BigDebug: Interactive Debugger for Big Data Analytics in Apache Spark

Muhammad Ali Gulzar, Matteo Interlandi, Tyson Condie, Miryung Kim
University of California, Los Angeles

ABSTRACT

To process massive quantities of data, developers leverage data-intensive scalable computing (DISC) systems in the cloud, such as Google's MapReduce, Apache Hadoop, and Apache Spark. In terms of debugging, DISC systems support post-mortem log analysis but do not provide interactive debugging features in realtime. This tool demonstration paper showcases a set of concrete use-cases on how BIGDEBUG can help debug Big Data Applications by providing interactive, realtime debug primitives. To emulate interactive step-wise debugging without reducing throughput, BIGDEBUG provides *simulated breakpoints* to enable a user to inspect a program without actually pausing the entire computation. To minimize unnecessary communication and data transfer, BIGDEBUG provides *on-demand watchpoints* that enable a user to retrieve intermediate data using a guard and transfer the selected data on demand. To support systematic and efficient trial-and-error debugging, BIGDEBUG also enables users to change program logic in response to an error at runtime and replay the execution from that step. BIGDEBUG is available for download at http://web.cs.ucla.edu/~miryung/software.html.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, distributed debugging, error handling and recovery*

Keywords

Debugging, big data analytics, interactive tools, data-intensive scalable computing (DISC), fault localization and recovery

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 Google's MapReduce [4], Apache Hadoop [1], and Apache Spark [13]. In these 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 critical gain in scalability creates an enormous challenge for data scientists in understanding and resolving 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 program 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 can easily miss errors, for example, when the faulty data is not part of the downloaded subset.

We showcase BIGDEBUG, a library providing expressive and interactive debugging for big data analytics in Apache Spark. This tool demonstration paper is based on our prior work [7] on the design and implementation of interactive debugging primitives for Apache Spark [2]. Designing BIGDEBUG requires re-thinking 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. As another example, requiring the user to inspect the millions of intermediate records at a watchpoint is clearly infeasible.

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 systematic and 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.

This paper is organized as follows. Section 2 describes the background on Apache Spark. Section 3 describes individual debugging features of BIGDEBUG using two motivating scenarios along with relevant screen snapshots. Section 4 describes the implementation details of BIGDEBUG. Section 5 describes related

FSE '16 Seattle, Washington USA
ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

```
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 1: College student data analysis program in Scala

work and Section 6 concludes. The appendix includes a walk through of our demonstration plan with associated screen snapshots. BIGDEBUG tool is publicly available at https://sites.google.com/site/sparkbigdebug/.

2. BACKGROUND: APACHE SPARK

Apache Spark [2] is a large scale data processing platform that achieves orders-of-magnitude better performance than Hadoop MapReduce [1] for iterative workloads. 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 for traditional relational algebra operators (e.g., group-by, join, filter). Spark programmers leverage Resilient Distributed Datasets (RDDs) to apply a series of transformations to a collection of data records (or tuples) stored in a distributed fashion *e.g.,* in HDFS [11].

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 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, which can initiate another series of transformations ending with an action.

3. MOTIVATING SCENARIOS WITH DEBUG FEATURES

Suppose Alice is a Spark user and she wants to process all US college student data. Because of the massive size of the data, 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). She starts by parsing the data into appropriate data types and then groups the records for each category. Once she has all related records grouped together, she computes the average and then collects the final results. A sample input record is in the following format:

```
1   Timothy   Sophomore   21
```

The final program that Alice has written is shown in Figure 1. At line 2, she loads the US college student data from an Amazon S3 storage in the cluster. Line 4 reads each data record in the input data and generates a key value pair, where a key is the status category for a student and the value is the age of that student. At lines 5 and 6, she groups all records with respect to the key and calculates the average for each category. 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.

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 (4040 by default). Using this interface, she can view the DAG of the data flow program. On the left hand side of Figure 2, a yellow node 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 colored in blue.

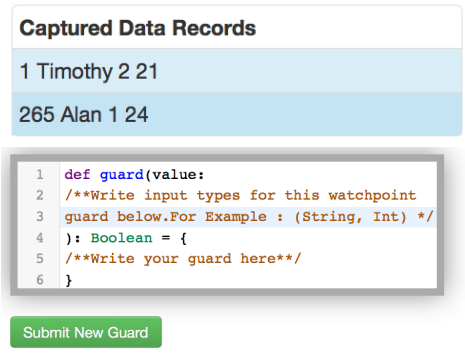

Figure 3: A user can edit the guard predicate using an editor window.

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

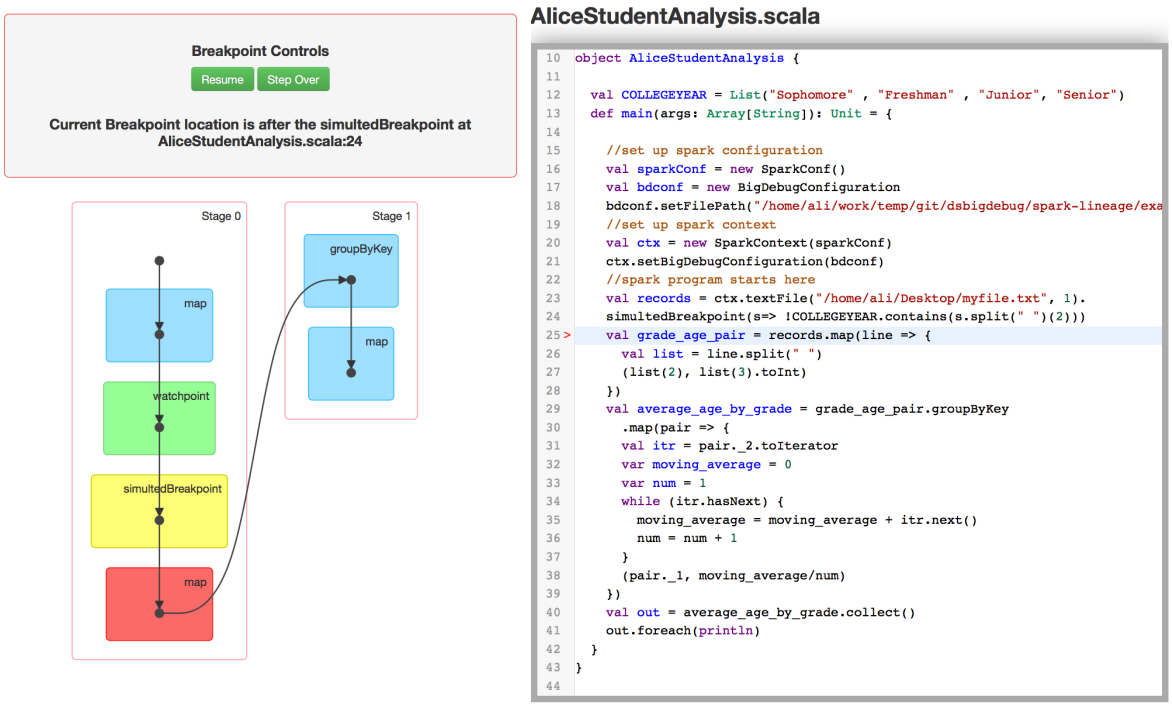

Figure 2: BIGDEBUG extends Spark's user interface to provide runtime debugging features

point before the breakpoint. This approach avoids restarting the entire job from scratch.

For example, in Figure 2, since a simulated breakpoint is in place, 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 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 a breakpoint, Alice discovers that some records use number 2 instead of Sophomore to indicate the status year.

```
1    Timothy    2    21
```

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, in this case, `map` transformation marked in blue in the DAG. She can then insert a new user-defined function to replace the old user-defined function using a code editor shown in Figure 3. 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 user-defined 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 may use `resume` and `step over` as control 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. This feature increases the robustness of the system.

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 and highlight the corresponding code line in the main editor window to be red as well. When Alice clicks on the red node in the DAG, she is redirected to the crash culprit page of Figure 4. This page contains precise and useful information about the following crash culprit record:

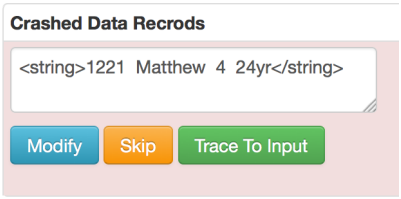

Figure 4: A user can either modify or skip the crash inducing records

```
1221    Matthew    4    24yr
```

When 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 enhances user experience by allowing batch repair as well.

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 in the database community 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 [8].

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 feature 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 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.

4. IMPLEMENTATION

The API for BIGDEBUG is shown in Figure 5 and targets Scala programs. All the features in BIGDEBUG is 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 screenshot of this interface is shown in Figure 2. Instead of creating a wrapper of existing Spark modules to track the input and output of each stage, BIGDEBUG directly extends Spark to monitor pipelined intra-stage transformations; its API extends the current RDD interface of Spark. A user can 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 a `LineageContext`. All debugger control commands are linked with a driver that broadcasts the debugger control 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.

```
1  //RDD.scala
2  abstract class RDD[T: ClassTag](
3  ....
4  def watchpoint(f: T => Boolean): RDD[T]
5  def simulatedBreakpoint
6  def simulatedBreakpoint(f:T => Boolean)
7  def enableLatencyAlert(set : Boolean)
8  def setCrashConfiguration(set :
       CrashConfiguration)
9  def setFunction(f : T => U)
10 def goBackAll: LineageRDD
11 def goNextAll: LineageRDD
12 def goBack: LineageRDD
13 def goNext: LineageRDD
14 ...
```

Figure 5: BIGDEBUG's API

5. RELATED WORK

Fisher et al. [5] interviewed 16 data analysts at Microsoft and studied the painpoints of big data analytics tools. Their study finds that a cloud-based computing solution makes it far more difficult to debug. Xu et al. [12] parse console logs and combine source code analysis to detect abnormal behavior. Fu et al. [6] map free-form text messages in log files to logging statements in source code. None of these post-mortem log analysis approaches help developers debug DISC applications in realtime.

Inspector Gadget [9] is a framework proposal for monitoring and debugging data flow programs in Apache Pig [10]. The proposal is based on informal interviews with 10 Yahoo employees who write DISC applications. While Inspector Gadget proposes features such as step-through debugging, crash culprit determination, tracing, etc., it simply lists desired debug APIs, but leaves it to others to implement the proposed APIs. Arthur [3] is a post-hoc instrumentation debugger that targets Spark and enables a user to selectively replay a part of the original execution. However, a user can only perform *post-mortem* analysis and cannot inspect intermediate results at runtime. It also requires a user to write a custom query for post-hoc instrumentation. To localize faults, Arthur requires more than one run. In our prior work, we describe the design and evaluation of interactive debugging primitives [7] and data provenance for Apache Spark [8]. This demonstration paper builds on these prior works to showcase the UI and tool features of BIGDEBUG.

6. FUTURE WORK

BIGDEBUG offers interactive debugging primitives for an in-memory data-intensive scalable computing (DISC) framework. In terms of future work, instead of having a user specify a guard for an on-demand watchpoint, extracting data invariants from intercepted intermediate results may be useful for helping the user debug a DISC program. Another area for future work is tool-assisted auto-

mated fault localization in BIGDEBUG. For example, with the help of automated fault localization, we envision that a user can isolate the trace of failure-inducing workflow, diagnose the root cause of an error, and explain the cause-effect chain for unexpected results.

REFERENCES

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

[2] Spark. https://spark.apache.org/.

[3] A. Dave, M. Zaharia, and I. Stoica. Arthur: Rich post-facto debugging for production analytics applications. Technical report, Citeseer, 2013.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker. Interactions with big data analytics. *interactions*, 19(3):50–59, May 2012.

[6] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, Washington, DC, USA, 2009. IEEE Computer Society.

[7] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[8] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[9] C. Olston and B. Reed. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1221–1224. ACM, 2011.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[11] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.

[12] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

APPENDIX

This appendix serves as a walk-through of BIGDEBUG using a simple *WordCount* program implemented in Spark. We will go over each debugging primitive with corresponding screenshots to help understand BIGDEBUG's features in detail. This demonstration will consist of the following steps:

- Designing a WordCount application in Apache Spark
- Simulating errors and delays
- Configuring BIGDEBUG
- Inserting necessary debugging primitives at compile time.
- Interacting with BIGDEBUG's UI at runtime
- Tracing records backward and forward through stages

A.1 Designing a WordCount Application on Spark

WordCount is the 'HelloWorld!' application for DISC systems. The program computes the frequency of unique words in a large text file. We start off by reading the data from a file (a local or `hdfs` file) using the Spark method `textFile(...)`. Spark partitions the file into smaller subsets and then reads the file line by line. We split each line into tokens of words and apply a `map` function to emit a record $(\mathsf{word}, 1)$ for each word. The count of the same word is accumulated and collected at the driver. The final version of the WordCount application is shown below.[1]

```
1  val conf = new SparkConf()
2  conf.setAppName("WordCount-"
       ).setMaster("spark://localhost:7077")
3  val sc = new SparkContext(conf)
4  val file = sc.textFile("README.md", 4)
5  val fm = file.flatMap(line =>
       line.trim().split(" "))
6  val pair = fm.map{word =>(word, 1)}
7  val count = pair.reduceByKey(_ + _)
8  count.collect().foreach(println)
```

A.2 Simulating Errors and Delays

To demonstrate the debugging features of BIGDEBUG, we introduce data-dependent crashes. To achieve this, we throw an exception if a record includes a particular string such as "ab". To demonstrate latency monitoring, we purposely introduce delays in the original WordCount program by making the program sleep for a few milliseconds, if a word contains "x".

```
1  ...
2  val pair = fm.map{word =>
3    if(word.contains("x")){
4      Thread.sleep(500* (word.length)) //
           Introduces a delay if a word contains
           "x" The length of delay is related to
           the word length of a data record
5    }
6    if(word.contains("ab")){
7      val str = null
8      str.toString // Throws a null pointer
           exception, if a record contains "ab"
9    }
10   (word, 1)}
11 val count = pair.reduceByKey(_ + _)
12 ...
```

[1]Current version of BIGDEBUG only supports self-contained Spark applications

A.3 Configuring BigDebug

BIGDEBUG enables a user to customize configuration settings. More details on configuration settings can be found at on-line API documentation page.[2] For this exercise, we will set the configuration with the default settings below.

```
1  val bdconf = new BigDebugConfiguration
2  bdconf.setCrashMonitoring(true) // Enable
       crash monitoring on all RDDs
3  bdconf.setCrashResolution("s") // Skip
       crashes, if faced any
4  bdconf.setLatencyUpdatePeriod(3000) // Set
       the latency request period every 3seconds
5  bdconf.setLatencyThreshold(2) // Mark a
       record as a straggler record, if its
       latency is 2standard deviations above the
       average time
6  bdconf.setTerminationOnFinished(false) // To
       avoid termination of UI after the job is
       finished
7  bdconf.watchpointViewThreshold(10) // Show
       only 10records at a watchpoint on UI at a
       time
```

A.4 Instrumenting a Program with Debug Primitives

We will insert two watchpoints one before and one after `reduceByKey`, enable record-level latency on one RDD (the `map` transformation) and enable fine-grained record tracing. To enable tracing in this example, we will use the `LineageContext` followed by calling `setCaptureLineage` with `true` as an argument. The on-line API documentation describes the exact usage of these features.

A.5 Interacting with BigDebug's UI at Runtime

At this point, our sample WordCount application is complete. We can now run this application in the cluster mode and start the debugging session using the debugger tab of the Spark UI (by default at `http://localhost:4040/debugger`). Once the scheduler schedules the Spark job and submits its tasks to the executors, we will see the UI shown in Figure 6.

BIGDEBUG's UI contains a wide range of debugger commands and visualizes the intermediate records being captured at the executor nodes. The right hand side of the UI contains a code editor view of the running application. The left hand side of the page provides a DAG visualization of all RDDs in the subject program. The color of the node represents the status of that RDD. WatchpointRDDs are colored in green, while other RDDs are colored in blue. If a crash occurs in one of the RDDs, its corresponding node will turn red. Since we introduce data-dependent crashes in the `map` transformation, the `map` node will eventually turn red. If we click on one of the RDD nodes in the DAG, we will be re-directed to another page containing further information about that RDD. A user may click on either a green, red, or blue RDD node in the DAG and she will be directed to a corresponding page, which is described below.

Click on a Green [Watchpoint] RDD Box. A green box represents a WatchpointRDD. Clicking on a green box takes users to a watchpoint RDD page. This page shows intermediate records captured using a watchpoint RDD in a table view. BIGDEBUG would feed captured data records into this table on the fly in a stream processing fashion. Moreover, at the bottom of the page, there is

[2]http://sites.google.com/site/sparkbigdebug/api

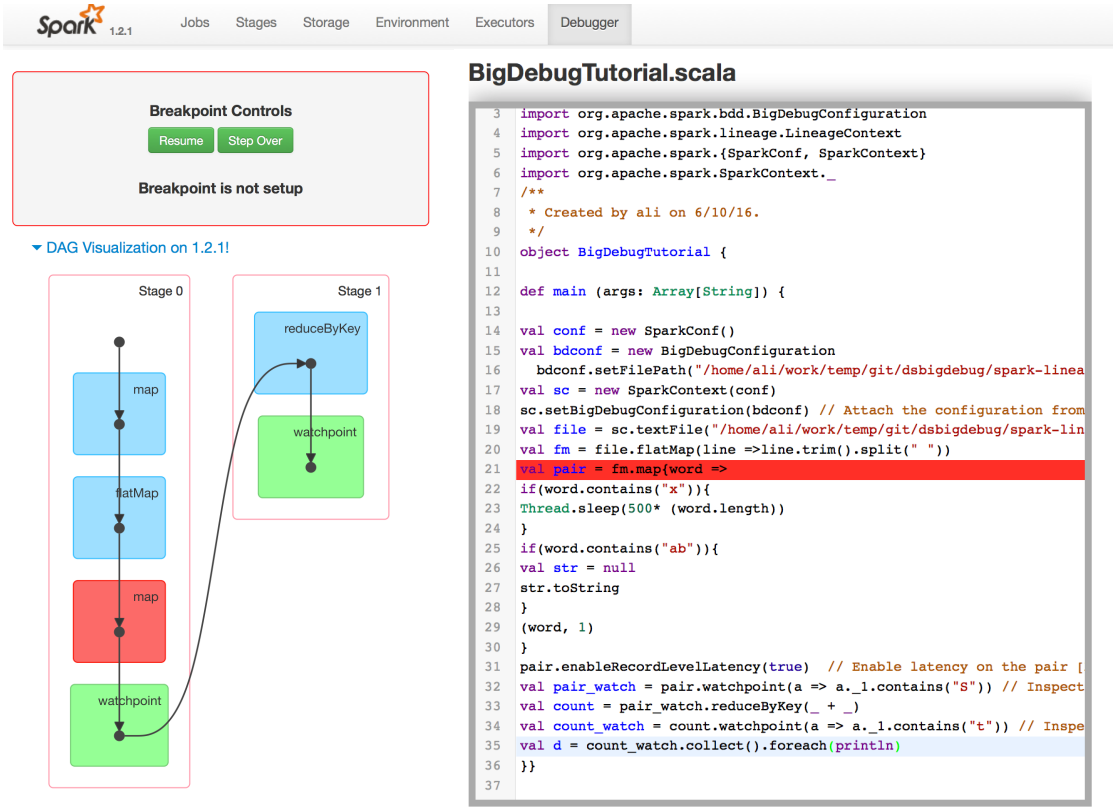

Figure 6: Screen snapshot of BIGDEBUG when a sample program WordCount is executed

a code box that takes an updated watchpoint guard as input from a user. This guard can be dynamically updated. If a new guard is written and submitted using the `Submit New Guard` button, the watchpoint RDD will be updated with the new guard. From there onwards, only those records that match the newly-supplied guard will be captured and shown to the user. Figure 7 shows the WatchpointRDD page. Furthermore, the UI displays the data records up to k number of records, according to BIGDEBUG's configuration setting. If a user needs to inspect the complete data captured by a watchpoint, she may use `Dump Watchpoint` feature to load all intermediate data matching the guard.

Click on Red [Crash]/ Blue RDD Box. When an RDD box turns red, it indicates that a crash has occurred while performing the transformation. In our sample application run, few seconds into the execution, the `map` RDD box turns from blue to red. This behavior is expected, because we introduced crashes in the user-defined function closure in the `map` transformation. After clicking the red box, the user will be redirected to the RDD page where she can see the crash culprits in `xml` format. In this demo, we configured BIGDEBUG such that it skips any crash culprit. Such configurations can be also seen in the UI. For every crash culprit, BIGDEBUG shows crash remediation actions. Figure 8 shows an example crash remediation view.

If a user configured BIGDEBUG to allow fixing the crashes manually at runtime by either skipping or modifying data records individually, the UI would be similar to Figure 9. A user can modify an individual record using an editable text editor and click the resolve button to inject it back into the running application. If the user chooses to skip a record, only that particular record will be skipped. BIGDEBUG's crash resolution option can be set to m (modify), lm (lazy modify), or lmb (lazy modify in batch). For example: `bdconf.setCrashResolution("lm")`

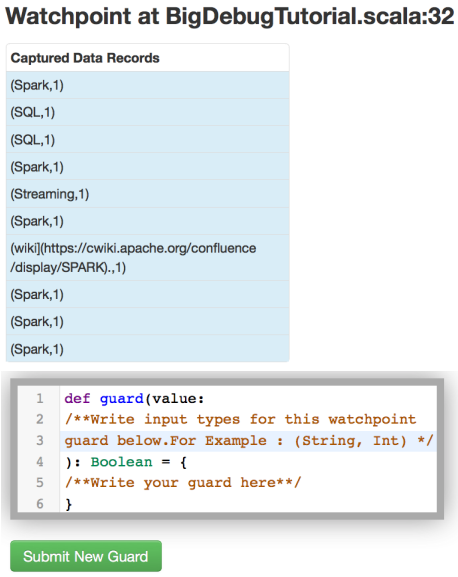

Figure 7: Watchpoint UI from the WordCount program

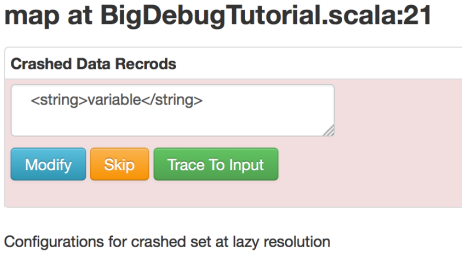

Figure 9: Crashed RDD UI from the WordCount program with "Lazy Modify" configuration

Figure 8: Crashed RDD UI from the WordCount program with "Skip" configuration

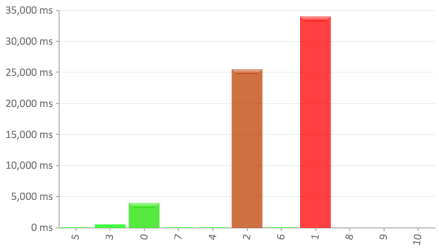

Figure 10: Task level latency is shown in a bar chart. A horizontal axis represents an individual task ID and a vertical axis represents its execution time.

Click on a column in Task Level Latency Chart. A task level latency chart such as the one in Figure 10 visualizes the time span of each task in our sample application. When running our WordCount program, if we set the minimum number of tasks to 4 `lc.textFile("README.md", 4)`, Spark splits the file into 4 partitions and subsequently 4 tasks. That is why we see 8 bars in the latency chart– 4 bars from stage 1 and 4 bars from stage 2. A column represents a task and the length of the column represents the execution time of each task. BIGDEBUG supports a fine-grained latency monitoring at the record level. To see fine-grained latency at the record level, a user can click on individual columns. On the executor's UI, a user can select individual RDDs in a drop down menu, on which she enabled latency alert. The RDD selection page on the executor is shown in Figure 11.

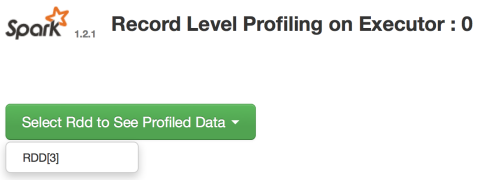

Figure 11: While inquiring record level latency, a user must select one of the latency monitoring enabled RDDs.

After a user selects a latency-enabled RDD from the drop-down menu, she can see individual straggler records (i.e., delay-causing records) in a streaming fashion. Only straggler records will be reported to the user. In Figure 12, the execution time is the time taken to apply a particular transformation on each record. Going back to our example, the simulated delays are only introduced, when an input intermediate record contains string "x" and the time delay is a function of the record length. The chart on BIGDEBUG's UI shows that the longest delay is from the longest string record.

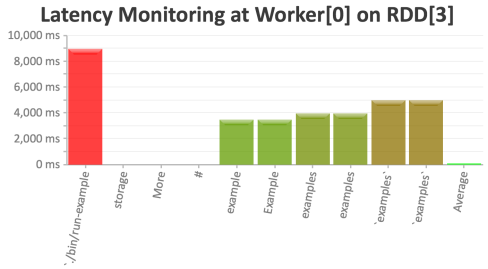

Figure 12: Top most straggling records are visualized in bar chart showing the delays relative to average

A.6 Backward and Forward Tracing

Tracing capabilities in BIGDEBUG is useful, when a user tries to investigate the origin of incorrect results (or observations). Backward tracing in BIGDEBUG helps a user identify the original input record(s) responsible for faulty output.

Backward Tracing. Suppose that a user picks a random output and marks it as faulty. Below is a subset of output records for our sample WordCount Spark application. When the lineage is active, BIGDEBUG attaches to each data record a lineage tag to help users in selecting and navigating the lineage information.

```
(("yarn-client",1),763862)
((computing,1),937268)
((application,1),231984)
((Python„2),432423)
((prefer,1),765865)
((example:,1),43245)
((##,8),100)
((other,1),43252)
((file,1),324343)
((building,3),634432)
```

Let's assume that `(##,8)` is faulty. We set the capture lineage to `false` so that no further tracing meta data is generated while answering data provenance queries, and we retrieve the lineage information as shown in the following code snippet.

```
1 /***
2   * Continuing from the our original code
3   * */
4 count_t.collect().foreach(println)
5 lc.setCaptureLineage(false)
6 var linRdd = count_t.getLineage()
```

Once we have the `LineageRDD` instance, we further `filter` to remove the correct outputs from the scope of our tracing.

```
1 linRdd.collect
2 linRdd = linRdd.filter(100) // Index of
      (##,8) is 100
```

`linRDD` contains only the faulty records. To investigate what we have at this location of the lineage, we can invoke `show`.

LineageRDDs are in fact references to a set of lineage tags; by invoking show on any LineageRDD object we can see the actual records linked to a given lineage tag. Since we filtered everything except (##,8), calling show on linRDD will display that record (##,8). If we invoke goBack on linRDD and invoke collect and show, BIGDEBUG will go one step backward in the DAG relative to the current RDD location and then collect tags at that point. The actual data records are then retrieved from the lineage tags by invoking show on the same LineageRDD.

```
linRdd = linRdd.goBack()
linRdd.collect
linRdd.show
```

The output from the above instructions is shown below.

```
((##,5),1120)
((##,3),1120)
```

In our sample application, the above output is from the end of the first stage after the map transformation, since that is one step back from the final result. BIGDEBUG ignores narrow dependencies in the DAG, while going back and forth. Narrow dependencies mean intra-stage dependencies between intermediate records, and such dependencies are not captured by the current data provenance support for Spark [8]. These are the only intermediate records that contributed to generating the faulty output. Repeating the above procedure will take us back all the way to the original input data contributing to the output record (##,8).

```
linRdd = linRdd.goBack()
linRdd.collect
linRdd.show
```

The output of above tracing instructions will show the following excerpt of the input data:

```
## Online Documentation
## Building Spark
## Interactive Scala Shell
## Interactive Python Shell
## Example Programs
## Running Tests
## A Note About Hadoop Versions
## Configuration
```

After inspecting the returned data from the lineage query, we can see all the 8 lines with "##" which contributed to the final output record of (##,8).

Forward Tracing. Forward tracing works in a similar fashion as backward tracing. To perform forward tracing, we must first retrieve the lineage reference for the input file RDD (or alternatively, and go all the way back to the original input records using the goBackAll() API).

```
linRdd = file.getLineage()
linRdd.show
```

Now let's say we want to perform forward tracing on the input record number 2. Since Spark reads a file line by line, record number 2 is the third line in the input file. We use the filter operation of LineageRDD and invoke collect and show to filter everything except the third line.

```
linRdd = linRdd.filter(2)
linRdd.collect.foreach(println)
linRdd.show
```

These statements, when executed, produce the following:

```
Spark is a fast and general cluster
computing system for Big Data.  It provides
```

Going next one step can be done by calling goNext on linRdd. This will take the current LineageRDD to one step forward in the DAG from the current location (the first RDD). After calling goNext, we will call collect and show.

```
linRdd = linRdd.goNext
linRdd.collect
linRdd.show
```

```
((is,4),3370)
((cluster,1),872092154)
((computing,1),1394183372)
((Big,1),66784)
((computing,1),1394183372)
((general,2),80148248)
((Data.,1),65803684)
((provides,1),987494926)
((fast,1),3135580)
((for,6),101577)
((system,1),887328209)
((a,4),97)
((It,2),2379)
((Spark,10),80085693)
((and,7),96727)
```

Repeating the previous step again displays a subset of the output records, originating from line 2. The output of the following data provenance query contains only those results dependent on line number 2.

Finally, if we call goNext and show one last time we can retrieve the output records generated by all the words composing the input line previously selected.

```
linRdd = linRdd.goNext
linRdd.collect
linRdd.show
```

The output from the above statements is the following:

```
((is,6), 8975)
((cluster,2), 4325324)
((computing,1), 53546543)
((Big,1), 543546)
((computing,1),1394183372)
((general,2), 28597454)
((Data.,1), 884328675)
((provides,1), 43785682)
((fast,1), 57853722)
((for,11), 4764782)
((system,1), 18457)
((a,9), 937563)
((It,2), 37865)
((Spark,14), 9756824)
((and,10), 97568333)
```