Arthur: Rich Post-Facto Debugging for Production Analytics Applications

Ankur Dave, Matei Zaharia, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Debugging the massive parallel computations that run in today's datacenters is hard, as they consist of thousands of tasks processing terabytes of data. It is especially hard in *production* settings, where performance overheads of more than a few percent are unacceptable. To address this challenge, we present Arthur, a new debugger that provides a rich set of analysis tools at close to zero runtime overhead through *selective replay* of data flow applications. Unlike previous replay debuggers, which add high overheads due to the need to log low-level nondeterministic events, Arthur takes advantage of the *structure* of large-scale data flow models (*e.g.,* MapReduce), which split work into deterministic tasks for fault tolerance, to minimize its logging cost. We use selective replay to implement a variety of debugging features, including rerunning any task in a single-process debugger; ad-hoc queries on computation state; and forward and backward tracing of records through the computation, which we achieve using a program transformation at replay time. We implement Arthur for Hadoop and Spark, and show that it can be used to find a variety of real bugs.

1 Introduction

Cluster computing frameworks such as Hadoop [1] and Dryad [15] have been widely adopted to enable sophisticated processing of large datasets. These systems provide a simple "data flow" programming model consisting of high-level transformations on distributed datasets, and hide the complexities of distribution and fault tolerance.

While these frameworks have been highly successful, debugging the parallel applications written in them is hard. To solve correctness or performance issues, users must understand the actions of thousands of parallel tasks, which produce terabytes of intermediate data across a cluster.

Debugging becomes especially difficult in *production* settings. Although tools for testing assertions, tracing through data flows, and replaying code exist (*e.g.,* Inspector Gadget [20], Daphne [16], liblog [11], and Newt [8]), they invariably add overhead. For large-scale applications running 24/7, even 10% overhead can be expensive, so most operators do not use these tools in production, making bugs that occur in production time-consuming to diagnose and fix.

In this paper, we present a new debugger, Arthur, that can provide these debugging features at close to zero runtime overhead, using *selective replay* of parts of the computation. Unlike previous replay debuggers for distributed systems [11, 13, 3, 4], which need to log a wide array of non-deterministic events (*e.g.,* message interleavings) and are therefore expensive, we achieve our low overhead by taking advantage of the *structure* of modern data-parallel applications as graphs of deterministic tasks. Task determinism is implicitly assumed by the fault and straggler mitigation techniques in these frameworks [9, 15], but we use this same feature to efficiently replay parts of the task graph for debugging.[1]

While the core idea of selective replay is simple, we show that it can be used to implement a rich set of debugging tools. These include:

- Forward and backward tracing of records through just the portion of the job that they affect.
- Interactive ad-hoc queries on intermediate datasets.
- Re-execution of any task in the job in a single-process profiler or debugger.
- Introduction of assertions or instrumentation (*e.g.,* print statements) into the job.

To implement these features, Arthur must tackle several challenges. First, despite frameworks' assumption of task determinism, user error may cause *nondeterministic replay*, making it impossible to reconstruct a task's output. We do not aim to reproduce nondeterministic results, but instead detect them using checksums of task output across re-executions. We show that this checksumming adds minimal overhead to the original execution.

Second, to enable interactive debugging, Arthur also needs to be fast at replay time. We achieve high performance by (1) only replaying the subset of the job's task graph that is needed for a particular debugging action (*e.g.,* to rebuild the input for one task), (2) parallelizing the replay over a cluster, and (3) caching frequently

[1] Note that Arthur does not aim to debug non-deterministic problems, such as the user accidentally writing a non-deterministic task, but it will *detect* them using a checksum of each task's output. We show that this checksumming adds minimal overhead.

queried datasets in memory to provide fast access. As a result, many debugging queries can be answered within several seconds, even for large applications.

Third, Arthur's tracing feature requires *tracing records across a variety of operators* (*e.g.,* map, filter, reduce, and group-by), taking into account the semantics of each operator. We perform tracing using a program transformation that augments each operator in the job to propagate tags with each record. To make the tracing efficient, we use a compressed tag set representation based on Bloom filters. A major advantage of our program transformation approach is that we do not need to modify the parallel runtime (Spark) to propagate tags.

We implement Arthur to support loading execution logs from either Hadoop or Spark [23] (a recent cluster computing framework with a concise Scala API). The system then replays the job in a Spark-based parallel runtime. It provides an interactive Scala shell from which the user can replay tasks, query intermediate datasets using arbitrary Spark queries, and run other analyses.

We evaluate Arthur on a variety of synthetic errors and three real bugs in Hadoop and Spark programs. The issues we test include logic bugs such as incorrect input handling, performance problems such as data skew, and unexpected program outputs that Arthur can trace back to specific input records. In all cases, Arthur's suite of tools can quickly narrow in on the problem. At recording time, Arthur adds less than 4% overhead and produces logs at most several megabytes in size. At replay time, Arthur finishes most analyses in a fraction of the running time of the original job, thanks to selective replay and in-memory caching, and supports querying intermediate datasets in sub-second time.

To summarize, our contributions are:

- The observation that data flow frameworks' decomposition of jobs into deterministic tasks, which is fundamental in these systems for fault tolerance, can also be used to for low-cost replay debugging.
- A set of rich and efficient debugging tools that selectively replay only the part of the computation needed for an analysis, including ad-hoc queries and forward and backward record tracing.
- A fast, interactive environment for debugging jobs that allows ad-hoc queries in the Scala language and provides high responsiveness using parallel execution and in-memory caching.

The rest of the paper is organized as follows. Section 2 describes the target environment for Arthur. We then discuss its architecture and capabilities in Sections 3–6. Section 7 discusses our implementation. We then evaluate Arthur (Section 8), discuss limitations and extensions (Section 9), and survey related work (Section 10).

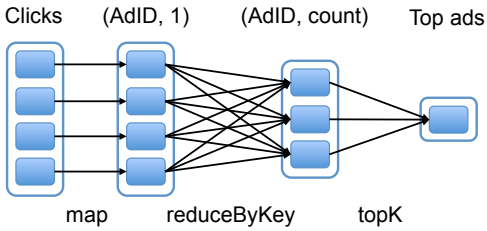


Figure 1: Dependency graph for the tasks produced in our example Spark program. Tall boxes represent datasets, while filled boxes represent partitions of a dataset, which are computed in parallel by different tasks.

2 Target Environment

Arthur is designed for parallel computing frameworks that use a data flow programming model, which include MapReduce [9], Dryad [15], Spark [23], Hyracks [5], Pig [21], FlumeJava [6], and others. It can also be applied to frameworks that are not typically thought of as data flow but do break computations into deterministic tasks, such as the Bulk Synchronous Parallel model in Pregel [18], where nodes operate on data locally and communicate at a barrier through the equivalent of a reduce operation. Many frameworks adopt this type of deterministic model for fault tolerance.

Conceptually, data flow frameworks allow users to build and manipulate parallel collections of records, which we refer to simply as *datasets*. Users manipulate these datasets through deterministic *transformation* operators, such as MapReduce's *map* and *reduce* operators and DryadLINQ's SQL-like transformations. Typically, each dataset is immutable and transformations return a new dataset, although in practice implementations may reuse storage space for different datasets.

Data flow frameworks implement this abstraction by parallelizing transformations into *tasks* that execute concurrently across the cluster. Each task transforms its input records independently. For example, a map task would run a user function on each element of its input.

Task determinism is implicitly assumed for the purpose of straggler and fault recovery. In straggler mitigation, the framework speculatively launches duplicate copies of slow-running tasks in the hope that slow performance is a machine-specific problem. In fault recovery, frameworks use the dependency graph between tasks to recompute data partitions lost after a failure [9, 15]. Both of these techniques require tasks to execute deterministically, and Arthur's replay approach further takes advantage of this assumption to provide accurate replay. Nevertheless, Arthur can detect nondeterminism and alert the user to the error, as described in Section 3.

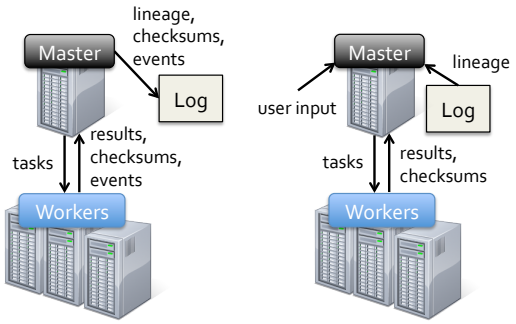
To illustrate the structure of a data flow program, Fig-

ure 1 shows the lineage graph for a simple Spark program that computes the top 100 ads clicked in a log. This program's code is shown below:

```
val topAds = clicks.map(c => (c.adID, 1))
                   .reduceByKey((a, b) => a+b)
                   .topK(100)
```

This program uses Spark's functional API in the Scala language [23] to take a dataset called `clicks` (loaded, *e.g.,* from a file) and run *map*, *reduce*, and *topK* transformations on it. The user code passed to these transformations (in this case, the Scala functions `c => (c.adID, 1)` and `(a, b) => a+b`) is expected to be deterministic.

Using Arthur, we can rerun just enough portions of the dependency graph to answer a particular debugging query. For example, if one of the *reduce* tasks is running slowly, we could replay all of the maps and that one reduce task, without having to replay the rest of the job.



(a) Recording (b) Debugging

Figure 2: Flow of information while recording a program's execution and replaying and debugging the program. MapReduce, Dryad, and Spark carry out user transformations by deploying tasks onto the cluster and receiving their results. Arthur logs additional information about the program's execution, which it can replay on demand after the program finishes.

3 Architecture

The main idea in Arthur is that we can *record* a data flow program's dependency graph at runtime, and *selectively replay* parts of the execution at debug time to answer users' questions. In this section, we describe how Arthur performs recording and replay.

At record time, Arthur runs as a daemon collocated with the cluster computing framework's master that logs several types of information. The most important is the program's *dependency graph*, which consists of every transformation in the program (*e.g.,* the map and reduce in MapReduce or an operator in DryadLINQ or Spark) along with what input datasets or external files it acts on, and how it was partitioned into tasks. Arthur also records a *checksum of each task's output*; this allows the debugger to compare the checksums at replay time to the original ones, and alert the user that a task is nondeterministic if they differ. Finally, Arthur logs the execution time of each task, as well as the *cause of failure* (*e.g.,* an uncaught exception) for any tasks that fail. Figure 2a summarizes the flow of information at record time.

After the program has finished running, the user launches Arthur in replay mode and loads the program's execution log. Arthur then accepts queries through an interactive shell and uses the recorded information to *replay* parts of the program's execution on demand. Replay takes place in parallel on the cluster; Arthur replays tasks from the appropriate parts of the dependency graph by launching them using Spark. Figure 2b illustrates the flow of information at replay time.

Arthur's basic replay functionality, described in Section 4, supports rerunning portions of the program exactly. This allows users to visualize the program's dependency graph, explore intermediate datasets, and rerun specific tasks locally in a conventional debugger.

Arthur also provides a more powerful type of replay that involves *modifying* the original operator graph. This makes it possible to perform analyses such as tracing records forward and backward through the data flow (described in Section 5). It also makes it possible to insert post-hoc instrumentation into the execution graph, such as assertions on intermediate datasets and other custom code (described in Section 6).

4 Basic Features

Arthur's core features are built on top of its ability to load the original program graph and replay parts of it on demand. The user accesses these features through an interactive shell based on Spark's Scala shell.

4.1 Data Flow Visualization

The simplest tool that Arthur provides is to use the lineage of datasets in the execution log to provide a visualization of the program's data flow graph. Such a visualization can be helpful in understanding the data access patterns and general structure of the program, and it only requires local analysis rather than re-execution of tasks in the job. Figure 3 shows an example lineage graph produced by Arthur on a PageRank application. Arrows point from datasets to their dependencies. The graph indicates that dataset 4 is used repeatedly in future computations, suggesting that it might be a good candidate to be cached in memory on the cluster, for example.

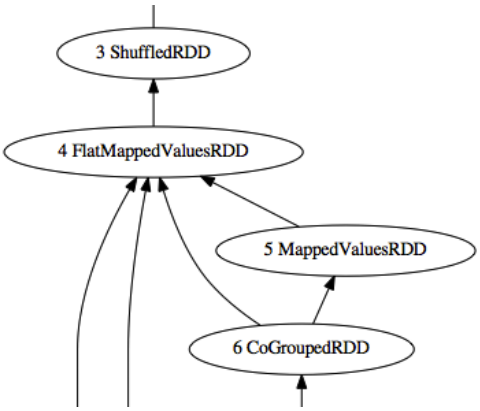


Figure 3: Partial lineage graph of a Spark application, as plotted by Arthur.

4.2 Exploration of Intermediate Datasets

In debuggers for programs on a single machine, variable inspector windows and "print" commands give visibility into a program's intermediate state. Arthur can provide a similar experience for data flow programs by allowing the user to query any intermediate dataset post-execution from the interactive debugger shell. To query an intermediate dataset, we (1) read the dependency graph that Arthur recorded, (2) find the tasks required to rebuild it, and (3) run them on workers across the cluster. Queries on datasets can be written using any of the operators in Spark [23], which include relational operators, sampling, and transformations using arbitrary Scala code.

For example, the following console session shows how a user might explore an intermediate dataset in the program from Section 2 that computes the top 100 ads clicked in a log. The user loads the `topads.log` execution trace and queries dataset 2, which is a collection of ads and the number of user clicks on each ad.

```
scala> val r = new EventLogReader("topads.log")
  r: EventLogReader = EventLogReader@726b37ad

scala> r.datasets(2).take(5) // sample first 5 ads
                             // and click counts
                             // from dataset 2
  res0: Array[(AdID, Int)] = [...]

scala> r.datasets(2).map(pair => pair._2).mean()
  res1: Int = 258288  // mean # of clicks per ad
```

Because Arthur executes its operations in parallel on the cluster, queries on intermediate datasets run at least as quickly as the original program, and frequently more quickly because only part of the job needs to run in order to produce the requested output. In addition, Arthur uses Spark's capability to cache information in memory once it is computed, enabling it to respond to repeated queries on the same dataset at interactive speeds.

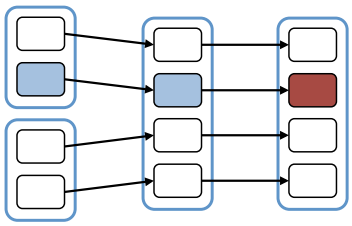


Figure 4: Tasks that need to be rerun for local task replay. To rerun a task (dark red), Arthur first runs its ancestors (light blue) and saves the last output. It is only necessary to run these tasks rather than the entire job.

4.3 Task Replay

Logic errors in bulk operators such as *map* functions can be difficult to debug because they execute in parallel on many machines. Programmers typically debug such operators by printing trace information from within the operator and later reading logs on machines that produced exceptions or incorrect results. Instead, it would be helpful to use conventional step-through debuggers and profilers. For example, if a certain task is throwing an exception on specific input, stepping through the user code in that transformation would make it easier to debug.

Arthur supports running specific tasks locally under such tools. To rerun a task locally, Arthur first computes the input to that task by running the tasks that it depends on; these tasks persist their outputs to disk. Arthur then launches a small wrapper program locally that receives the task metadata, fetches the outputs of parent tasks, and executes the task in an isolated environment. The user can then attach a conventional debugger such as JDB before the task runs, making it possible to set breakpoints, catch exceptions, and step through the operator's execution on the input data of interest.

Local task replay only requires a small portion of the program to be re-executed. In particular, only the task's ancestors must be run, rather than the entire program. Figure 4 shows that in order to debug an incorrect result from a particular task, Arthur only needs to rerun those tasks which contribute results to that task's input.

5 Record Tracing

Finding the set of records that stemmed from or led to a given record can be helpful in debugging the program's operations. For example, in a post to the Spark user group, a user described a word count application that unexpectedly output a count for the empty string in addition to the counts for each word in the input. Tracing that record backward through the program would reveal that the empty string stemmed from empty lines in the input, allowing the user to fix the input parsing bug. Arthur pro-

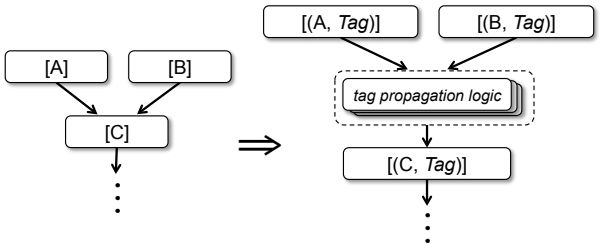
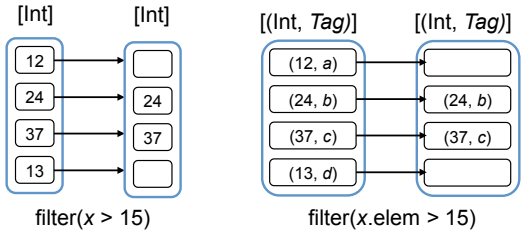


Figure 5: For tracing, Arthur rewrites the dependency graph to propagate *tags*, which represent provenance, along with each element. For example, the original dependency graph contains an operator that merges datasets of A and B elements to form a dataset of C elements. In the modified graph, the original operator is wrapped with logic to propagate the tags.



(a) *filter* on integers (b) *filter* on tagged integers

Figure 6: The original *filter* operator applies a user-supplied predicate directly to each element, while the augmented operator extracts the integer element before passing it to the predicate.

vides the ability to perform such tracing "post-hoc" by rerunning a *transformed* version of the original program.

5.1 Mechanism

Arthur provides record tracing using a program transformation in which it modifies each operation in the original program graph to propagate a *tag*, which represents provenance, along with each element. In addition to performing its previous function, each operator in the new execution graph is augmented to propagate tags, as illustrated in Figure 5. We implement forward and backward record tracing by using this tagging primitive to mark elements of interest and track their ancestors or descendants in the data flow, as described in the next sections.

The program transformation approach takes advantage of the functional, high-level nature of operations in modern data flow frameworks, which provide Arthur with precise information about how data moves through the program. To extract this information, it is necessary for the definition of each operator to include operator-specific tag propagation information. For example, Figure 6 shows that the *filter* operator propagates tags by extracting the element and passing it to the filtering function. In general, an operator f from datasets of type A to datasets of type B must also come with a function from the original operator f to an operator from datasets of type (A, Tag) to those of type (B, Tag).

This approach to record tracing relies upon the fine-grained semantics of dataset operations for accuracy. Operations like the *filter* above carry each input record to at most one output record, allowing the system to track records without any loss of fidelity. On the other hand, some frameworks, such as Hadoop, provide a coarser-grained API where a "map" function operates on multiple input records at once using an iterator, and writes output to a second iterator. Such operations expose only a coarse data flow structure, limiting the fidelity possible with a simple program transformation. More involved techniques such as static analysis of user-supplied operations could improve fidelity. In addition, for these types of operations, Newt [8] proposes a timing-based approach using the observation that any particular output record could only have been influenced by the input records that have appeared until that point. We use this approach in Arthur to handle Hadoop's map operator and a similar operator called *mapPartitions* in Spark.

5.2 Forward Tracing

Forward tracing is straightforward to implement on top of the tag-propagating program transformation. Arthur transforms the dependency graph into one that propagates a Boolean tag in addition to each record. It initializes the input records of interest with a `true` tag and other records with a `false` tag, runs the modified job using the Boolean *or* operation to combine tags, and finds which output records end up with a `true` tag. We show that forward tracing requires $< 1.5\times$ overhead at debug time, while leaving the original runtime unaffected.

When only a few records are of interest, Arthur traces them through just the relevant *subset* of the execution graph. For forward tracing, Arthur reruns the required tasks in each stage with tagging, inspects these tasks' outputs, looks up the elements' shuffle keys to determine which tasks in the next stage read records from these tasks, and repeats the process on the new set of tasks.

5.3 Backward Tracing

Like forward tracing, backward tracing builds upon tag propagation. Because operators are not guaranteed to be invertible, backward tracing cannot simply tag output records with booleans and run the program in reverse. Instead, it tags each input element with a *unique* tag, runs

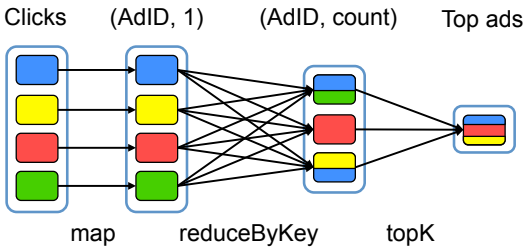


Figure 7: To trace an output record (rightmost rectangle) *backward* through the data flow, we tag each input element uniquely, run the job to propagate the tags to the outputs, and find which input elements contributed tags (leftmost blue, yellow, and red rectangles).

the job, examines the tags that ended up on the output records of interest, and finds which input elements contributed those tags. This process is illustrated in Figure 7.

We implement unique tags using integers. Each input records is tagged with a unique integer based on its position in the dataset, and tags are stored as sets of integers which are combined using the *union* operation.

This approach works well for programs such as database queries where each record has a clear provenance. However, in iterative programs such as PageRank, each output record is influenced by a large number of input records. As a result, tags tend to diffuse widely, and in the extreme case each output record may end up with a tag from every input record. This approach therefore performs poorly for long jobs because of the high space overheads that tags impose in later stages. Representing tags as Bloom filters provides a 1.78× speedup on PageRank, at the cost of false positives.

An alternative approach to backward tracing is to trace the output records of interest backward through each stage, starting with the last stage. In each stage, Arthur tags each record in the shuffle output from the previous stage with a unique label, runs the stage, and finds which input elements contributed to the tags of the output elements under consideration. This approach allows the tag dependency structure to be precomputed, allowing further backward tracing queries to be performed using just a single cluster-wide join operation.

6 Post-Hoc Instrumentation

Arthur's ability to replay a *modified* version of the data flow graph, which we used in tracing, also opens the door to other types of analyses. One that we have explored is *post-hoc instrumentation* of the code, where assertions or print statements can be added to part of the job after it was executed and can be verified in the debugger. While step-through debugging and tracing are powerful tools for tracking down problems, often the best way to understand program execution, especially in a large application, is to test assertions, and Arthur provides the ability to inject these without runtime overhead.

The simplest type of assertion one can test is about the contents of a particular dataset. For example, suppose that we were debugging a PageRank application, and we wanted to ensure that the PageRank of each node was positive at all iterations. We could attach an assertion to the dataset for a given operation as follows:

```
scala> val ranks = r.datasets(2) // dataset ID #2

scala> ranks.assert(r => r > 0)
```

Arthur will test the assertion by adding a no-op *map* operator after the computation of `ranks` that verifies the predicate and reports any records that do not pass it to the master. By default, Arthur's assertions are attached "lazily" (they are not tested right away), so it is possible to attach multiple assertions to multiple datasets, and then test all of them using a `EventLogReader.checkAssertions()` function. Because Arthur's shell is simply a Scala interpreter, it is also possible to attach these assertions programmatically (*e.g.,* search through the list of datasets for all the ones created on a particular line of the program and add assertions to all of them).

Apart from these types of data assertions, Arthur also allows users to instrument their code more closely by replaying a *modified* version of the original binary. As long as the functions in the program still produce the same outputs (*i.e.,* any modifications are only for print statements or assertions), the system can still run the program on the cluster. Currently, this modification has to be done manually before starting Arthur, but we also wish to support dynamic modification of the program code using the Java VM's class reloading feature [17] in the future.

Finally, it would be straightforward to extend the assertion mechanism to support "distributed assertions" (in the form of a Spark expression that has to hold for an entire dataset, *e.g.,* that the sum of PageRanks is close to 1) [10]. Currently, these can be checked using manual Spark queries on the datasets, as in Section 4.2.

7 Implementation

We implemented Arthur in about 2000 lines of Scala code. The system supports recording applications written in either Hadoop or Spark, and replaying them in a Spark-based parallel runtime where different debug operations can be invoked interactively from a Scala shell.

At recording time, Arthur needs to obtain (1) the graph of operators used in the parallel job and (2) checksums of intermediate datasets, used to verify that re-execution

has been deterministic. In Hadoop, the operator graph is trivial, because it is always a single MapReduce step, so we only use the job's configuration to rerun the same map and reduce functions. In Spark, we added an event logging module that logs the datasets used in each parallel operation to a file. For checksums, we use a simple Java OutputStream wrapper that computes a checksum as data is written out. We only perform checksumming for data at task boundaries (*e.g.,* for the output data in a map task), where it is either written to a file or sent over the network, so the checksumming adds little overhead because the cost of sending the data over the network is much more expensive.

At replay time, Arthur replays both Hadoop and Spark computations in the Spark engine, to take advantage of features such as in-memory caching and interactive queries. We use an existing layer on top of Spark, called SHadoop, to execute Hadoop map and reduce tasks within Spark. (This is conceptually simple because Spark also supports map and reduce operators.) We begin by loading the job's code from a path provided by the user, followed by an event log with the parallel operations run during the job and their operator graphs (as discussed above), then present the user with an interactive Scala shell where they can view the operations and datasets and run queries on them. The actual replay of both the original operators and any transformed versions of them (*e.g.,* for assertions or tracing) is implemented by submitting jobs to the existing Spark engine, so it does not require changes to Spark.

The debugging interface for Arthur is a shell in the Scala programming language, based on Spark's interactive Scala shell. It provides an object model to load and debug jobs, and lets the user define local variables or functions using the complete Scala language, and query datasets using functional operators written in Spark. Users can also explicitly control which datasets to cache in memory for repeated queries. For example, the following console session shows how one might query an intermediate dataset in a PageRank computation, whose log is read from `pagerank.log`, and replay a task:

```
scala> val r = new EventLogReader("pagerank.log")
  r: EventLogReader = EventLogReader@726b37ad

scala> r.datasets
#00: hadoopFile at PageRank$.main(PR.scala:31)
#01: map        at PageRank$.main(PR.scala:31)
#02: map        at PageRank$.main(PR.scala:35)
#03: groupBy    at PageRank$.main(PR.scala:35)
#04: flatMap    at PageRank$.main(PR.scala:35)
#05: map        at PageRank$.iterate(PR.scala:91)
#06: cogroup    at PageRank$.iterate(PR.scala:92)
[...]

scala> r.datasets(2).count()
  res0: Long = 129941   // # of elements in dataset 2
```

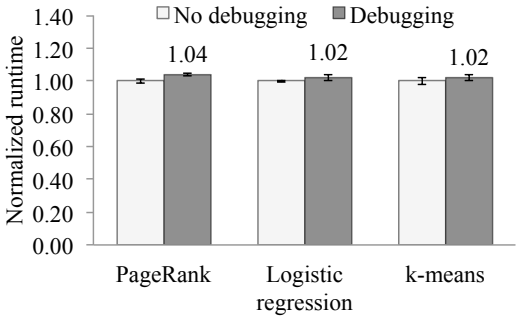


Figure 8: Performance comparison of various Spark applications with and without debugging.

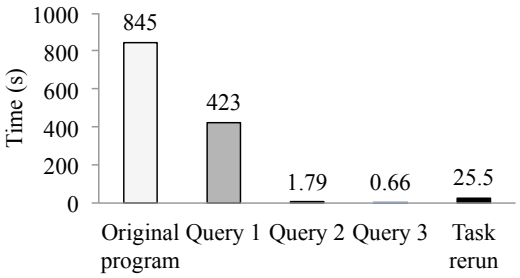


Figure 9: Running time of various interactive queries in the debugger. Arthur only runs the tasks necessary to answer each query, so Query 1 is faster than the original program. Subsequent operations benefit from in-memory caching.

```
scala> r.debugTask(3, 1) // replay task 1 of dataset 3
[launches the JDB debugger]
```

Finally, for replaying individual tasks in a single-process debugger, we wrote a small wrapper program that reads a serialized Task object from Spark and reads its input from a file (computed in parallel using the cluster), then executes that task locally, so that we can spawn this program in a local subprocess and attach JDB or other debugging tools to it.

8 Evaluation

We evaluated Arthur using a variety of real bugs and injected errors in Hadoop and Spark programs. We find that Arthur's overhead at record time is only 2–4%, making it feasible to run continuously. At debug time, we found that Arthur can respond to queries at interactive speeds (on the order of a second) by caching frequently-used datasets in memory, and often needs to rerun only a subset of the job even for the first time it answers a query. Finally, we evaluated Arthur's applicability to different types of bugs by showing how it can diagnose task fail-

ures, incorrect output, nondeterministic behavior, and deterministic performance problems in various programs.

8.1 Recording Overhead

We tested Arthur's recording overhead on three Spark programs: PageRank, logistic regression, and k-means. Our PageRank program runs 15 iterations on the articles in a 54 GB Wikipedia dataset, our logistic regression program runs 10 iterations on 10 million 10-dimensional vectors of doubles, and our k-means program runs 5 iterations to cluster 100 million 4D points into 4 clusters. Figure 8 compares the runtimes of these applications with and without Arthur when running on a 20-node cluster. Because the overhead from task output checksumming constitutes most of Arthur's recording overhead, applications with large task outputs see higher overhead than those with small task outputs. Each PageRank iteration updates the rank for every page, forcing Arthur to checksum a large amount of information. On the other hand, each iteration of logistic regression only outputs a single vector, and each iteration of k-means only updates a small number of cluster centers. Regardless of application, however, Arthur's logging overhead was at most 4%, and its log size less than 5 MB.

8.2 Replay Performance

At replay time, Arthur can often run faster than the original program thanks to selective replay and in-memory caching. Figure 9 shows the running times of various queries when debugging the PageRank program. Query 1 counts the number of articles whose PageRank in iteration 3 is greater than a threshold. Arthur only needs to run the first three iterations to answer the query, so it runs faster than the original program. Next, Query 2 calculates the average PageRank in the third iteration, and Query 3 sample the PageRank of ten articles. These queries reuse the same dataset as Query 1, so they benefit from in-memory caching. Finally, rerunning a single task locally was fast compared to the original program.

We also benchmarked the performance of forward and backward record tracing during replay of the PageRank application. Forward tracing was 1.48× slower than the original program, while backward tracing with integer set tags was 5.52× slower. The large slowdown for backward tracing was due to diffusion of tags, as described in Section 5.3. Both versions correctly identified the records that an input depended on (*e.g.,* in tracing through three iterations of PageRank, we find neighbours at most three links away). We also tested backward tracing with Bloom filter tags. This reduced the slowdown to 3.09× the original program due to the more compact size of the Bloom filter representation, but caused about 30% of records in the output to be erroneously tagged. The actual performance ratio and error rate depends on the size of Bloom filter used.

8.3 Applicability

Though Arthur is focused towards debugging deterministic problems, we have observed these to be more common than nondeterministic errors for complex distributed programs due to the fact that, like MapReduce and Dryad, Spark requires transformations to be deterministic. To illustrate the kinds of errors that Arthur can detect, we describe its applicability to three deterministic errors: task failures, incorrect output, and performance problems. We also describe Arthur's ability to detect unintended nondeterminism through checksumming on a real bug from Conviva, as well as its support for loading Hadoop traces on a pre-existing bug in Mahout.

Deterministic Task Failures As an example of a deterministic task failure, we injected an input processing bug into our Wikipedia PageRank program. This program extracts the links from each article by parsing its XML representation and searching the parse tree for link elements. Certain tasks were consistently throwing XML parsing exceptions, so we used Arthur to search the event log for failing tasks. We reran one of the tasks in a conventional debugger and found that the culprit records contained `\N` in place of the article's XML. It turned out that our Wikipedia dump used this string to represent an empty article, so we fixed the bug by adding error-handling code for that case.

Incorrect Output As an example of incorrect output, Carat, a real Spark program for processing time series of devices' power usage, contained a bug causing the output to contain nonsensical negative values for power usage. We used Arthur to inject injecting assertions at every stage and then replayed the program. Arthur detected the assertion failures immediately after the location of the bug and we were able to halt the program early, avoiding the inconvenience of rerunning the program in its entirety.

Deterministic Performance Problems We debugged Monarch [22], a real Spark program that used logistic regression to classify spam status updates. The program was running much more slowly than expected, and we observed that a few straggler tasks were consistently finishing last, tens of seconds later than the typical task. We recorded the task IDs that appeared to be stragglers and, once the program had finished, we loaded its trace into Arthur and reran the tasks locally under JDB. Examining

the input partition to the task revealed that it contained several very large feature records, identifying partition skew as the source of the problem. We were able to reproduce the problem because data flow frameworks typically sharded data into partitions deterministically.

Unintended Nondeterminism We used Arthur to detect an bug arising from unintended nondeterminism at Conviva, a video analytics company. An analytics query intended to operate on new records that had arrived in the last few minutes was performing the filter by comparing record timestamps against the current system time from within the query, as in the following example:

```
records.filter((System.currentTime() - _.time)
               < INTERVAL)
```

When we used Arthur to replay the query, we received checksum mismatch warnings because the time had changed from the original execution, and the query now matched fewer records. Examining the operator that triggered the warnings revealed the bug, and we fixed the bug by computing the time in the driver program instead of in the operator, so that the reference time would remain the same across executions of the task:

```
val now = System.currentTime()

records.filter((now - _.time) < INTERVAL)
```

Hadoop Jobs To demonstrate Arthur's support for loading Hadoop job traces, we chose a pre-existing bug in Mahout [2], a Hadoop-based machine learning library. We reproduced the bug, MAHOUT-363, in Arthur. This bug involved a NullPointerException due to a logic error in the *map* code within Mahout. We were able to load the affected Hadoop job into Arthur, identify the map task causing the error, and rerun and step through the task locally until the NullPointerException. Inspecting the task code and state in JDB confirmed that the exception was due to the Mahout Cache's failure to handle a null feature vector.

9 Discussion

Arthur can perform detailed analysis of job executions with nearly zero runtime overhead by leveraging the determinism and structure of modern data-parallel applications. While the core idea behind Arthur is simple, we showed that it efficiently supports a wide range of analyses, which can be sped up by only replaying the relevant parts of the job. We believe that Arthur's approach is important for two reasons. First, because the deterministic data flow model we exploit was primarily adopted for fault tolerance, we believe that it will remain present not only in today's frameworks (*e.g.,* MapReduce, Dryad, Spark), but in future ones as well. Indeed, it is interesting that not only the determinism itself, but also the decomposition of jobs into small tasks, is used to *speed up recovery on failure* (by minimizing the work redone), and both elements directly speed up selective replay. Second, because of the intrinsically high hardware cost of big data computations, any instrumentation at runtime is expensive, so replay may be the *only* effective way to debug production problems. Therefore, it is important to study which analyses can be performed this way.

Because of its reliance on replay, Arthur does have limitations that more invasive debuggers would not. We discuss some of these next, followed by ways in which Arthur can be extended. We also discuss how parallel runtimes could be extended to enable easier replay.

9.1 Limitations

Arthur's replay approach has several limitations, some of which can be avoided with more care during execution:

Nondeterministic User Code Arthur cannot replay bugs where a user's code (*e.g.,* a map function) is nondeterministic, although it *detects* them using checksumming. While users of data flow frameworks are asked to try to write deterministic code to enable fault recovery, nondeterminism can still be a bug, so it is important to be able to fix it. There are two interesting possible approaches. One is that, once nondeterminism has been detected, Arthur can try to *reproduce* nondeterministic behavior (though maybe not the same as the original run) by simply running multiple copies of the problematic task. It could also run these tasks in a more expensive replay debugger, such as R2 [13], that can recreate nondeterministic events once it seems them the first time. A second approach would be to try to identify nondeterministic code through static analysis (*e.g.,* see whether particular libraries are being called).[2]

Inter-Task Interactions Sometimes, bugs are not caused by a particular task, but by the interaction between multiple tasks on the same machine. For example, Hadoop runs a series of tasks in the same Java VM to amortize startup costs, but if each task leaks memory or uses a library with global state, the behavior of a task may depend on which others have run before it. Arthur cannot guarantee to run the same tasks together at replay time (especially if doing selective replay). Auxiliary monitoring tools, such as a memory usage monitor, might be used to detect some of these conditions.

[2]The most common nondeterministic library that might be called is a random number generator, but fortunately, runtimes *can* make that deterministic by seeding the generator consistently for a given task ID. For example, Spark does this for its built-in *sample* operation.

Lost Input Files Arthur implicitly assumes that the input files for each job are still available at replay time. Fortunately, most data warehouses operate in an "append-only" fashion, and retain files for a long time after ingestion. HDFS does not even support random updates.

Communication Order A subtle issue that can happen in some frameworks is that even though the code in each task is deterministic, an instance of that task might fetch input data from other tasks in a nondeterministic order. For example, in Spark, a reduce task performing a commutative operation (such as a sum) fetches results from multiple map tasks in parallel and receives chunks from different tasks in different orders. Although the operation is, technically, expected to be commutative, some bugs might manifest only depending on the input order. In our implementation, we modified Spark to log the order of chunks fetched and use the same order at replay time. For Hadoop, this problem is not present because Hadoop always sorts the input to a reduce function.

Nondeterministic Programming Models While Arthur works for many current frameworks that perform deterministic computations, such as MapReduce, Dryad, Spark, Hyracks, and Pregel, it cannot be applied to programming models that allow nondeterministic, asynchronous messaging, like MPI or Graphlab [12].

9.2 Extensions

While we have implemented several useful debugging tools in Arthur, there are other analyses that would be interesting to implement in the model, especially by taking advantage of the parallelism of the cluster. In addition, if replay is going to be the cheapest way to debug production problems, it would also be interesting to extend runtime frameworks to better support it.

Parallel Profiling Arthur provides a very effective foundation to run shadow profiling [19], a technique where multiple copies of the program are run in parallel with sample profiling to collect highly detailed statistics. Arthur's model naturally allows doing this for only a subset of the job (*e.g.,* one task) and feeding the same input to every copy.

Minimal Example Discovery When a deterministic bug, such a task crashing, occurs, it would be useful to automatically "narrow down" on a smaller example that produces the problem by trying smaller subsets of the job's input. This approach is taken in some existing testing tools, such as QuickCheck [7], and clearly benefits from a parallel search.

Extending Runtimes for Debuggability Some of the limitations we highlight in the previous section lead directly to ways to extend parallel runtimes for easier debuggability (and, ultimately, more chance of recovering correctly from faults as well). Some ways that frameworks could help Arthur include isolating tasks from each other in separate processes,[3] providing hooks to fetch task inputs in a specific order (as we have done in Spark), and retaining intermediate data on the filesystem after job completion (if the framework normally writes temporary files and then deletes them), and allowing this to be used as an input during replay to avoid recomputation. Most of these changes would make programs in these frameworks easier to understand in general.

10 Related Work

Debuggers for Data Flow Frameworks Two recent systems for debugging parallel data flow programs are Inspector Gadget [20] and Daphne [16].[4]

Inspector Gadget is a debugger for programs in the Pig scripting language that adds instrumentation into the program to monitor various properties (*e.g.,* the time spent in each task, the number of records matching a predicate, or user-specified assertions). However, this approach requires the user to instrument their job before they run it, and does not allow the user to rerun a task in a local debugger or to run ad-hoc queries on intermediate datasets that she did not add instrumentation for in advance. The runtime overhead from instrumentation can be as high as 70% for some analyses (*e.g.,* data sampling and latency analysis using tags), making it expensive to run in production. In contrast, our replay approach lets users ask ad-hoc questions about the job *after* it finished, including rerunning parts of the job with the same kinds of instrumentation available in Inspector Gadget, and additionally supports local step-through debugging of tasks.

Daphne lets users visualize and debug DryadLINQ programs. Daphne provides a "job object model" for viewing the tasks in a job, hooks for attaching a debugger to a remote process on the cluster, and the ability to replay a task in a single-process debugger *as long as its input data is still available on the cluster*. This approach works in DryadLINQ because all communication between tasks is through files on disk, but it will not work in the increasing number of frameworks that perform computations in memory (such as Pregel [18], Graphlab [12], or Spark [23]), or for jobs where the intermediate data has been deleted. In contrast, Arthur can *recompute* the input to any task. In addition, Arthur also provides checksumming to verify that the user's code

[3] One reason this was not done in Hadoop is due to the startup cost of new Java VMs, but this does not need to be a fundamental limitation.

[4] "Arthur" was chosen to continue this trend of cartoon characters.

Property	Inspector Gadget	Daphne	liblog/ R2/ODR	Arthur
Job visualization	✓	✓	✗	✓
Queries on intermediate data	✗	✗	✗	✓
Local task replay	✗	✓*	✓	✓
Assertions	✓	✗	✓	✓
Profiling	✓	✓	✓	✓
Record tracing	✓	✗	✗	✓
Runtime overhead	5–70%†	minimal	$> 20\%$	$< 5\%$

Table 1: Comparison of Inspector Gadget, Daphne, general replay debuggers, and Arthur. Note that (*) Daphne's task replay requires that all intermediate data in the job is saved to disk and available at debug time, and (†) Inspector Gadget requires instrumenting jobs at runtime, with varying overhead based on the analysis done.

runs deterministically (an assumption in Daphne) and a rich set of capabilities that are not present in Daphne because they require running new code on the cluster, such as running ad-hoc queries on intermediate datasets.

In general, our implementation provides a superset of the features in these debuggers, and shows that these features can be be implemented "post-facto" using selective replay of the parts of the job that a particular feature requires. Other features in Arthur, such as the ability to reconstruct intermediate datasets and run ad-hoc queries on them, or to add new assertions after the job has finished, are unique in our approach because they require running new computations on the job's intermediate data without knowing these computations during the original job's execution. Table 1 summarizes the features in Arthur in comparison to other debuggers.

Record Tracing and Provenance Several projects have explored how to efficiently capture provenance of records in data-parallel computations to enable tracing. RAMP [14] defines and captures provenance for "generalized map and reduce workflows," which are programs composed of an acyclic graph of map and reduce steps. However, because it does this tracing during job execution, it can add substantial runtime overheads (20–76%). Newt [8] is a provenance capture and replay framework for Hadoop and Hyracks that supports capturing record-level provenance at runtime, and then replaying just the part of the job that produced a particular output record. Unlike RAMP, it also handles map operators that work on a stream, by looking at the interleaving of records being read and written to determine which input records affected an output record. (RAMP assumes that the map function processes just one record at a time and cannot maintain state between records.) However, Newt still incurs about 14–26% runtime overhead, and it lacks other debugging functions, such as checking assertions or running ad-hoc queries on intermediate datasets. Inspector Gadget [20] supports forward tracing through tag propagation and backward tracing by tagging all input records, but it again needs to perform this tagging at runtime.

All of these approaches could be used within Arthur to capture provenance for records in (part of) the job when recomputing it, while avoiding the runtime overhead in production. Our current tracing module uses the properties of Spark and Hadoop operators, as well as a Newt-like approach for map functions that operate on iterators.

Replay Debuggers Replay debugging for distributed systems has been extensively studied through systems such as `liblog` [11], R2 [13], ODR [3], and DCR [4]. However, these systems are designed to replay *general* distributed programs, and thus work by recording all sources of nondeterminism, including message passing order across nodes, system calls, and accesses to memory shared across threads. This results in significant overhead at runtime (often more than 20%), or even larger slowdowns at replay time ($> 10\times$) for systems that log fewer events but infer the order of missing events [3]. In contrast, our debugger leverages the *structure* of datacenter computing frameworks to deterministically replay tasks. This approach allows us to catch a large class of logic and performance bugs, and although we cannot replay some of the nondeterministic bugs that other systems capture (*e.g.,* race conditions between threads in the same task), we can still detect them via checksumming. Our recording overhead is also low enough that event logging can be turned on by default in production.

11 Conclusion

As cluster programming frameworks are adopted for more applications, debugging the programs written in them is increasingly important. This is challenging both because of the scale of the applications and because of the cost of the hardware resources involved, which makes any runtime overhead for debug information expensive. We have proposed an approach based on *selective replay* that exploits the deterministic nature of computations in these frameworks to efficiently rerun parts of the program. We show that this approach enables a rich set of analyses, including rerunning tasks in a conventional step-through debugger, checking assertions, tracing records forward and back through the computation, and interactively querying intermediate results. The cost to log the operations we require is minimal (less than 4%), allowing our recording to be "always on" in produc-

tion use. The deterministic operations we leverage are a crucial element of current programming frameworks because they enable fault recovery [9], so we believe that they will remain present in future frameworks, making our approach applicable there as well.

References

[1] Apache Hadoop. `http://hadoop.apache.org`.

[2] Apache Mahout. `http://mahout.apache.org`.

[3] ALTEKAR, G., AND STOICA, I. ODR: output-deterministic replay for multicore debugging. In *SOSP* (2009).

[4] ALTEKAR, G., AND STOICA, I. DCR: Replay-debugging for the datacenter. Tech. Rep. UCB/EECS-2010-33, UC Berkeley, 2010.

[5] BORKAR, V., CAREY, M., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE '11* (2011), pp. 1151–1162.

[6] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI '10* (2010), ACM.

[7] CLAESSEN, K., AND HUGHES, J. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP '00* (2000).

[8] DE, S., LOGOTHETIS, D., AND YOCUM, K. Scalable lineage capture for debugging DISC analytics. OSDI poster session, 2012.

[9] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).

[10] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: global comprehension for distributed replay. In *NSDI '07* (2007), pp. 21–21.

[11] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *USENIX ATC* (2006).

[12] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: distributed graph-parallel computation on natural graphs. In *OSDI '12* (2012).

[13] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: an application-level kernel for record and replay. In *OSDI* (2008).

[14] IKEDA, R., PARK, H., AND WIDOM, J. Provenance for generalized map and reduce workflows. In *CIDR 2011*.

[15] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[16] JAGANNATH, V., YIN, Z., AND BUDIU, M. Monitoring and debugging DryadLINQ applications with Daphne. In *HIPS* (2011).

[17] LIANG, S., AND BRACHA, G. Dynamic class loading in the java virtual machine. In *OOPSLA '98* (1998), pp. 36–44.

[18] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD* (2010).

[19] MOSELEY, T., SHYE, A., REDDI, V. J., GRUNWALD, D., AND PERI, R. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07* (2007), pp. 198–208.

[20] OLSTON, C., AND REED, B. Inspector Gadget: a framework for custom monitoring and debugging of distributed dataflows. In *SIGMOD* (2011).

[21] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pp. 1099–1110.

[22] THOMAS, K., GRIER, C., MA, J., PAXSON, V., AND SONG, D. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy* (2011).

[23] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).