicit persist/unpersist annotations by Spark application developers are not required to achieve good performance.

II. Spark Background

In the following section, we discuss the challenges faced by developers making use of Spark for iterative computation. Section II-A describes the types of applications we are aiming to optimize and the problems developers face with their implementation. In Section II-B we show how this problem is partially addressed via caching of intermediate results, but we also show how simple approaches to caching fall short.

A. Iterative Computation in Spark

A simple example of a Spark application using iteration is given in Figure 1a. The program consists of a series of lazy `map` transformations followed by a `count` action (which forces evaluation). Since Spark uses lazy evaluation and there are no actions within the loop, Spark simply constructs a directed acyclic graph (DAG) of these transformations while the loop runs. Each transformation consumes *resilient distributed datasets* (RDDs) produced by previous transformations, and produces a new RDD for consumption downstream. As the application iterates, Spark simply records metadata about how

 DOI: 10.1109/BigData50022.2020.9377866

to compute the RDD generated by transformations in one iteration from the RDDs generated during the previous iteration. This metadata defines the *lineage* of the RDD. An action such as the `count` on the last line of the program schedules a job which immediately executes all of the pending transformations.

Since the example program consists of a single action and no aggregation, all transformations are executed as a single job, resulting in consistently fast execution as shown in Figure 2, Variant V1. All benchmarks in this figure were executed on a data set of 100 million randomly generated points using the same hardware described later in Section V. Spark was configured with 4 executors each with 16GB of memory and an additional 32GB of memory allocated to the driver program.

```
1  // data refers to a Spark RDD
2  for (i <- 1 to iterations) {
3    data = data.map(...).filter(...)}
4  data.count()
```

(a) Variant V1: Simple Spark application

```
1  // data refers to a Spark RDD
2  for (i <- 1 to iterations) {
3    data = data.map(...).filter(...)
4    if (data.count() < limit) break }
5  data.count()
```

(b) Variant V2: With data-dependent termination

```
1  // data refers to a Spark RDD
2  for (i <- 1 to iterations) {
3    val oldData = data
4    data = data.map(...).filter(...)
5    data.persist()
6    if (data.count() < limit) break
7    oldData.unpersist() }
8  data.count()
```

(c) Variant V3: With data-dependent termination and caching

Fig. 1: Variants of an iterative Spark application

Unfortunately, small changes to the application code can result in a significant degradation of application performance. Consider Figure 1b, which shows a variant of the sample program from Figure 1a. In this version, the developer wants a data-defined termination condition: stopping when the size of the RDD drops below a specified value. An action (`count`) is now executed for each iteration. Because of this, execution of the application will result in one Spark job per iteration, rather than a single job as was the case for variant V1. Computation of the current count for the `data` RDD will be scheduled on each iteration of the loop. In addition to the transformations no longer being lazy, Spark will also reevaluate all the transformations for the previous iteration since results are discarded after each iteration.

The seemingly minor change from variant V1 to variant V2 leads to substantial change in execution time as shown in Figure 2. The result is quadratic runtime instead of the linear runtime that would be achieved without the intervening action. This can be resolved through the use of Spark's caching mechanisms, but this requires explicit programmer annotation.

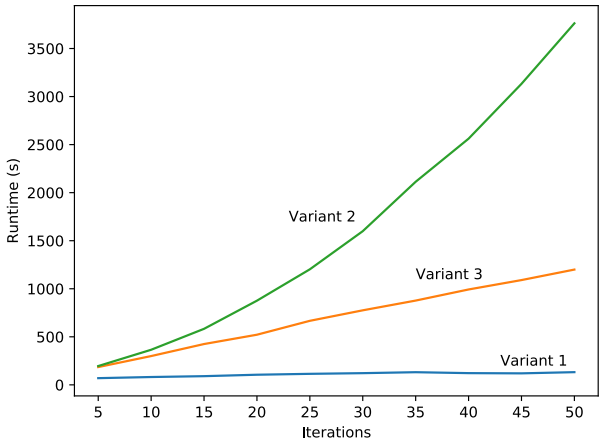

Fig. 2: Runtime for the Spark application from Figure 1

B. Caching

Spark programmers are expected to explicitly indicate when computed RDDs should be persisted. When writing a Spark application, a developer can store a reference to any RDD in a variable in the host programming language, such as the variable `data` in our sample program. This enables datasets constructed from the same set of transformations to be used multiple times with other additional transformations appended. By default, Spark does not attempt to reuse computed RDDs even when their lineage is identical. A persistence annotation in Spark specifies that the annotated RDD should be stored for later reuse. When the RDD is first computed, it is placed into an internal cache in either memory or disk as specified by the programmer. If the same RDD is reused in a different job, the executor running the job will check its local cache. If the RDD was previously computed and has not been evicted, the RDD will be read from the cache instead of being recomputed.

Thus, to improve variant V2, a developer could choose to persist the results of each iteration as in Figure 1c. With this explicit call to `persist`, results of each iteration will be cached in memory. Therefore, when the next iteration is computed, this data can be retrieved from the cache instead of recomputing it. Information which is no longer required can be removed from the cache via an explicit call to `unpersist` (line 7 in Figure 1c). The addition of these persistence annotations returns the application to linear runtime (with some additional overhead required for the action to be computed in each iteration compared to variant V1). The effect of enabling caching in the sample program is shown in Figure 2.

III. The Problem

Although `persist` and `unpersist` annotations are important to the performance of iterative applications, they

```
1  var rankGraph = graph.joinVerts(...)
                        .map(...)
2  var iteration = 0
3  while (iteration < numIter) {
4    rankGraph.persist()
5    val updates = rankGraph.aggregate(...)
6    prevRankGraph = rankGraph
7    rankGraph = rankGraph.joinVerts(updates)
                          .persist()
     // see the text
8    // rankGraph.foreachPartition(...)
9    prevRankGraph.unpersist() }
10 rankGraph.values.sum()
```

Fig. 3: Simplified Version of PageRank in Spark

Spark uses two RDDs for graphs; we treat them as one for simplicity.

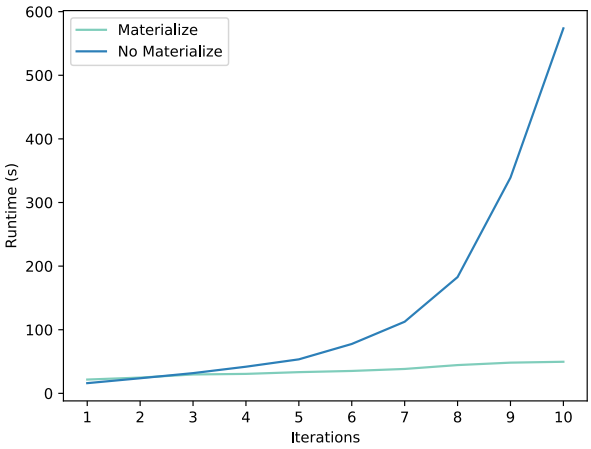

Fig. 4: PageRank runtime with/without eager materialization

can be difficult to apply correctly. For example, consider Figure 3, which shows a simplified version of Spark's PageRank implementation. This code starts with a graph (`rankGraph`) and iteratively refines it. On each iteration, the newly refined graph is persisted (line 4) and the previous version of the graph is unpersisted (line 9). PageRank's iterative refinement pattern is similar to that used by the application in Figure 1, and both applications make similar use of `persist` and `unpersist` annotations to control caching. However, while caching was effective for the application in Figure 1c, it will not help at all for the PageRank implementation in Figure 3!

In Figure 3, the programmer's intention is clearly to persist the new graph and unpersist the old graph on each iteration. However, the `persist` and `unpersist` annotations will not have the intended effect because, as described in Section II, Spark transformations are executed lazily. That is, Spark does not actually compute the nth version of the graph during the nth iteration of the loop. Spark's `persist` and `unpersist` annotations, in contrast, only affect the metadata, and they are applied eagerly as the application loop runs. Thus, on each iteration in Figure 3, the current version of the graph is flagged to be persisted, and then the flag is removed by the `unpersist` annotation. When lazy evaluation finally occurs (at line 10 in Figure 3), none of the graph versions have persistence flags, and Spark does no caching at all!

To recognize this problem, a Spark programmer must understand the non-intuitive interplay between Spark's persistence annotations and lazy evaluation. Solving the problem is also non-trivial. In the case of Spark's PageRank implementation, per-iteration caching is achieved by adding a gratuitous Spark action to each loop iteration to force Spark to eagerly evaluate each new graph version. (This is accomplished by the `foreachPartition` action which we have shown in a comment at line 8 of Figure 3.) A side effect of this materialization is that the `persist` and `unpersist` annotations now have their intended behavior.

Figure 4 shows the performance of Spark's PageRank implementation with and without the eager materialization induced by the `foreachPartition` action. (The full experimental setup is described in Section V.) Without eager materialization, runtime grows quadratically with the number of iterations, as shown by the "No Materialize" curve in Figure 4. This is because caching is ineffective, forcing Spark to recalculate previous versions of the graph on each iteration. With the action, runtime grows linearly (curve "Materialize"), since each graph version is cached until the next version has been calculated.

In summary, caching is critical for the performance of iterative Spark applications. However, Spark programmers must manually annotate their applications to guide caching. To make effective use of such annotations, the program must be able to recognize reuse of Spark RDDs by the application, and must mark reused RDDs for caching after they are defined by the application but before they are actually evaluated by the Spark runtime. As application complexity increases, identifying and tracking potential reuse becomes more challenging. Even if the programmer can identify reused RDDs, annotating the application for effective caching is challenging because of the complex interplay between application execution, caching, and lazy evaluation in Spark. With ReSpark, we avoid these problems by eliminating the need for manual annotation.

IV. ReSpark

ReSpark works by observing the application as it runs, and generating persistence annotations on the fly. That is, it *learns* how to cache RDDs by watching how they are defined and used. ReSpark also learns when to unpersist persisted RDDs. Its approach is general enough to accommodate applications with complex iteration patterns, such as nested loops. A ReSpark program should outperform an alternative without caching, while being simpler to write than a program with manual caching annotations. ReSpark is specifically designed to improve the performance of iterative applications. No changes to the Spark programming interface are required to use ReSpark except that Spark programmers can avoid the use of explicit persist and unpersist annotations.

```
1  var next = sc.parallelize(1 to 100)
               .map(i => (i,i))
2  var prev: RDD[(Int, Int)] = null
3  var n = 0
4  while (next.sum() < MIN_SUM) {
5    val updates = next.map({ case (i, j) =>
                                (i, j + 1) })
6    prev = next
7    next = prev.join(updates)
8               .map({ case (i, (j, k)) =>
                         (i, j + k) })
9  }
10 next.count()
```

Fig. 5: A Spark program benefiting from caching with ReSpark

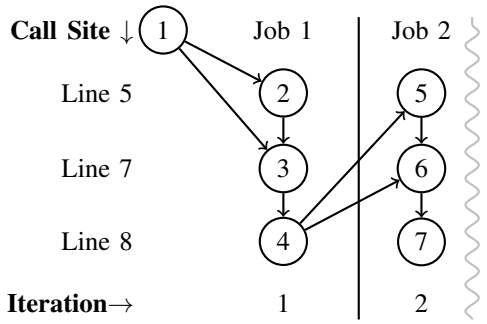

Fig. 6: Partial execution of the program in Figure 5

A. Predicting Reuse: An Example

Consider the example in Figure 5 which uses iteration to repeatedly generate new RDDs in a loop. In this case, the program sums up values in the RDD to decide when to terminate. This program will generate one Spark job for each iteration of the loop. The RDDs generated in the first job are shown in Figure 6 (consider only those to the left of the line indicating the job boundary). ReSpark makes use of the lineage information already maintained by Spark to predict reuse. As new RDDs are defined, Spark records their parents. An arrow from an RDD 1 to another RDD 2 indicates that RDD 2 is computed based on data from RDD 1 (i.e., 1 is a parent of 2). Until an action (e.g. `count`) occurs in the application, Spark simply continues to build up this metadata. Any RDD which is used more than once is a candidate for caching.

ReSpark must make caching decisions for RDDs computed by a job before the job is executed since caching directives are metadata attached to RDDs computed by a job when the job is submitted. Using only metadata from the first job, ReSpark would choose to cache RDD 1 since it is used twice (for computing RDDs 2 and 3). However, other RDDs computed by this job may be reused in a job later in the application program that we currently have no information on. Since job execution starts whenever the Spark runtime encounters an action, we must decide if RDDs that will be generated during the execution of that job should be cached. When executing job 1 for iteration 1, we can see reuse of RDD 1, but we have no knowledge of the future reuse of RDD 4. As shown in Figure 6, we can see that once job 2 has executed, RDD 4 will have been used twice (for defining RDDs 5 and 6). We refer to this problem of predicting which RDDs will be reused in future jobs as the *job horizon problem* since reuse beyond the current job is not visible.

Consider what happens when we prepare to execute subsequent jobs. Since the dependency structure of RDDs is tied to the structure of the application program, we may decide to infer that RDDs 4 and 7 will see similar reuse patterns since they were created at the same place in the application code. Predicting reuse in this way is core to ReSpark's automation of caching.

B. Predicting Reuse in ReSpark

One key piece of information (already maintained by Spark) useful for analyzing reuse is the *call site* of each RDD. The call site is simply a stack trace of the program, which uniquely identifies the point in the application where the RDD is defined. RDDs may be used at a call site in one of two ways: as input to a transformation to produce another RDD or in an action to compute a value from the RDD. By counting transformations and actions for an RDD, we can determine when reuse occurs for RDDs with a given call site.

We formulate our solution for making appropriate caching decisions in the presence of the job horizon problem as a prediction task. Specifically, we enable Spark to learn whether an RDD is expected to be reused given past behaviour. Since we have call site information for each RDD, we use this to predict whether new RDDs with the same call site will be reused and to decide whether they should be persisted. When a new RDD is defined, we need to decide if the new RDD should be persisted. Call site information is useful since we expect that RDDs defined at the same call site will experience similar reuse patterns. This is a key insight used by ReSpark.

In ReSpark, we use a simple approach to solving the problem of predicting the number of times each RDDs will be used. For each call site, we maintain a history of the RDDs created at that call site. For each of these RDDs, we track every call site where the RDD is used. We can predict the number of times future RDDs created at the same call site will be used and decide if they should be cached. That is, we expect that the number of uses of an RDD created at a particular call site is predictive of the number of uses of future RDDs defined at the same call site. Whenever the predicted number of uses of an RDD is greater than one, ReSpark will choose to cache the RDD. The expected number of uses also enables ReSpark to determine when an RDD is no longer needed and can be unpersisted (removed from the cache) as we discuss in the next section. The full set of algorithms used to update usage information is given in Figure 7.

ReSpark assumes that the reuse of the first RDD created at a call site is predictive of the reuse of all future RDDs created at the same call site. This is currently a limitation of ReSpark that

```
Procedure OnRDDDefinition(rdd)
    Add rdd to the list of RDDs created at rdd.callSite
    // Record each dependency of the new RDD
    foreach dependency dep of rdd do
        Add the rdd.callSite to the list of uses of dep
        if The first RDD created at dep.callSite has a use
          count greater than 1 then Persist dep;
    if call site of rdd has a use count greater than 1 then
        Persist rdd
Procedure OnRDDAction(rdd)
    Add the call site of the action to the list of uses ofrdd
```

Fig. 7: Algorithms for tracking RDD usage

we intend to address in future work. However, this assumption is sufficient for many useful algorithms which we analyze in Section V. To enable ReSpark to work with applications with changing reuse patterns, we expect to be able to use the same usage information we are currently collecting in concert with more advanced machine learning techniques to make more complex predictions. If ReSpark were to underestimate reuse, it is possible that cached data will still be available since it is lazily removed from the cache.

For an example of our simple approach to reuse tracking, in Figure 6 job 2 has executed and we see that RDD 4, the first RDD created on line 8, was used twice. The first use in defining RDD 5 on line 5 and again in defining RDD 6 on line 7. Since the expected number of uses is greater than one, ReSpark will then choose to cache any future RDDs created on line 8 of the program. When preparing future jobs, ReSpark is able to correctly predict that the RDD which will be created on Line 8, will be reused when the next job executes.

Since the focus of ReSpark is on iterative applications, one might wonder why we do not make use of information on iteration (i.e. the structure of loops in the application program). In fact, an early implementation of ReSpark did provide explicit information on iteration to the Spark runtime. However, the major drawback of this approach is that it requires annotating Spark programs with information about loops since Spark is oblivious to iteration, which occurs inside the application program. This could potentially be remedied via static analysis of Spark applications to identify iterations. Even so, we found that the performance of ReSpark did not benefit from this additional information so we have chosen an approach which works easily on unmodified applications.

C. Unpersisting RDDs

In addition to automatically determining what to persist, ReSpark also automatically determines when to unpersist RDDs it has previously persisted. ReSpark unpersists RDDs lazily, during evaluation, once RDDs have been used the predicted number of times. This avoids polluting the cache with RDDs which are not expected to be used in the future. To do this, we make use of metadata, `reuseCount` which ReSpark associates with each RDD it decides to automatically persist. The `reuseCount` indicates how many times an RDD is expected to be reused as predicted in Section IV-B.

ReSpark also sets a flag, `unpersistPending`, on each RDD it decides to automatically persist so that it does not unpersist RDDs which were explicitly persisted by the application. To decide when an RDD should be unpersisted, we simply need to track when each use occurs and decrement the `reuseCount`. When the `reuseCount` of an RDD reaches zero, then it can be unpersisted since we do not expect future use, enabling the space in the cache used by the RDD to be freed immediately. This approach avoids explicit materialization as discussed in Section II-B since we ensure the Spark runtime does not unpersist an RDD when it is expected to be reused. Note that if the RDD is not used the predicted number of times (i.e. the reuse count was overestimated), it will still eventually age out of Spark's LRU cache.

The decision as to when an RDD should be unpersisted is made using the expected usage information collected as described in the previous section. What remains is to count when each of these expected uses occurs. To track each use of an RDD, ReSpark ties each usage to an execution of a Spark *stage*. Stages are created by the Spark scheduler when an action executes to compute the result of actions or repartition data for transformations such as aggregation or joins. The algorithm used by ReSpark for discovering which RDDs are used is executed when each stage is scheduled (e.g., when an action occurs in the application program). ReSpark then traverses the lineage of the RDD computed by that stage. During this traversal, ReSpark records RDDs marked with the `unpersistPending` flag, which indicates RDDs that ReSpark has previously persisted. These are RDDs which should be unpersisted once their `reuseCount` reaches zero.

The goal of this traversal algorithm is to discover which of these RDDs are used by each stage Spark has scheduled. ReSpark maintains a simple data structure, `waitingStages`, during this traveral. It indicates which stages make use of an RDD. These are the stages which must complete for the RDD to be unpersisted. Details of the algorithm used to populate this structure given in Figure 8. Note that the highlighted line is an optimization which we describe later.

Once the `waitingStages` structure has been populated by the algorithm in Figure 8, we can use this information during job execution to decide when RDDs can be unpersisted. As each stage completes, the algorithm checks to see if there are any RDDs that may be ready to be unpersisted. If the stage which just completed corresponds to the last use of an RDD, then that RDD is unpersisted. This approach ensures that RDDs are persisted when requested and allows them to be unpersisted as soon when they are no longer needed. The full algorithm is shown in Figure 9.

Some intermediate stages, which Spark refers to as *shuffle* stages, require sending data between nodes in a Spark cluster. Shuffle stages can be expensive to recompute. To mitigate this cost, Spark stores the output of shuffle stages to disk in case it is used again. When an RDD which is the result of a shuffle is reused, we only need to wait for the shuffle to complete since Spark can use the shuffle output stored on disk instead of recomputing the RDD. Exploiting this behaviour of shuffle

```
Procedure OnResultStageScheduled(finalStage)
  waiting ← [(finalStage.rdd,
   finalStage.id)]
  Procedure visit(rdd, stageId)
    // Record RDD to be unpersisted later
    if rdd.unpersistPending then
      Add stageId to waitingStages for rdd

    // Mark ancestors to be visited
    foreach dependency dep of rdd do
      if dep is a shuffle dependency then
        Add (dep, dep.stageId) to waiting
      else
        Add (dep, stageId) to waiting

  // Visit stages following RDD lineage
  visited ← ∅
  do
    Pop (rdd, stageId) from waiting
    if (rdd, stageId) ∉ visited then
      visit(rdd, stageId)
      Add (rdd, stageId) to visited
  until waiting is empty;
```

Fig. 8: Finding stages to unpersist an RDD

```
Procedure OnStageFinish(stage)
  // Check for RDDs to unpersist
  foreach rdd in waitingStages for stage do
    Remove stage from the list of stages for rdd in
     waitingStages
    Decrement reuseCount for rdd
    if waitingStages for rdd is empty and
     reuseCount is zero then Unpersist rdd;
```

Fig. 9: Check for RDDs which can be unpersisted

stages is an optimization which allows ReSpark to unpersist data earlier since it can be provided by Spark's shuffle output on disk. We highlight this optimization in Figure 8 which simply consists of waiting for the stage preceding a shuffle to complete instead of the stage itself.

We can use the information collected in the `waitingStages` structure for one final optimization. Spark already uses delay scheduling [3] to postpone the execution of tasks in exchange for increased locality. This includes attempting to schedule tasks where data has been cached. However, this locality-based scheduling does not take into account tasks which are in progress and have not yet been cached. We use the information on pending tasks to extend this locality-based scheduling so Spark also attempts to colocate a new task with a running task which is already computing data that the new task requires. This locality-based scheduling increases the chances that any data placed in the cache by the running task will be available to the new task.

V. Evaluation

To evaluate the performance of ReSpark, we analyzed six iterative Spark applications, as summarized in Table I. The first four applications (shortest paths, PageRank, K-means, and SCC) are from version 2.0 of the Spark-Bench [4] benchmarking suite. These specific algorithms were selected since they comprise all iterative applications within Spark-Bench 2.0. These applications were created to test implementations of iterative algorithms that are implemented in the MLlib and GraphX machine learning and graph processing libraries that are distributed with Spark. We expect these algorithms to be highly optimized since they are implemented by the same team of developers working on the core of Spark. In particular, the library implementations of these algorithms all make use of developer-specified RDD caching and, in some cases, RDD materialization (see Table I). The input data used for these benchmarks was produced by the random data generation facilities of Spark-Bench.

Application	Data-dependent termination	Loop Nesting	Materialization
Shortest paths	✓		
PageRank			✓
K-means	✓		
SCC		✓	✓
LOPQ		✓	
BigITQ			

TABLE I: Summary of evaluated Spark applications

In addition to the Spark-Bench applications, we considered two other iterative applications for which we were able to obtain test data. The first, called LOPQ, implements locally optimized product quantization [5], an approximate nearest neighbour algorithm. LOPQ is a Spark implementation of the algorithm from developers at Yahoo! [6] LOPQ trains multiple k-means models on different splits of the data. The second, called BigITQ [7], is a Spark implementation of iterative quantization (ITQ) [8], an algorithm for producing similarity-preserving binary codes for large image collections. The ITQ algorithm constructs a matrix representing these codes which is iteratively updated to produce the final result. We ran this algorithm using the CIFAR-10 [9] dataset. Both BigITQ and LOPQ include developer-specified Spark caching annotations.

The primary goal of our evaluation is to compare the effectiveness of ReSpark's automatic caching annotations with that of developer-specified caching. To do this, we compare the performance of manually and automatically annotated versions of the applications. Ideally, we hope that the automatically annotated versions will perform as well as the manually annotated versions, indicating that ReSpark can eliminate the need for manual annotation, with no sacrifice in application performance.

To this end, we report the performance of each application executed three different ways, which we refer to as **ReSpark**, **Default**, and **NoCache**. For **ReSpark** execution, we remove all developer-added caching annotations and materialization from the application. We then run the resulting application using ReSpark. For **Default** execution, we ran the unmodified

applications, including all developer-specified caching annotations and materialization, on unmodified Spark. This serves as our primary baseline. For **NoCache** execution, we ran the original application with caching disabled to show the impact of caching on application execution times.

All of our experiments were run on a Spark cluster using the Hadoop distributed file system (HDFS) and Hadoop's YARN resource manager. All experiments were performed with a pre-release of Spark 2.4.0 using YARN running on Hadoop 2.7.3. Each server has two six core Xeon E5-2620 processors operating at 2.10 GHz and 64 GB of memory. For the experiments involving the SparkBench application, we used three servers for each experiment. One server ran the Spark master, the HDFS NameNode, and the YARN resource manager. The other two hosted HDFS data nodes, YARN node managers and Spark executors. Both the Spark driver and each of two Spark executors were allocated 4GB of memory. For the experiments involving BigITQ and LOPQ, for which we had larger test data sets, we used a similar configuration with four Spark execution nodes. Our primary performance metric in each experiment is the total runtime of the test application. All figures report the average of ten runs along with the 95% confidence interval. (Note that in many cases, the confidence interval is too small to be represented in the graph.)

In the remainder of this section, we present the results of our evaluation for each of the test applications in turn. To facilitate our analyses of the results, we present high-level summaries of the iterative structure of each application. These summaries take the form of graphs, such as the one shown in Figure 10. Each small labelled box represents the call site of an RDD in the application. The arrows between these boxes represents data dependencies between RDDs. That is, if an edge exists from RDD A to RDD B, RDD B is created based on some transformation of RDD A. RDDs can also depend on other RDDs created at the same call site on a previous iteration. These RDDs are represented by a similarly-named RDD surrounded by a dotted box. For example, in Figure 10, each instance of the Centers RDD is created from the previous instance of Centers as well as the Input RDD.

A. K-means clustering

K-means in Spark is part of the MLlib machine learning package and uses the parallel k-means clustering algorithm [10]. After initializing cluster centers, the algorithm iterates over the dataset moving each cluster center closer to its mean. Cluster centers are then recomputed until they converge or a maximum number of iterations (5) is reached. The structure of the algorithm is shown in Figure 10. The majority of the runtime in k-means clustering is iterating over the input to refine the cluster centers. Results of the baseline without caching in Figure 11 show that caching is critical for good performance. ReSpark is able to identify that the input to the algorithm should be cached and achieves nearly identical performance to the original program with manual persistence annotations.

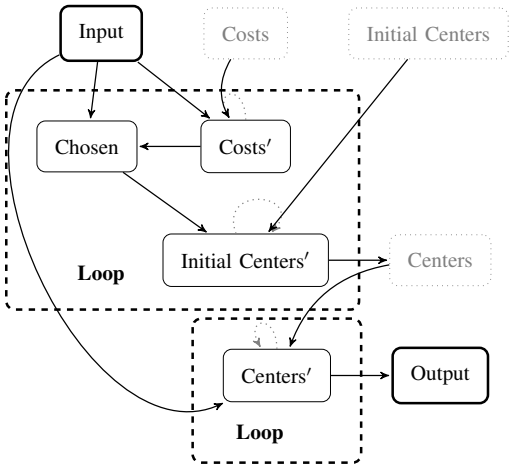

Fig. 10: Spark's K-means implementation

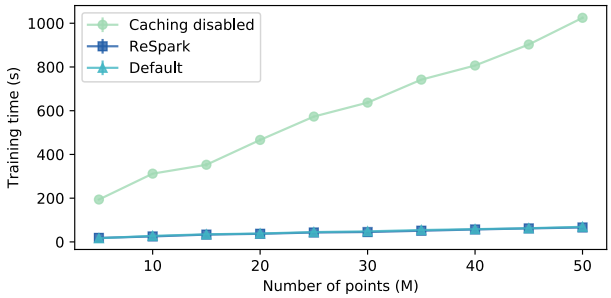

Fig. 11: K-means clustering benchmark results

B. Shortest paths

The shortest paths algorithm is part of Spark's GraphX graph processing library, which uses an approach similar to the Pregel [11] graph processing system. Spark's Pregel implementation simply iterates sending messages between vertices which update their local data. Only vertices which receive messages are permitted to send messages in subsequent rounds. Execution stops when all vertices have stopped sending messages. The shortest paths algorithm in Spark is formulated as a vertex program using Pregel. A simple overview of the structure of the Pregel implementation in Spark used by the

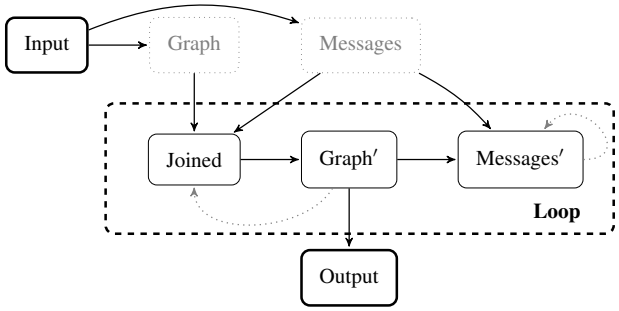

Fig. 12: Spark's Pregel implementation

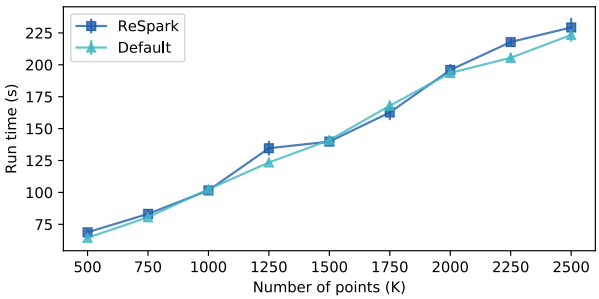

Fig. 13: Shortest paths benchmark results

shortest paths algorithm is given in Figure 12

Spark-Bench uses Spark's shortest paths algorithm to search for the shortest path between two vertices of a graph with log-normal degree distribution ($\mu = 4.0$ and $\sigma = 1.3$ [11]. The shortest paths algorithm is implemented using the Pregel model by maintaining the length of the shortest path to the destination vertex. Each vertex sends a message to its neighbours containing this length with one step added for the extra vertex. Vertices will then update their local data to contain the minimum of the current length and the lengths received in these messages. Eventually these distances will converge so the source vertex contains the length of the shortest path to the destination. Without caching, shortest paths can take several hours to run as the number of vertices in the graph grows. With ReSpark or manual caching, execution time is reduced to a few minutes, as shown in Figure 13.

C. Strongly Connected Components (SCC)

The strongly connected components (SCC) algorithm aims to partition a graph into components in which there exists a path between every pair of vertices. Spark's implementation first assigns a unique component ID to each vertex and tags each vertex as pending. The algorithm then finds all vertices without outgoing or incoming edges marked as final and the component IDs are assigned in the final graph. The second stage uses Spark's Pregel implementation to collect the minimum component ID from all remaining adjacent vertices. Finally, the third stage again uses Pregel to mark new vertices as final if there is a neighbour with the same colour which is marked as final. This continues until no unfinalized vertices remain or the configured number of iterations is complete. The structure of the algorithm including its connection with Pregel is shown in Figure 14. In addition to the caching which takes place inside Spark's Pregel implementation, RDDs generated at the **Graph** call site are also persisted. After materializing this graph, the graph from the previous iteration unpersisted.

We ran the SCC benchmark with its default of 3 iterations. The runtime of ReSpark again closely mirrors that of the original Spark-Bench implementation. Again, we do not include results without caching since runtime exceeds several hours.

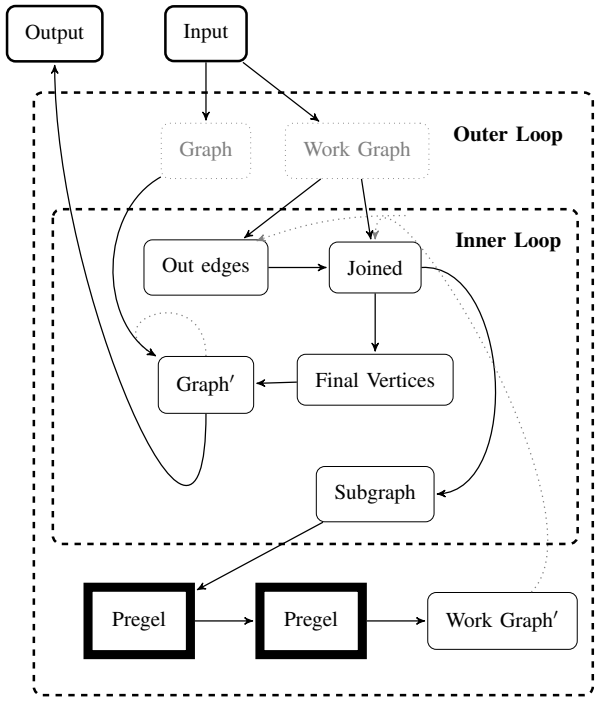

Fig. 14: Spark's SCC implementation (see Figure 12 for the Pregel implementation)

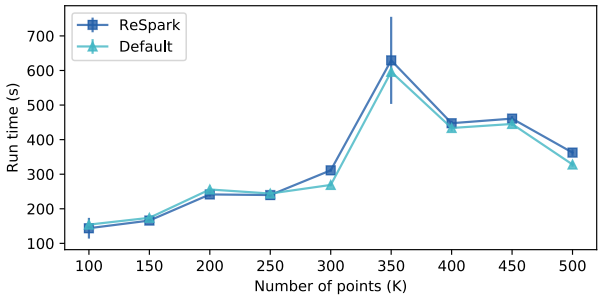

Fig. 15: SCC benchmark results

D. PageRank

Spark's GraphX library also includes an implementation of the PageRank [12] algorithm. PageRank uses edges in a graph to calculate a rank for each node based on the incoming edges. Spark's implementation repeatedly sends messages to adjacent vertices and then uses these messages to update the rank for each node. This algorithm runs a fixed number of iterations before terminating. An overview of the algorithm structure is given in Figure 16. Spark simply computes updates to the rank on each iteration and joins these updates with the original graph. To optimize this computation, GraphX in Spark persist RDDs generated during each iteration. The graph with the current rank of each node is then materialized the before the previous iteration is unpersisted to prevent the issues with unpersist discussed in Section IV-C.

We evaluate PageRank on graphs with a varying number

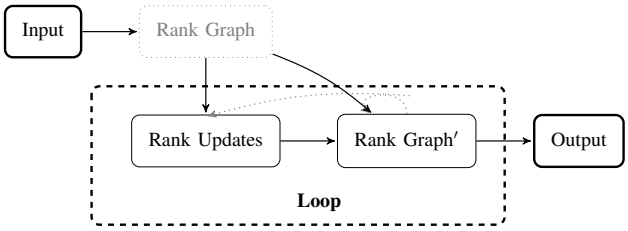

Fig. 16: Spark's PageRank implementation

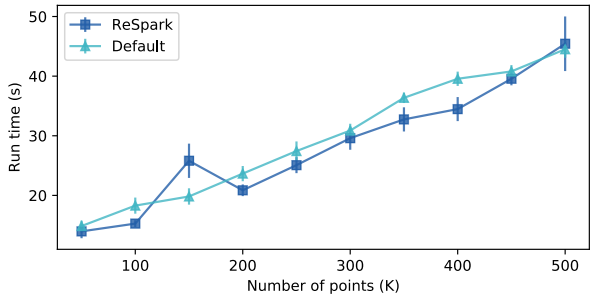

Fig. 17: PageRank benchmark results

of vertices generated as for the graphs for the shortest paths tests above. As shown in Figure 17, ReSpark again has similar performance to the original Spark implementation, without the need for explicit persistence annotations or materialization.

E. Approximate nearest neighbour

Nearest neighbour algorithms aim to identify a vector, from a given vector set, that is closest to a query vector. The relevant code is shown in Figure 18. We apply our iterative optimizations to the underlying k-means model as well as the iteration over the data splits. In addition, we remove all the provided caching annotations (lines 1, 4, and 6 in Figure 18). Table II shows the performance of ReSpark on the SIFT1M (2.3GB) and GIST1M (11.8GB) datasets [13]. ReSpark is able to identify the dataset which needs to be persisted on line 4, and results in a 16% slowdown on the larger dataset compared to the original program with manual persistence allocations. In contrast, the version of the program with the manual persistence annotation disabled is 26% slower on the same dataset. In this case, ReSpark was able to obtain approximately 62% of the benefit of caching without any manual persistence annotations.

Much of this overhead is due to changes in Spark's Python API (PySpark) which introduces the concept of a *pipelined* RDD, which combines multiple Python operations into a single RDD to minimize the communication overhead between the Scala backend and the Python runtime. With explicit caching in Python code, this pipeline is broken in order to create a separate RDD which can be cached. However, ReSpark is not able to break this pipeline since it is not visible within to the Spark runtime. We expect it would be possible to duplicate some of ReSpark's logic within the PySpark API to perform similar decisions at the Python level and communicate these back to the ReSpark runtime, breaking the pipeline where necessary.

```
1  vecs.persist()
2  for split in xrange(M):
3    data = vecs.map(lambda x: x[split])
4    data.persist()
5    sub = KMeans.train(data, ...)
6    data.unpersist()
7    subquantizers.append(sub)
```

Fig. 18: Iterative Python code for LOPQ in Spark

```
1  centered.persist()
2  for iter_id in range(NITER):
3    z = centered.map(...)
4    z.persist()
5    c = z.join(centered)....collect()
6    ub, _, ua = np.linalg.svd(c[0][1])
7    rot = np.array(ua.transpose()
                    .dot(ub.transpose()))
```

Fig. 19: Iterative Python code for ITQ in Spark

Instead, we opted for the simple approach of avoiding pipelining optimization entirely and creating separate RDDs for each PySpark transformation. We leave further optimizations of PySpark to reintroduce this pipelining for future work.

LOPQ

	SIFT1M	GIST1M	BigITQ
Default	543.16s	2138.39s	400.65s
ReSpark	554.54s	2483.88s	410.05s
NoCache	755.48s	2698.31s	408.77s

TABLE II: LOPQ and BigITQ runtime using ReSpark

F. Iterative quantization

A simplified excerpt of the code used for the algorithm is given in Figure 19. The results of these experiments are in Table II. In this case, we see that the algorithm benefits very little from caching. ReSpark introduces a small overhead of approximately 2% compared to caching which is done manually. This overhead results from the tracking done by ReSpark as well as the breaking of Python pipelines as discussed in the previous section.

VI. Related Work

Opportunities for optimizing iterative computations have been explored for Hadoop and MapReduce [2]. HaLoop [14] modifies MapReduce to allow developers to explicitly express iterative computation. These iterative jobs are then optimized by colocating their tasks operating on the same partition. Furthermore, the data used by each task is cached on each node. While this provided significant speedup, we note that the scheduling and caching policies of Spark are able to provide similar optimizations without specific awareness of iteration. In addition to scheduling optimizations, iMapReduce [15] allows map tasks to run asynchronously within an iteration.

Since computation in Spark is lazy, this explicit optimization is unnecessary. iHadoop [16] further exploits asynchrony by also checking loop termination in parallel with speculative execution of the following iteration. If the algorithm is deemed to have terminated, the additional iteration is aborted. Since computations in Spark are lazy, we note that attempting the same optimization may prove harmful. The results of the previous iteration may not be materialized and checking termination in parallel may result in redundant evaluation since the Spark scheduler does not avoid duplicate execution. We leave further exploration of this technique to future work.

Several other pieces of existing work have attempted to optimize the use of caching in Spark. Neutrino [17] and the work of Yang et al. [18] select caching strategies for each RDD partition. To perform this selection, an execution trace without caching is required, which ReSpark avoids. RDDShare [19] identifies views which can be cached in Spark SQL queries to improve future performance. This does not allow for optimization within a single job and only works with SQL queries. Quartet [20] has a similar goal of optimizing the sharing of data across jobs and functions. However, Quartet only optimizes the use of cached partitions of files and does not accelerate jobs using in-memory data. LCS [21] is a new cache eviction strategy for Spark that attempts to make more efficient use of the cache by keeping partitions which are expected to be more expensive to compute. While this results in more effective use of RDDs which have been annotated by application programmers, it does not solve the problem of deciding what to put in the cache. Furthermore, LCS still suffers from the `unpersist` issue discussed in Section IV-C.

Apache Flink [22] is designed to support batch and stream processing in a single execution engine. Flink includes support for *iteration steps* which expresses iteration explicitly to the runtime. While Flink does perform some automated caching and scheduling optimization, when using Flink's iteration steps, only data from the immediately preceding iteration can be used. This is unlike Spark where each iteration can make use of arbitrary references to previously computed data. However, similar optimizations to Spark scheduling similar to those used in Flink may prove beneficial.

VII. Conclusions and Future Work

We modified the Apache Spark distributed computing framework to enable automatic caching of intermediate results. Our modifications, which we refer to as ReSpark, avoid the need to explicitly mark which data should be cached, a process which can be surprisingly unintuitive and in some cases, require detailed knowledge of Spark internals. Instead, we automatically select which results should be placed in the cache and when those results can be removed. We removed explicit caching annotations from several programs and compared ReSpark's performance with the original applications. In comparing ReSpark's automated decisions with manual decisions made by expert Spark developers, we see that ReSpark is able to obtain much of the benefit of caching without the need for expert knowledge.

There are further opportunities to introduce automatic optimizations for Spark. While ReSpark has demonstrated the effectiveness of inferring cache annotations, it is still subject to Spark's LRU cache eviction policy. An eviction policy which considers recomputation cost may be more effective. In addition, the number of partitions used in an RDD or the number of executors also have an impact on performance. Like caching annotations, these values currently must be determined via expert knowledge of Spark and also trial and error. Automating caching also introduces the possibility to modify caching decisions based on parameters such available memory and processing power. These runtime-dependent optimizations present a rich opportunity for future work.

References

[1] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan 2008.

[3] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. EuroSys '10'*. New York, NY, USA: ACM, 2010, pp. 265–278.

[4] M. Li *et al.*, "SparkBench: a Spark benchmarking suite characterizing large-scale in-memory data analytics," *Cluster Computing*, vol. 20, no. 3, pp. 2575–2589, Sep 2017.

[5] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *IEEE CVPR*, June 2014, pp. 2329–2336.

[6] C. Mellina and M. Kurovski, retrieved Jan. 31, 2019. [Online]. Available: https://github.com/yahoo/lopq

[7] R. Girdhar, retrieved Jan. 31, 2019. [Online]. Available: https://github.com/rohitgirdhar/BigITQ

[8] Y. Gong *et al.*, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *IEEE TPAMI*, vol. 35, no. 12, pp. 2916–2929, Dec 2013.

[9] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[10] B. Bahmani *et al.*, "Scalable k-means++," *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 622–633, Mar. 2012.

[11] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. of SIGMOD '10*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[12] L. Page *et al.*, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, Technical Report 1999-66, November 1999.

[13] H. Jégou *et al.*, "Product quantization for nearest neighbor search," *IEEE TPAMI*, vol. 33, no. 1, pp. 117–128, Jan. 2011.

[14] Y. Bu *et al.*, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 285–296, Sep 2010.

[15] Y. Zhang *et al.*, "iMapReduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, Mar 2012.

[16] E. Elnikety *et al.*, *iHadoop: Asynchronous Iterations for MapReduce*. IEEE, Nov 2011, pp. 81–90.

[17] E. o. Xu, "Neutrino: Revisiting memory caching for iterative data analytics," in *HotStorage 16*. USENIX Association, 2016.

[18] Z. Yang *et al.*, "Intermediate data caching optimization for multi-stage and parallel big data frameworks," in *IEEE CLOUD*, 2018, pp. 277–284.

[19] H. Chao-Qiang *et al.*, "Rddshare: Reusing results of spark rdd," in *IEEE DSC 2016*, Jun, pp. 370–375.

[20] F. Deslauriers *et al.*, "Quartet: Harmonizing task scheduling and caching for cluster computing," in *HotStorage 16*. USENIX Association, 2016.

[21] Y. Geng *et al.*, "LCS: An efficient data eviction strategy for Spark," *International Journal of Parallel Programming*, pp. 1–13, Nov 2016.

[22] P. Carbone *et al.*, "Apache Flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.