Debugging Big Data Analytics in Spark with *BigDebug*

Muhammad Ali Gulzar, Matteo Interlandi, Tyson Condie, Miryung Kim
University of California, Los Angeles , USA
{gulzar, minterlandi, tcondie, miryung}@cs.ucla.edu

ABSTRACT

To process massive quantities of data, developers leverage Data-Intensive Scalable Computing (DISC) systems such as Apache Spark. In terms of debugging, DISC systems support only post-mortem log analysis and do not provide any debugging functionality. This demonstration paper showcases BIGDEBUG: a tool enhancing Apache Spark with a set of interactive debugging features that can help users in debug their Big Data Applications.

1. INTRODUCTION

An abundance of data in many disciplines of science, engineering, national security, health care, and business is now urging the need for developing Big Data analytics. To process massive quantities of data in the cloud, developers leverage Data-Intensive Scalable Computing (DISC) systems such as Apache Hadoop [2], and Apache Spark [3]. In DISC systems, scaling to large datasets is handled by partitioning data and assigning tasks that execute a portion of the application logic on each partition in parallel. Unfortunately, this gain in scalability creates an enormous challenge for data scientists in understanding and resolving program errors.

The application programming interfaces (API) provided by DISC systems expose a batch model of execution: applications are run in the cloud, and the results, including notification of runtime failures, are sent back to users upon completion. Therefore, debugging is done *post-mortem* and the primary source of debugging information is an execution log. However, the log presents only the *physical view*—the job status at individual nodes, the overall job progress rate, the messages passed between nodes, etc, but does not provide the *logical view*—which intermediate outputs are produced from which inputs, what inputs are causing incorrect results or delays, etc. Alternatively, a developer may test their programs by downloading a small subset of big data from the cloud onto their local disk, and then run the application in local mode. However, this approach is not thorough and can easily miss errors when the faulty data is not part of the downloaded subset.

In this demonstration we showcase BIGDEBUG, a library providing expressive and interactive debugging features for Big Data analytics in Apache Spark [3]. This tool demonstration paper reiterates the technology from our previous system [6] and is based on our prior work on the design and implementation of interactive debugging and workflow analysis primitives and optimizations for Apache Spark [4, 7, 8]. Designing BIGDEBUG required rethinking the notion of breakpoints, watchpoints, and step-through debugging in a traditional debugger such as `gdb`. For example, simply pausing the entire computation would waste a large amount of computational resources and prevent correct tasks from completing, reducing the overall throughput. Requiring the user to inspect the millions of intermediate records at a watchpoint is also clearly unfeasible. To emulate interactive step-wise debugging without reducing throughput, BIGDEBUG provides **simulated breakpoints** that enable a user to inspect a program without actually pausing the entire computation. It also supports **on-demand watchpoints** that enable a user to retrieve intermediate data using a guard predicate and transfer the selected data on demand. To understand the flow of individual records within a data parallel pipeline, BIGDEBUG provides **data provenance** capability, which can help understand how errors propagate through data processing steps. To support efficient trial-and-error debugging, BIGDEBUG enables users to change program logic in response to an error at runtime through a **realtime code fix** feature and **selectively replay** the execution from that step. Finally, BIGDEBUG proposes an **automated fault localization** service that leverage all the above features together to automatically isolate failure-inducing inputs, diagnose the root cause of an error, and resume the workflow for only affected data and code. We think that the BIGDEBUG system will contribute in improving developer productivity and correctness of Big Data applications.

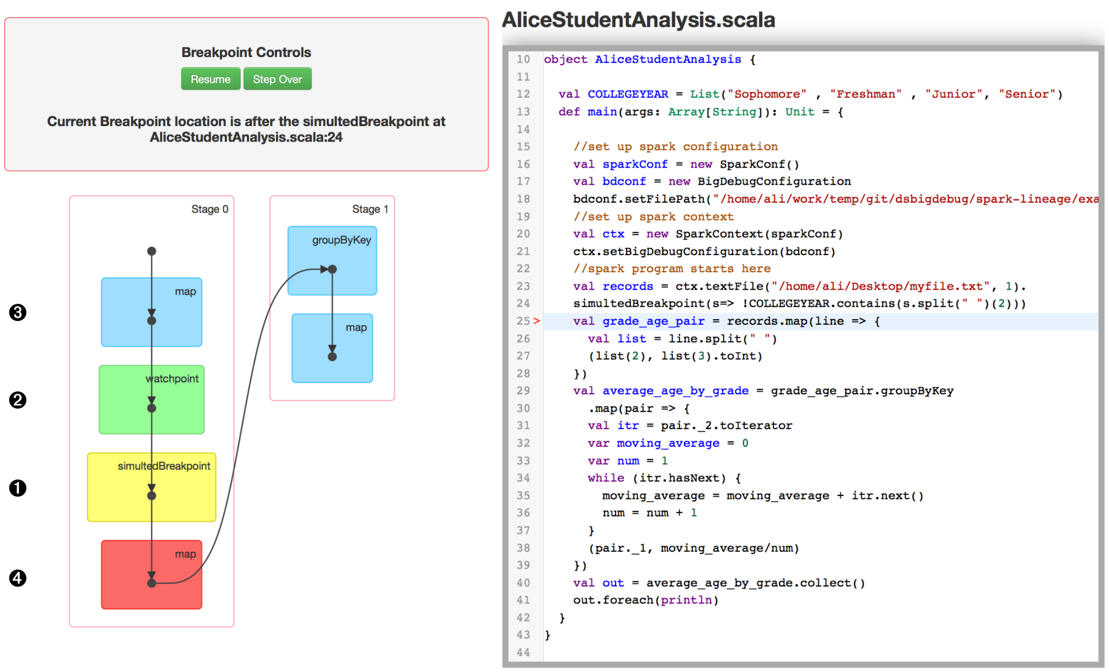
For readers interested in the performance overhead of each single feature packed into BIGDEBUG, we refer to our previous papers [4, 7]. The current version of BIGDEBUG is publicly available at [1].

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4197-4/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3035918.3058737

2. BACKGROUND: APACHE SPARK

BIGDEBUG targets Spark because of its wide adoption and support for interactive ad-hoc analytics. The Spark programming model can be viewed as an extension to the Map Reduce model with direct support of a large variety of operators (e.g., group-by, join, filter). Spark programmers leverage Resilient Distributed Datasets (RDDs) [9] to apply a series of transformations to a collection of data records stored in a distributed fashion *e.g.,* in HDFS.

Calling a transformation on an RDD produces a new RDD that represents the result of applying the given transformation to the input RDD. Transformations are lazily evaluated. The actual evaluation of an RDD occurs when an action such as `count` or `collect` is called. The Spark platform consists of three main entities: a *driver program*, a *master* node, and a set of *workers*. The master node controls distributed job execution and provides a rendezvous point between the driver and the workers. Internally, the Spark



Current Breakpoint location is after the simultedBreakpoint at AliceStudentAnalysis.scala:24

AliceStudentAnalysis.scala

```
10  object AliceStudentAnalysis {
11
12    val COLLEGEYEAR = List("Sophomore" , "Freshman" , "Junior", "Senior")
13    def main(args: Array[String]): Unit = {
14
15      //set up spark configuration
16      val sparkConf = new SparkConf()
17      val bdconf = new BigDebugConfiguration
18      bdconf.setFilePath("/home/ali/work/temp/git/dsbigdebug/spark-lineage/exa
19      //set up spark context
20      val ctx = new SparkContext(sparkConf)
21      ctx.setBigDebugConfiguration(bdconf)
22      //spark program starts here
23      val records = ctx.textFile("/home/ali/Desktop/myfile.txt", 1).
24      simultedBreakpoint(s=> !COLLEGEYEAR.contains(s.split(" ")(2)))
25>     val grade_age_pair = records.map(line => {
26        val list = line.split(" ")
27        (list(2), list(3).toInt)
28      })
29      val average_age_by_grade = grade_age_pair.groupByKey
30        .map(pair => {
31        val itr = pair._2.toIterator
32        var moving_average = 0
33        var num = 1
34        while (itr.hasNext) {
35          moving_average = moving_average + itr.next()
36          num = num + 1
37        }
38        (pair._1, moving_average/num)
39      })
40      val out = average_age_by_grade.collect()
41      out.foreach(println)
42    }
43  }
44
```

Figure 1: BIGDEBUG extends Spark's user interface to provide runtime debugging features

master translates a series of RDD transformations into a Directed Acyclic Graph (DAG) of *stages*, where each stage contains some sub-series of transformations, until a *shuffle step* is required (*i.e.,* data must be re-partitioned). The Spark scheduler is responsible for executing each stage in topological order, with *tasks* that perform the work of a stage on input partitions. Each stage is fully executed before downstream dependent stages are scheduled. The final output stage evaluates the action that triggered the execution. The action result values are collected from each task and returned (via the master) to the driver program.

3. DEMONSTRATION SCENARIO

```
1  val log = "s3n://xcr:wJY@ws/logs/enroll.log"
2  val text_file = spark.textFile(log)
3  val avg = text_file
4       .map(line = > (line.split()[2] ,
            line.split()[3].toInt) )
5       .groupByKey()
6       .map(v => (v._1 , average(v._2)) )
7       .collect()
```

Figure 2: College student data analysis program in Scala

In this section we will walk through the demonstration of BIGDEBUG with the help of Alice, an imaginary Spark user. Suppose Alice wants to process all US college student data. Because of the dataset massive size, she cannot store and analyze the data in a single machine. Suppose that she intends to compute the average age of all college students in each year (freshman, sophomore, junior, and senior) using the program of Figure 2. `1 Timothy Sophomore 21` is the format of a sample input record.

She starts by loading the US college student data from an Amazon S3 storage into the cluster (line 2). She the parses the data into appropriate key-value date types, where a key is the status category for a student and the value is the age of that student (line 4). Records are then grouped with respect to the key, and the average for each category is computed (lines 5 and 6). Finally, at line 7 she executes the job and requests the result to be sent to the driver.

Simulated Breakpoint and Guarded Watchpoint

To maximize the throughput in a big data debugging session, BIGDEBUG provides **simulated breakpoints** that enable a user to inspect a program state in a remote executor node without actually pausing the entire computation. When such breakpoint is in place, a program state is regenerated, on-demand, from the last materialization point, while the original process is still running in the background. The last materialization point refers to the last stage boundary before the simulated breakpoint. These materialization points are determined beforehand by Spark's scheduler.

To reduce developer burden in inspecting a large amount of intermediate records at a particular breakpoint within the workflow, BIGDEBUG's **on-demand guarded watchpoints** retrieve intermediate data matching a user-defined predicate and transfer the selected data on demand. Furthermore, BIGDEBUG enables the user to update the guard predicate at runtime, while the job is still running. This dynamic guard update feature is useful when the user is not familiar with the data initially, and she wants to gradually narrow down the scope of the intermediate records to be inspected.

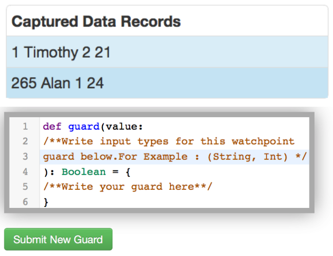
For example, suppose that Alice wants to inspect the program state at line 3. She can insert a simulated breakpoint using BIGDEBUG's API i.e., `simulatedBreakpoint(r => !COLLEGEYEAR.contains(r.split()[2]))` with the guard predicate indicating that the second field is not one of the pre-defined college years. The benefit of this breakpoint combined with the guarded watchpoint is twofold. First, Alice can now inspect intermediate program results distributed across multiple nodes on the cloud, which is impossible in the original Spark. Second, she can also inspect records matching the guard predicate only, which tremendously reduces the inspection overhead.

While the Spark program instrumented with breakpoints is running on the cluster, Alice can use a web-based debugger interface by connecting to a configured port. Using this interface, she can view the DAG of the data flow program. On the left hand side of Figure 1, the yellow node (❶) in the DAG represents a breakpoint. Alice can use the code editor window on the right hand side to see the Spark program in execution. Statements with a breakpoint are tagged using a red arrow.

Realtime Code Fix

After inspecting a program state at a breakpoint, if a user decides to patch code appearing later in the pipeline, she can use the **realtime code fix** feature to repair code on the fly. In this case, the original job is canceled and a new job is created from the last materialization point before the breakpoint. This approach avoids restarting the entire job from scratch. For example, in Figure 1, since a simulated breakpoint is in place (❶), BIGDEBUG records the last materialization point before the breakpoint, in this case, after `textFile`. While the job is still executing, Alice can inspect the internal program state at the breakpoint. She can click on the green node (❷) on the DAG, which redirects her to a new web page, where intermediate records are displayed. When she requests to view the internal program state, the captured records from the guarded watchpoint are transferred to the driver node and displayed as shown in Figure 3. Upon viewing the intermediate records at the breakpoint, Alice discovers that some records use number 2 instead of Sophomore to indicate the status year:

```
  1   Timothy   2   21
```
.


Captured Data Records

1 Timothy 2 21

265 Alan 1 24

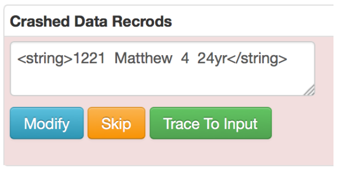
```
def guard(value:
/**Write input types for this watchpoint
guard below.For Example : (String, Int) */
): Boolean = {
/**Write your guard here**/
}
```

Figure 3: A user can edit the guard predicate using an editor.

From this outlier record, Alice immediately learns that her program should handle records with a status year written in numbers. To apply realtime code fix, Alice can click on the corresponding transformation (❸) marked in blue in the DAG. She can then insert a new user-defined function to replace the old one using the related code editor provided by the BIGDEBUG UI. The code fix can now handle status year both in number and string formats. When Alice presses a submit button, BIGDEBUG compiles and redistributes the new function to each worker node and restarts the job from the latest materialization point. When the job finishes its execution, the final result after the fix is shown to Alice. In addition to a realtime code fix feature, Alice can use *resume* and *step over* commands. These control commands are available in BIGDEBUG's UI.

Crash Culprit Remediation

In normal Spark, a runtime exception terminates the whole job, throwing away hours of computation while giving no information of the root cause of the error. When a Spark program fails with a runtime exception on the cluster, BIGDEBUG reports a **crash culprit** record in the intermediate stage but also identifies a **crash-inducing input(s)** in the original input data. While waiting for a user intervention, BIGDEBUG runs pending tasks continuously to utilize idle resources in order to achieve high throughput. If a crash occurs, the original job keeps on running, while the user is notified of the fine-grained details of the crash. Once the crash culprit is reported to the user, the user can choose among three **crash remediation options**. First, a user can choose to **skip** the crash inducing record. The final output, in this case, will not reflect the skipped records. Second, a user can **modify crash culprit records** in realtime, so that the modified record can be injected back into the pipeline. Third, a user can **repair code**. The whole process of modifying crash culprits is optimized through lazy remediation. While the user takes time to resolve crash culprits, BIGDEBUG continues processing the rest of the records, while also reporting any additional crashing record. More details about crash remediation methods are discussed elsewhere [4].


Crashed Data Recrods

<string>1221 Matthew 4 24yr</string>

Figure 4: Options provided by the crash remediation UI

Suppose that, after several hours of computation, a runtime exception occurs during the data processing. BIGDEBUG alerts Alice on the intermediate record responsible for the crash. These alerts turn the corresponding transformation node of the DAG to be red (❹ in our example workflow of Figure 1) and highlight the corresponding code line in the main editor window to be red as well. When Alice clicks on the red node (❹) in the DAG, she is redirected to the crash culprit page of Figure 4. This page contains the following crash culprit record:

```
1221   Matthew   4   24yr
```
.

While Alice is informed of the crash culprit record, BIGDEBUG continues executing the rest of the records and waits for the crash resolution from Alice. Alice may skip or modify the crash inducing intermediate record directly. Figure 4 shows the options provided on the UI to perform these remediation operations on the crash-inducing records. Alice skips the crashing record by pressing the *skip* button on the crash culprit UI. BIGDEBUG also allows the batch repair of modifying all crash-inducing records at once using a user-defined repair script.

Forward and Backward Tracing

BIGDEBUG supports **fine-grained tracing** of individual records by invoking a data provenance query on the fly. The *data provenance* problem refers to identifying the origin of final (or intermediate) output. Data provenance support for DISC systems is challenging, because operators such as `aggregation`, `join`, and `group-by` create many-to-one or many-to-many mappings for inputs and outputs and these mappings are physically distributed across different worker nodes. BIGDEBUG uses data provenance capability implemented through an extension of Spark's RDD abstraction [7]. Fine-grained tracing allows users to reason about the faults in the program output or intermediate results, and explain why a certain problem has occurred. Using backward tracing, a crash culprit record can be traced back to the original inputs responsible for the crash record. Forward tracing allows user to find the output records affected by a selected input.

For example, during crash remediation, Alice can invoke forward and backward tracing features at runtime to find the original input records responsible for the crash. On the crash culprit UI, Alice can invoke the backward tracing query by pressing the *trace to input* button. BIGDEBUG performs backward tracing in a new

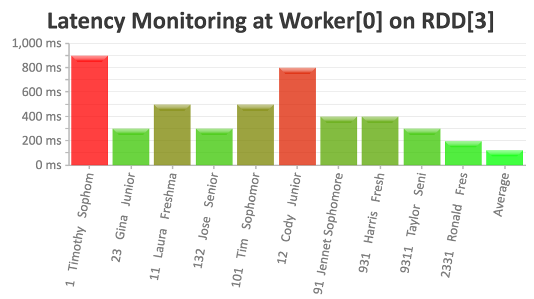


Figure 5: Top most straggling records are visualized in bar chart showing the delays relative to average

process to trace crash-inducing records in the original input data. Alice can also perform step-by-step backward tracing, showing all intermediate records tracing back to crash-inducing input records.

Fine-Grained Latency Monitoring

In big data processing, it is important to identify which records are causing delay. To localize performance anomalies at the record level, BIGDEBUG wraps each operator with a latency monitor. For each record at each transformation, BIGDEBUG computes the time taken to process each record, keeps track of a moving average, and sends a report to the monitor if the time is greater than k standard deviations above the moving average, where default k is 2.

Users can select a latency-enabled RDD from a drop-down menu; straggler records (i.e., delay-causing records) are then visualized in a streaming fashion. Figure 5 depicts a set of stragglers that Alice sees when enabling latency monitoring for RDD 3.

Automated Fault Localization

With automated fault localization, developers can isolate the trace of failure-inducing inputs and diagnose the root cause of an error. We equipped BIGDEBUG with the well known *Delta Debugging* [10] (DD) fault localization algorithm. DD performs repetitive runs on different configurations of input to systematically isolate the root cause of failures. The DD algorithm splits the original input into different sub-configurations using a binary search strategy and runs the same program with these sub-configurations as inputs. If one of the tests fails for a particular subset, it recursively applies the same procedure for only that input configuration. BIGDEBUG run a version of the DD algorithm specifically optimized for Big Data applications [5].

```scala
def test(record:Tuple2[Int,Float]):Boolean = {
  return 15< record._2 < 26
}
```

Figure 6: Test function to verify the validity of the output

Going back to our scenario, Alice perform the analysis depicted in program 2 and found out that the average age of sophomore-year students is 31.4 years which does not seem to be reasonable. In order to get the correct results from this analysis, Alice decides to understand the origin of the invalid output. Alice has different sets of tools at her disposal: data provenance for instance. The problem with data provenance is that millions of input records can be returned for this specific query. Alice hence decides to use DD on the inputs isolated by data provenance to localize the source of the fault. She writes a test function, depicted in Figure 6, which is used by delta debugging to identify if an input configuration returns an erroneous output. This test function checks the validity of average age value (known from prior knowledge). DD automatically and iteratively apply the test function over data splits until it outputs the precise source of the fault.

4. IMPLEMENTATION

All the features in BIGDEBUG are supported through the corresponding web-based user interface. BIGDEBUG extends the current Spark UI and provides a live stream of debugging information in an interactive and user-friendly manner. A screen-shot of this interface is shown in Figure 1. Instead of creating a wrapper around existing Spark modules to track the input and output of each stage, BIGDEBUG directly extends Spark to monitor pipelined intra-stage transformations. Additionally, BIGDEBUG's API extends the RDD interface. Instead of using the UI, a developer can therefore programmatically use function calls like `watchpoint()` and `simulatedBreakpoint()` on an RDD object to insert watchpoints and breakpoints. BIGDEBUG allows user to enable crash and latency monitoring on individual RDDs by calling appropriate methods on that RDD object. Tracing works at the granularity of each job and can be enabled or disabled through the `SparkContext` object. For instance, Alice could use tracing capabilities outside the crash culprit remediation feature, by programmatically use the tracing API, similarly to [7].

All debugger configuration and control commands are linked with a driver that broadcasts the debugger information to each worker. The runtime code patching is received and compiled at a driver and is then loaded into each worker, where an instrumented task is running. We think that the interactive nature, and double visual / programmatic flavor of BIGDEBUG enable users to better debug their Big Data applications *w.r.t.* post-mortem analysis.

5. CONCLUSION

BIGDEBUG offers interactive debugging primitives for an in-memory Data-Intensive Scalable Computing (DISC) framework. BIGDEBUG is able to scale to massive dataset of the order of terabytes [4], and thanks to its interactive set of features improves both Big Data application debugging experience, and fault localizability. BIGDEBUG is available for download at [1]. A version of this demo was previously published at FSE 2016 Research Tool Demonstration Track [6]; the goal of this paper is to advertise our work in the SIGMOD community.

6. ACKNOWLEDGMENT

This work is supported by NSF CCF- 1527923, CNS-1239498, CCF-1460325, CNS-1161595, IIS-1302698, CNS-1351047.

7. REFERENCES

[1] Bigdebug. https://sites.google.com/site/sparkbigdebug/.
[2] Hadoop. http://hadoop.apache.org/.
[3] Spark. https://spark.apache.org/.
[4] M. A. Gulzar and et al. Bigdebug: Debugging primitives for interactive big data processing in spark. In *ICSE*, 2016.
[5] M. A. Gulzar, X. Han, M. Interlandi, and et al. Interactive debugging for big data analytics. In *HotCloud*, 2016.
[6] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Bigdebug: Interactive debugger for big data analytics in apache spark. In *FSE*, pages 1033–1037, 2016.
[7] M. Interlandi, K. Shah, S. D. Tetali, and et al. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.
[8] M. Interlandi, S. D. Tetali, and et al. Optimizing interactive development of data-intensive applications. In *SoCC*, 2016.
[9] M. Zaharia and et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[10] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 2002.