TagSniff: Simplified Big Data Debugging for Dataflow Jobs

Bertty Contreras-Rojas
Qatar Computing Research Institute
Data Analytics Group
Doha, Qatar
brojas@hbku.edu.qa

Jorge-Arnulfo Quiané-Ruiz
Qatar Computing Research Institute
Technische Universität Berlin
DFKI GmbH
jorge.quiane@tu-berlin.de

Zoi Kaoudi
Qatar Computing Research Institute
Technische Universität Berlin
DFKI GmbH
zoi.kaoudi@tu-berlin.de

Saravanan Thirumuruganathan
Qatar Computing Research Institute
Data Analytics Group
Doha, Qatar
sthirumuruganathan@hbku.edu.qa

ABSTRACT

Although big data processing has become dramatically easier over the last decade, there has not been matching progress over big data debugging. It is estimated that users spend more than 50% of their time debugging their big data applications, wasting machine resources and taking longer to reach valuable insights. One cannot simply transplant traditional debugging techniques to big data. In this paper, we propose the TagSniff model, which can dramatically simplify data debugging for dataflows (the de-facto programming model for big data). It is based on two primitives – tag and sniff – that are flexible and expressive enough to model all common big data debugging scenarios. We then present Snoopy – a general purpose monitoring and debugging system based on the TagSniff model. It supports both online and post-hoc debugging modes. Our experimental evaluation shows that Snoopy incurs a very low overhead on the main dataflow, 6% on average, as well as it is highly responsive to system events and users instructions.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

data debugging, dataflow systems, distributed systems, big data.

ACM Reference Format:
Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, and Saravanan Thirumuruganathan. 2019. TagSniff: Simplified Big Data Debugging for Dataflow Jobs. In *SoCC '19: ACM Symposium of Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

SoCC '19, November 20-23, Santa Cruz, CA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The dataflow programming model has become the *de-facto* model for big data processing. The abstraction of data processing as a series of high-level transformations on (distributed) datasets has been very influential. Users code their big data applications in a high-level programming model without caring about system complexities, such as node coordination, data distribution, and fault tolerance. The resulting code forms a dataflow, which is typically a directed acyclic graph (DAG): the vertices are transformation operators and the edges represent data flowing from one operator to the other. Almost all of the popular big data processing platforms, such as Hadoop [3], Spark [4], and Flink [1], support this programming model. It is not an exaggeration to claim that this approach was a key enabler of the big data revolution.

1.1 The State of Big Data Debugging

While big data *processing* has become dramatically easier in the last decade, the state of big *data debugging* is very much in its infancy. Debugging has always been a tedious and time-consuming task. It is estimated that users spend 50% of their time debugging their applications, resulting in a global cost of 312 billion dollars per year [8, 17]. This only gets exacerbated for (big) data debugging, which focuses on the finding and fixing errors caused by the intricate interplay between code and data. Data debugging is more like looking for a needle in a haystack.

Traditional debugging tools are inadequate for two main reasons. First, they are designed for code and not data debugging. Bugs in big data processing could stem from either the code or the data: although the code is correct, it still fails due to errors in the data, e. g., a null or malformed value. Second, they are not appropriate for *distributed* data debugging on multiple workers with a huge amount of intermediate data. Users typically debug their applications on a local machine and in a trial and error basis: They sample the data and follow some guidelines given by expert users.

The research community has recognized this problem and has carried out several attempts to tackle it [10, 12, 14, 16, 18]. However, the proposed solutions are often ad-hoc, task-specific, and not sufficiently flexible. Inspector Gadget [18] proposed a powerful debugging model based on monitors, coordinators, and drivers. While powerful, it is still challenging for non-expert users to write

their debugging tasks using the proposed APIs. BigDebug [12] tried to re-think the traditional debugging primitives and proposed their corresponding big data brethren: *simulated breakpoints* and *on-demand watchpoints*. Nonetheless, it requires extensive modification to the data processing systems, incurs considerable overhead, and does not provide support for post-hoc debugging. Arthur [10] introduced the concept of selective replay as a powerful tool for enabling common debugging tasks, such as tracing and post-hoc debugging. However, replay-based debugging approaches are limited to post-hoc debugging and hence do not support online debugging. Other works, such as Titian [14] and Newt [16], focus on efficiently implementing lineage for specific debugging tasks and hence cannot support a wider variety of debugging tasks.

1.2 Simplifying Big Data Debugging

Big data debugging is fundamentally very different from traditional code debugging. It thus requires a new suite of abstractions, techniques, and toolkits. In this paper, we make progress towards this elusive goal: We introduce the TagSniff model, an abstract debugging model with two powerful primitives and present Snoopy, an efficient implementation of the TagSniff model.

The TagSniff model. TagSniff is an abstract debugging model that is based on two primitives – tag and sniff – that are flexible enough to allow users to instrument their dataflows for their sophisticated debugging requirements. The tag primitive attaches annotations (tags) as metadata to a tuple if the tuple satisfies the user's conditions. The sniff primitive is used for identifying tuples requiring debugging or further analysis based on either their metadata or values. The flexibility of these primitives stems from the fact that users can specify their requirements through UDFs. TagSniff also comes with a set of convenience methods, which are syntactic sugar for users facilitating online and post-hoc debugging tasks. They internally use the tag and sniff primitives. We show that with TagSniff one can express most of the popular debugging scenarios.

An efficient implementation of TagSniff. Snoopy implements the TagSniff model in Spark. It uses wrappers on the vertices of the dataflow and injected Spark operators in the edges of the dataflow. The wrappers annotate (using the tag primitive) tuples in the dataflow. The sniffers can pull (using the sniff primitive) relevant information out of the main dataflow for remotely debugging the dataflow job. The goals of Snoopy are: (i) provide the TagSniff abstraction for users to easily instrument their applications, (ii) allow a wide variety of debugging tasks, (iii) allow users to add custom functionality for debugging data of interest, (iv) be as lightweight as possible to not affect the performance of the application dataflow, and (v) be portable to any underlying data processing system. A key characteristic of Snoopy is its novel architecture that enables both *in-place* and *out-of-place* debugging. TagSniff is built on top of Rheem [7], a cross-platform system, and thus does not require any modification of the underlying data processing platform.

The rest of the paper is organized as follows. Section 2 discusses the challenges and desiderata of big data debugging. Section 3 introduces the TagSniff model. Sections 4 and 5 explain how one can use the TagSniff model for online and post-hoc debugging. Sections 6 and 7 describe and evaluate Snoopy. Section 8 discusses related work and Section 9 concludes with final remarks.

2 MOTIVATION

We begin by enumerating the major debugging challenges encountered by programmers of big data applications when using traditional debugging approaches. We then discuss the changes needed for two major debugging modes – online and post-hoc. By synthesizing various user studies [18, 23] and prior work [10–12, 20], we identify the desiderata for big data debugging.

2.1 The Changing Face of Debugging

Frameworks like Spark have made big data *processing* much easier. However, big *data debugging* is still at its infancy. Suppose an analytic task on a terabyte of data failed to produce the expected results. There are two common, but ineffective, approaches to debugging this analytical task:

(i) The first approach brings the tools developed for "small data" debugging to big data. So, one could attach a debugger to a remote Spark process and try the traditional mechanisms, such as issuing watchpoints, pausing the Spark runtime, and stepping the code line by line. This approach is expensive as it results in the pausing of the entire Spark runtime. Furthermore, due to the sheer size of the data, one cannot simply step through the code and watch the intermediate results for each tuple. Doing so is extremely time-consuming.

(ii) The second approach tries to evaluate the task on a local machine over a sample of the input dataset. This is based on the fact that erroneous outputs are typically triggered by a small fraction of data. Therefore, one could take a sample of the input dataset and evaluate it on a local machine. If the sample does not trigger the issue, try a larger sample and so on. Eventually, the data becomes too large to hold in a single machine and/or use traditional debugging techniques. This approach is doomed to fail too.

We make the following three observations:

(1) Most of the bugs are often caused by the interplay between code and data. Traditional debugging tools are designed for *code* debugging and not *data* debugging.

(2) Traditional debugging tools are not appropriate for *distributed* debugging. Typical data processing jobs involve hundreds of tasks that are run on dozens of workers generating a huge amount of intermediate data.

(3) Recent attempts for big data debugging are ad hoc, task-specific and inflexible. There is a need for an abstraction that can address the code-data distributed debugging while hiding the internal complexity of the system.

2.2 Debugging Modes

We distinguish between two major modes, *online* and *post-hoc*, for debugging big data jobs.

Online mode. Online debugging happens when the main dataflow job is still alive. Users can inspect intermediate results and do trial-and-error debugging. Providing such verisimilitude is quite challenging as popular data processing systems operate in a batch mode. If one pauses the dataflow job, this could potentially pause the computation done by thousands of workers. This results into reduced throughput and wastage of processing resources. Ideally, the online

Table 1: Desired debugging tasks.

Debugging mode	Task	Description
Online	crash culprit	When a crash is triggered, return the tuple, the operator and the node that caused it.
	pause	Allow the user to pause execution (virtually or truly) when a certain (user-defined) condition is met and step through, either to go the next tuple or to go to the next operator for the same tuple.
	alert	Alert the user when a certain (user-defined) condition is met. Conditions can be on a single tuple, on a set of tuples or on a latency metric.
Post-hoc	replay	Replay the execution of the entire or part of the main dataflow job.
	trace	Forward or backward trace of tuples: given a tuple t, find all tuples that either stem from t (forward) or led to t (backward).
	profile	Profile any kind of metric, such as data distribution, latency distribution, runtime overhead, and memory usage.
	assert	Evaluate if the input or output tuples satisfy certain assertions, which is also useful for comparison with ground truth input/output tuples.

mode should: (i) allow a user to inspect intermediate results with or without pausing the dataflow execution, and (ii) provide a set of primitives so that a user can select intermediate data relevant for debugging programmatically. Very few systems [12] provide support for online big data debugging.

Post-hoc mode. This is the most common mode for big data debugging. Users instrument the main dataflow job to dump information into a log. One can then write another job (e. g., in Spark) to analyze the log and identify the issue. While common, this approach of using log files is often not sufficient. This is because a *logical view* [12] is not available in the logs, such as which input records produce a given intermediate result or the eventual output (i. e., lineage). This information is often invaluable for effective debugging. Ideally, the post-hoc mode should allow a user to (i) get the logical view of the job without any effort and (ii) provide an easy way to express common post-hoc debugging scenarios. Very few systems provide extensive support for post-hoc debugging. Most of them support specific scenarios, such as lineage [16] or task replay [10], and cannot be easily generalized to others.

2.3 Desiderata

Common debugging tasks. Based on various user studies [18, 23] and prior work [10–12, 20], we identify the most popular debugging tasks in Table 1 and grouped them in seven major categories. Very few systems can support all of them. Typically, the users roll their sleeves and implement task-specific variants of these common tasks at a significant development cost.

Desiderata for primitives. The requirements for primitives include (i) concise enough to handle the scenarios from Table 1, (ii) be flexible enough to handle customized debugging scenarios, (iii) provide support for both monitoring and debugging.

Desiderata for a debugging system. To be an effective tool for big data debugging, it must (i) provide holistic support for the debugging primitives, (ii) handle common debugging scenarios with no changes to the main dataflow job, (iii) allow users to add custom functionality for identifying tuples of interest, (iv) have detailed granularity at different levels (machine, dataset, and tuple level), (v) have very low overhead to the main dataflow job, and (vi) be generic to common big data processing systems without modifying them.

Table 2: An example of tuple tags.

Tag	Description
crash	Caused the dataflow to fail
debug	Requires online debugging
display	Needs to be displayed to the user
log	Has to be stored in a log
pause	Requires the dataflow execution to pause
trace	Needs to be tracked through the execution
skip	Has to skip the remaining transformations

3 THE TAGSNIFF MODEL

We introduce the *tag-and-sniff* debugging abstraction, TagSniff for short. TagSniff provides the dataflow instrumentation foundations for supporting most online and post-hoc debugging tasks easily and effectively. It is composed of two primitives, tag and sniff, that operate on the *debug tuple*. A unique characteristic of these primitives is that users can easily add custom debugging functionality via user defined functions (UDF). In the following, we call *TagSniff system* any system that implements this abstract debugging model.

Example 1 (Running example: Top100Words)**.** *We consider the task of retrieving the top-100 most frequent words. The following listing provides the (slightly simplified) Spark code:*

```
1  val tw = textFile.flatMap(l => l.split(" "))
2  val wc = tw.map(word => (word, 1))
3  val wct = wc.reduceByKey(_ + _)
4  val top100 = wct.top(100)
```

Listing 1: Top-100 frequent words (Top100Words).

3.1 Debug Tuple

Let us first define the debug tuple on which our primitives operate. A debug tuple is the tuple[1] that flows between the dataflow operators whenever debugging is enabled. For example, in Listing 1 of Example 1, datasets tw, wc, wt, and top100 would contain debug tuples in debug mode. A debug tuple is composed of the original tuple prefixed with annotations and/or metadata: <|tag$_1$|tag$_2$|..., <tuple>>. Typically annotations describe how users expect the system to react, while metadata adds extra information to the tuples, such as an identifier. Table 2 illustrates an example set of annotations. For simplicity, we refer to both annotations and metadata as tags. Tags are inserted by either users or the debugging system and mainly stem from dataflow instrumentation. The users can manipulate these tags to support sophisticated debugging scenarios, e. g., lineage. To enable this tag manipulation, we provide the following methods on the debug tuple:

▹ `add_tag (tag: String): Unit`: takes as input a string value and appends it in the tags of the debug tuple.
▹ `get_tag (tag: String): String`: returns all the tags that start with the input string value.
▹ `has_tag (tag: String) : Boolean`: takes as input a string value and returns true if this value exists in the tags of the tuple.
▹ `get_all () : String`: returns all the tags (annotations and metadata) of the debug tuple.

For simplicity reasons, we henceforth refer to a debug tuple simply as tuple.

3.2 Tag and Sniff Primitives

Our guiding principle is to provide a streamlined set of instrumentation primitives that make common debugging tasks easy to compose and custom debugging tasks possible. We describe these primitives below:

▹ `tag (f: tuple => tuple)`: It is used for adding tags to a tuple. The input is a UDF that receives a tuple and outputs a new tuple with any new tags users would like to append. A TagSniff system should then react to such tags.
▹ `sniff (f: tuple => Boolean)`: It is used for identifying tuples requiring debugging or further analysis based on either their metadata or values. The input is a UDF that receives a tuple and outputs true or false depending on whether the user wants to analyze this tuple or not. A TagSniff system is responsible for reacting to the sniffed tuples based on their tags.

A TagSniff system can materialize this abstract model in many different ways. We believe that two non-intrusive approaches for exposing the tag and sniff primitives is to specify them as *annotations* or additional *methods* in the dataflow. The system should then handle these annotations or methods to convert them to the appropriate code. This results in very little intrusion in the main dataflow while still being easy to add.

3.3 Examples

Let us now present a couple of debugging tasks whose instrumentation can be expressed with the TagSniff model without writing a huge amount of boilerplate code.

[1]A tuple is any kind of data unit, e. g., a line text or a relational tuple.

Example 2 (Data Breakpoint). *Suppose the user wants to add a data breakpoint in Listing 1 for tuples containing a null value to further inspect them. She would then write the* tag *and* sniff *primitives as follows:*

```
1  tag(t => if (t.contains(null)) t.add_tag("pause"))
2  sniff(t => return t.has_tag("pause"))
```

Listing 2: Add a breakpoint in tuples with *null* values.

Example 3 (Log). *Suppose the user wants to log tuples that contain null values to be used for tracing later on. She would then need to generate an identifier for each tuple and add it to the tuple's metadata. This could be done in the* tag *primitive, while the* sniff *primitive would simply detect such tuples. Notice that the user can use an external library to generate her own tuple identifiers.*

```
1  tag(t => if (t.contains(null)) {
2          id = Generator.generate_id(t)
3          t.add_tag("id-"+id)
4          t.add_tag("log")})
5  sniff(t => return t.has_tag("log"))
```

Listing 3: Log tuples with *null* values for tracing.

The above two examples show that users can instrument their dataflows using the tag and sniff primitives only, without writing huge amount of boilerplate code.

3.4 Discussion

Note that, as our goal is to keep the model as simple as possible, we defined TagSniff at the tuple granularity only. The reader might then wonder how to use TagSniff on a set of tuples, i. e., tagging and sniffing a set of tuples that satisfies a certain condition. This is possible if the dataflow job itself contains an operator grouping tuples, such as reduce, group, or join. Otherwise, one would have to modify the dataflow or create a new one to check if some conditions on a set of tuples hold. We consider this task as a data preparation/-cleaning task, and not data debugging, and is thus out of the scope of our framework. Still, one could do such checks with TagSniff in a post-hoc manner (i. e., after the dataflow execution terminates) as we will see in Section 5. To sum up, TagSniff is abstract enough to be implemented at any granularity: from one tuple to a set of tuples; from one operator to a set of operators; from one worker to a set of workers.

4 ONLINE DEBUGGING

Online debugging takes place while the job is still running. Thus, interactivity is crucial for online debugging as it allows users to (i) add breakpoints for data inspection, (ii) be notified with the appropriate information when a crash is triggered, and (iii) be alerted when certain conditions on the data are met. In contrast to traditional code debugging, interactivity in big data applications is mainly about the interplay between data and code. Therefore, new interactivity functionalities are required.

In the following, we demonstrate the power of TagSniff by describing how it can be used for the three scenarios above. In particular, we discuss how a TagSniff system should react to specific tag and sniff calls to support online debugging scenarios. We present how a user can debug a job in a post-hoc manner in the next section.

4.1 Data Breakpoints

Dataflow jobs are typically specified as a series of operators that perform pre-defined transformations over the datasets. That is, whenever an interesting tuple arrives and the dataflow pauses (either virtually or truly), a user would like to proceed by further inspecting how (i) an operator affects tuples, and/or (ii) a tuple is transformed by the rest of the dataflow. For this reason, we advocate two interactivity actions: *next tuple* and *next operator*.

Next tuple by TagSniff. Suppose that the dataflow is paused on the first tuple containing a null value by providing the tag and sniff functions of Listing 2. In other words, the user is interested in inspecting tuples containing a null value. Once the user has finished inspecting a given tuple, one has to show the next tuple that matches the user defined constraints. Showing the next tuple in the dataset instead – as done by traditional debugging – is not appropriate. We now describe how this functionality could be achieved using TagSniff. Once a TagSniff system receives the next tuple instruction, it should remove the tag pause from that tuple and send it to the next operator. This resumes the execution. The TagSniff system would then apply the tag and sniff function to the next incoming tuple in the inspected operator. If it satisfies the user condition (in this case it contains a null value), the dataflow execution is paused again. This results in having the dataflow execution being resumed and paused at any tuple satisfying the tag conditions.

Next operator by TagSniff. Suppose now the user wants to resume a paused dataflow by checking how the tuple, which caused the dataflow to pause, is transformed by the downstream operators. Again, she can achieve this with the sniff function of Listing 2. A TagSniff system would simply propagate the tag pause together with the tuple in order to pause the execution with the sniff function in the downstream operator. Thus, this functionality is relevant when users want to "follow" tuples and observe how they are being transformed by the operators in the dataflow.

Interactivity convenience methods. To facilitate users who want to use the next tuple and next operator tasks, we propose two convenience methods that a TagSniff system could provide: the next_tuple( ) and next_operator( ). Internally, they instantiate the tag and sniff primitives as discussed above. Note that the system could expose these convenience methods to users via a debugging user interface: a graphical one, where these methods are ideally implemented as built-in buttons, or as a command-line one.

4.2 Crash Culprit

A crash culprit is a tuple that causes a system to crash. In a dataflow job, a crash culprit causes an operator, and hence the entire dataflow, to crash. The objective is, thus, to identify not only the tuple but also the operator and node where a runtime exception occurs.

Crash culprit by TagSniff. Whenever a runtime exception occurs, a TagSniff system should catch the exception and invoke the tag primitive. The latter annotates the tuple with the tag "crash" as well as with the exception trace TRC, the operator id OID, and the node IP address. Then, the system invokes the sniff primitive to identify this tuple by inspecting for the crash tag. Note that these tag and sniff instances are specified by the TagSniff system and not by the user. We illustrate these two instances below:

```
1  tag(t => t.add_tag("crash-"+TRC+":"+OID+":"+IP))
2  sniff(t => return t.has_tag("crash"))
```

Listing 4: Catch crash culprits.

4.3 Alert

An alert functionality notifies a user that a tuple satisfied some condition of interest to the user. Users can add conditions on a single tuple or set of tuples as well as on information computed at runtime, e. g., on a latency metric.

Alert by TagSniff. Assume a user wants to be notified in the Top100Words example whenever there is a group of words that takes too long to be processed as this can be a potential bottleneck. This is possible with a tag primitive that adds a timestamp to the tuple metadata. A TagSniff system should then call this primitive *before* and *after* a tuple is executed by the ReduceByKey operator. The sniff primitive would then retrieve the timestamp metadata from the debug tuple to get the first and second timestamps, compute the latency of the ReduceByKey invocation and check if it is above some threshold. Listing 5 illustrates these primitives:

```
1  tag(t => t.add_tag("timestamp-"+ System.currentTimeMillis()))
2  sniff(t => {timestamps = put_in_array(t.get_tag("timestamp"))
3        return (timestamps[1] - timestamps[0] > THRESHOLD)})
```

Listing 5: Identify performance bottlenecks.

5 POST-HOC DEBUGGING

Post-hoc debugging takes place on the execution logs once the main dataflow job finishes. As mentioned previously, simple execution logs only provide a simplistic view where the input, intermediate, and output tuples are decoupled. Here, we describe how users can leverage the TagSniff primitives to produce much richer execution logs with a logical view. Users can then analyze these logs to identify the underlying issue. This calls for new querying functionalities that facilitate the analysis of rich execution logs. For example, obtaining lineage information or replaying a part of the dataflow execution for a subset of tuples might require quite some coding expertise. Although TagSniff can support a wide variety of post-hoc debugging tasks, our exposition focuses on how one can achieve each of the common post-hoc tasks listed in Table 1.

Similar to the online debugging cases described in the previous section, here we discuss how a TagSniff system should react to specific tag and sniff calls to support post-hoc debugging. We also introduce a set of convenience methods that prevent users from writing many lines of code. There are many ways in which a TagSniff system could expose these post-hoc convenience methods. Depending on the dataflow language used by the user, these methods can be special keywords in case of a declarative language, such as Pig Latin [19], or operators in case of a programmatic language. For example, one could write a Spark-like extension for these methods, which a TagSniff system should parse. We opted for the latter choice. In the following, we thus present our illustrative examples assuming the latter choice.

5.1 Forward and Backward Tracing

Intuitively, forward tracing allows users to identify which output tuples were generated from a given input tuple. More generally, this

process allows users to understand how a given tuple is transformed by various operators in the dataflow. Conversely, backward tracing allows users to identify the input tuple(s) that generated a given output tuple, which could be construed as a special case of lineage. Note that both forward and backward tracing could be executed on the entire dataflow or a portion of it.

Forward tracing with TagSniff. Suppose a user wants to trace an input tuple throughout the entire dataflow if it contains an empty word. Using the logs, the user can either run an ad-hoc dataflow or run the original dataflow properly instrumented with TagSniff. We argue the latter is much simpler. The tag primitive annotates all tuples containing an empty value as trace, otherwise as skip. A TagSniff system would apply this tag function at the source operator followed by a sniff function. This sniff function returns true for all tuples because each of them requires the system to act: either display the tuple to the user (trace) or remove the tuple from the dataflow (skip). The TagSniff system would also apply this sniff function in all the following operators to keep displaying the tuples. Note that if an operator is many-to-one (e. g., an aggregate), sniff will return true if any of the input tuples have the tag trace attached to it. Listing 6 illustrates the above tag and sniff:

```
1  tag(t => if (t.equals(" ")) t.add_tag("trace")
2     else t.add_tag("skip"))
3  sniff(t => return true)
```

Listing 6: Trace words with an empty string.

Backward Tracing with TagSniff. Suppose now a user wants to trace which input tuples contributed to a specific output tuple. If the execution log has enough information from the main dataflow instrumentation, the user can simply run an ad-hoc dataflow on the log to find out the contributing input tuples. If not enough lineage information is available, the user can then achieve the same by running the original dataflow properly instrumented with TagSniff on the logs. A TagSniff system would proceed to execute the dataflow in reverse from the last operator back to the first operator: It would fetch from the log all the tuples output by the reduceByKey operator, i. e., dataset wct in Listing 1, and run the top operator; It would then tag each tuple in wct with a unique identifier and trace them to identify the input tuples that contributed to the specific output tuple; Once the contributing tuples are identified, it would repeat this process until the first operator (i. e., flatMap) in the dataflow is reached.

Tracing convenience methods. To prevent users of writing a lot of code, we propose two convenience methods that a TagSniff system can provide: the forward_trace() and backward_trace(). These two convenience methods use internally the tag and sniff primitives as explained above. Let us illustrate how a user would express the backward tracing task explained above with the backward_trace() method. Listing 7 provides an idealized demonstration of how the user can do this.

```
1  val r = new Reader("debugging.log")
2  val ds = r.get_dataset(4)
3  val es = ds.filter(pair => pair[0] == " ")
4  val lines = es.backward_trace()
5  lines.collect().foreach(println)
```

Listing 7: Backward Tracing of Empty String

The user utilizes the Reader class to read and parse the execution log. She then uses the function get_dataset, which allows her to get any of the intermediate datasets: either by using the code line number of the main dataflow (Listing 1 in this example) or the operator identifier. Once she got the output dataset (top100), she filters it out by retaining only the empty words. Next, she applies the function backward_trace to track all these empty words (i. e., dataset es) all the way to the beginning. She could also trace only a limited number of steps by passing an argument to this convenience method. Finally, she prints all the input tuples that produced an empty word.

5.2 Selective Replay

Selective replay allows a user to replay portions of the dataflow graph. Selective replay has several applications [10], such as understanding how a subset of the dataset is affected by the dataflow, performing interactive queries on intermediate datasets for debugging, and re-executing part of the workflow with modified inputs.

Selective replay with TagSniff. Suppose that a user is interested in selectively replaying the execution of the map and reduceByKey operators (lines 2-3 in Listing 1). To achieve this, she can load the intermediate output of line 1 (i. e., dataset tw) and run the rest of the original dataflow instrumented with TagSniff. A TagSniff system would add a tag and sniff primitive after the reduceByKey operator, where the tag primitive adds the skip tag to all tuples output from that operator and sniff returns true for all tuples. The TagSniff system would then be responsible to remove all tuples tagged as skip from the main dataflow. This would lead the system to halt with the output of the reduceByKey operator. Listing 8 illustrates these tag and sniff functions:

```
1  tag(t => t.add_tag("skip"))
2  sniff(t => return true)
```

Listing 8: Skip all tuples after ReduceByKey**.**

Replay convenience method. To facilitate this task we propose the replay convenience method: replay(`from=start_line, until=end_line`), which is built on top of the TagSniff model. Listing 9 provides an idealized demonstration of how the user could use this method for the above example.

```
1  val r = new Reader("debugging.log")
2  val ds = r.get_dataset(1)
3  val output = ds.replay(from=2, until=3)
```

Listing 9: Selective replay.

5.3 Post-hoc Assertions

One can apply post-hoc assertions on input, intermediate, or output datasets to verify if a given condition is satisfied. Note that this could be combined with other methods to perform more sophisticated debugging, such as comparing the final output against a known ground truth.

Post-hoc assertions with TagSniff. Suppose that a user wants to verify if all strings passed as input to the map function in our WordCount running example (Listing 1) had at least length 1. This could be easily achieved with TagSniff where tag applies the display tag to tuples that fail to satisfy this condition. Then sniff returns

true for all tuples that contain the display tag so that a system shows them to the user for further inspection.

```
tag(t => if (len(t) < 2) t.add_tag("display"))
sniff(t => if (t.has_tag("display"))
    return true)
```

Listing 10: Skip tuples that failed an assertion.

Assert convenience method. We propose the convenience method assert (`tuple => Boolean`) also built on top of the TagSniff model. Listing 11 illustrates how this convenience method could be used for the above scenario.

```
val r = new Reader("debugging.log")
val ds = r.get_dataset(1)
val dse = ds.assert(w => len(w) < 2 )
dse.collect().foreach(println)
```

Listing 11: Post-hoc assertion.

5.4 Performance Profiling

Performance profiling is the task of analyzing execution logs to understand the dataflow footprint in terms of different performance metrics. For example, knowing the latency and throughput at either the tuple or operator level.

Performance profiling with TagSniff. A particular interesting scenario for performance profiling is straggler tuples, a pernicious problem in big data analytics. Most data processing systems, such as Spark, only provide coarse monitoring support at the job and worker level. Often, it is important to know how long processing each tuple took so that bottlenecks could be identified. This can be achieved by running an ad-hoc dataflow on the logs in case the logged tuples contain proper timestamps. If not, one could perform selective replay with the tag and sniff in Listing 5: tag adds a timestamp to the tuple before and after the operator execution and sniff checks if the latency of the tuple processing is above some threshold. Alternatively, a user could use the assert convenience method. For example, assume she wants to identify straggler tuples when splitting lines into words (flatMap operator). Listing 12 illustrates how she could use the assert convenience method to achieve this.

```
val r = new Reader("debugging.log")
val ds = r.get_dataset(1)
val dse = ds.assert(t => {
  time = put_in_array(t.get_tag("timestamp"))
  return (time[1] - time[0] > THRESHOLD)}
dse.collect().foreach(println)
```

Listing 12: Profiling.

6 A TAGSNIFF INSTANTIATION

We now discuss how we put all together into Snoopy, a Spark-based debugging system for monitoring and debugging Spark jobs. Note that, although we designed the system for Spark, one could easily adapt it for another data processing platform. This is thanks to its implementation on top of Rheem [5–7] rather than on top of Spark directly. Rheem translates the TagSniff primitives to Spark jobs, but it could also produce code for any other underlying data processing platform (e. g., a Flink job).

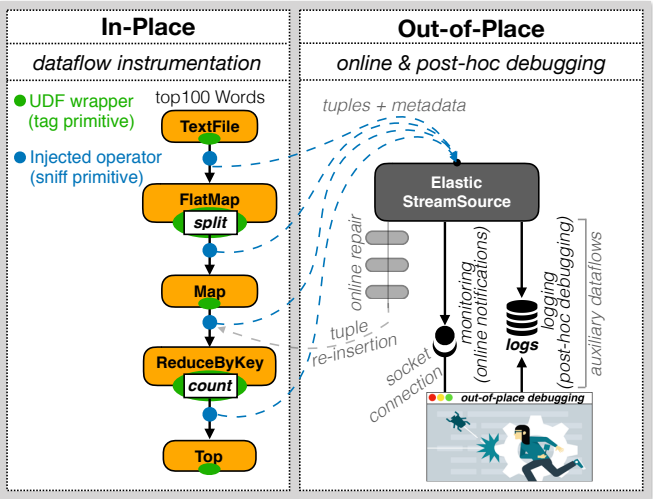


Figure 1: Snoopy internals.

```
val ln = new RDDbug(spark.textFile(file))
  .setTag(t => if (line_number(t) % 10000 == 0)
       t.add_tag("pause"))
val tw = ln.flatMap(l => l.split(" "))
val wc = tw.map(word => (word, 1))
  .setSniff(t => return t.has_tag("pause"))
val wct = wc.reduceByKey(_ + _)
  .setTag(t => t.add_tag("now-"+ System.currentTimeMillis()), PRE)
  .setTag(t => {t_start = put_in_array(t.get_tag("now"))
     if (System.currentTimeMillis() - t_start > 60000)
       t.add_tag("alert")}, POST)
  .setSniff(t => return t.has_tag("alert"))
val top100 = wct.top(100)
```

Listing 13: Instrumented Top100Words job.

Snoopy implements the tag and sniff primitives as well as the convenience methods we proposed in the previous sections. It supports both *in-place* and *out-of-place* debugging, as illustrated in Figure 1. The in-place debugging is essentially dataflow instrumentation. The out-of-place debugging is the analysis of the data coming from the instrumentation and takes place in a separate set of nodes that is not part of the cluster running the main dataflow. Snoopy comes with a GUI whereby users can monitor their jobs and interact with the system.

To better illustrate the functionalities of Snoopy and the user interaction, in the following, we consider the example of finding the top-100 frequent words (Listing 1) and assume three example scenarios: the user wants to (1) pause the job execution every 10k input lines and inspect the intermediate results for the next two operators (*data breakpoint*); (2) get an alert whenever a word takes more than 60 seconds in the reduceByKey operator (*alert*); and (3) catch any input line or word that makes the application crash (*crash culprit*).

At the core of Snoopy API is the RDDbug[2], which wraps the Spark RDD to provide additional methods for the TagSniff primitives and convenience methods. This enables users to easily instrument their jobs. In contrast to BigDebug [12] that instruments

[2]pronounced as *"rd-debug"*.

Spark jobs between stages, Snoopy allows for instrumentation, and hence debugging, at the *operator* level. This allows more flexibility in debugging at varying granularities, such as individual tuples, RDD or a node. Also, in contrast to Inspector Gadget [18], our instrumentation is lightweight and the user does not have to learn a new API. Listing 13 shows how she has to instrument the top-100 frequent words job to enable the three above debugging scenarios. We use this as our running example to illustrate the system functionality as well as the user interaction.

6.1 Tagging Tuples

To be able to annotate tuples, Snoopy adds a wrapper on each UDF in the dataflow, e. g., split and count in the WordCount job of Figure 1. The wrapper is mainly responsible for attaching system-defined and user-defined tags to every single tuple before (*pre-tag*) and after (*post-tag*) the tuple is processed by a Spark operator. The user can optionally specify when to apply her tag function by passing a PRE or POST constant as parameter in the setTag method. Snoopy annotates a tuple using the UDF provided in the setTag method of the RDDbug. By default, it also annotates a tuple with a unique identifier and the operator identifier that transforms the tuple. Note that Snoopy differs from Titian [14] in that it does not modify the RDD itself but it simply adds the appropriate identifiers in the tuple metadata, which is then processed out-of-place.

User interaction. Let's now see how the user should tag the tuples in her job to enable the debugging example scenarios outlined above. First, she has to invoke the setTag method on the RDDbug to identify the input lines satisfying the pausing condition, i. e., every 10k lines, and add the "pause" tag to these tuples (Line 2 in Listing 13). For the alert scenario, she has to invoke the setTag method on the reduceByKey operator and annotate a tuple with a timestamp before the tuple is processed by the operator (via the PRE constant in Line 7). She then annotates the tuple once it has been processed by the operator (via the POST constant): She inputs a UDF that parses the start timestamp of the tuple and adds the tag "alert" if it took more than 60 seconds to be processed (Lines 8–10). For the crash culprit scenario, she does not have to instrument the dataflow. Snoopy handles crash culprits behind the scenes without user intervention: it catches any runtime exception in the job and invokes the setTag method in the operator where the crash occurred to insert the tag "crash" as well as the exception trace, such as in Listing 4.

6.2 Sniffing Tuples

Snoopy reacts to tuple annotations by inspecting every single tuple that flows between two Spark operators and identifies those tuples requiring an action. It does so by injecting a *sniffer* operator (which is a flatMap operator in Spark) between each pair of operators in the dataflow. This sniffer operator identifies tuples of interest by applying the UDF function of the setSniff method. If it outputs true, Snoopy needs to perform a given action. It can perform three actions: (i) send a copy of the tuple out of the main dataflow for out-of-place debugging (*send-out* action); (ii) remove the tuple from the main dataflow (*skip-tuple* action); and/or (iii) pause the dataflow execution (*job-halt* action). Table 3 illustrates how pre-defined tags map to these actions. Users can add their own tags and map them to these actions via a configuration file. In this way, Snoopy is extensible to ad-hoc debugging analysis that users may wish to perform.

Table 3: Pre-defined tag-based actions.

Action	Tuple tags
send-out	alert, breakpoint, crash, display, fix, log, profile, trace
skip-tuple	crash, skip, fix
job-halt	pause

If the action that Snoopy has to perform is send-out, it clones the entire tuple together with its metadata and sends the copy for out-of-place debugging. It then clears the tuple metadata and sends the tuple to the next downstream operator: it keeps only the tuple identifier and the trace tags (if they exist). Clearing the metadata is crucial for keeping the memory overhead low. In case the tuple requires a skip-tuple action, Snoopy does not put it back into the dataflow.

If a job-halt action is required, Snoopy pauses the dataflow execution by simply holding the tuple[3]. As Spark processes tuples in a pull-model fashion, holding a tuple causes the entire dataflow to pause in that particular Spark worker: the sniffer does not request for a new tuple to the upstream operator; the downstream operators keep waiting for the next tuple to arrive. Additionally, if the job-action is at the RDD level, the sniffer requests all the other sniffers located at the same position in the dataflow (level-mate sniffers, for short), but running on different Spark workers, to pause too. In case a level-mate sniffer is not active anymore, Snoopy forwards the request to the first active sniffer in the downstream operators. Then, all sniffers resume the execution, either after a timeout or by user instruction, by sending the held tuple to the next downstream operator. Snoopy can resume the job by receiving the user's instruction via the next_tuple or the next_operator convenience methods (see Section 4.1). When receiving next_tuple, Snoopy removes the "pause" tag from the current pausing tuple and sends it to the next downstream operator. This causes the job execution to resume and pause again whenever another pausing tuple is found. In the case of receiving next_operator, Snoopy simply sends the pausing tuple to the next downstream operator. In contrast to next_tuple, this causes the job to resume and pause when the next operator (reduceByKey in our example) finishes processing the pausing tuple.

User interaction. Let us discuss now how the user has to instrument her job to enable Snoopy to sniff tuples. For the data breakpoint scenario, she has to inject her setSniff method two operators after her setTag method (Line 5) so that she can inspect the intermediate results in this part of the dataflow. Snoopy, thus, pauses and sends the intermediate results to the GUI whereby she can resume the job by invoking either the next_tuple or the next_operator convenience methods (see Section 4.1). For the alert scenario, she invokes the setSniff method right after its corresponding setTag methods (Lines 7–10). She inputs a UDF to send-out every tuple with an "alert" tag for displaying it to the user. Finally,

[3]We discuss how it achieves a simulated pause in Section 6.3.

for the crash culprit, the user does not need to set any sniff function. It is Snoopy that inserts a sniff function (as in Listing 4) in the operator where the crash occurred. This, makes the system to send-out the tuple for out-of-place debugging and remove the tuple from the main dataflow.

6.3 Debugging Tuples

It is the out-of-place debugger that is responsible for analyzing the (meta-)data produced by the job instrumentation, i. e., the tuples with their tags (tuples, for short). The out-of-place debugger consists mainly of two main parts: a data stream source (*ElasticStreamSource*) and a set of *auxiliary dataflows* to further process the tuples coming from the job instrumentation. The right-side of Figure 1 illustrates these two parts.

The ElasticStreamSource receives tuples with tags that map to the send-out action (see Table 3) and dispatches them to the user for further analysis. In detail, it sends them to the GUI, for manual user inspection, and to the subscribed auxiliary dataflows, for automatic debugging, such as value imputation. The GUI and auxiliary dataflows get tuples of interest in a publish-subscribe fashion but with a pull-model for consuming the tuples. Internally, this component uses a double buffering mechanism to temporarily store tuples until the GUI and auxiliary dataflows consume them.

An auxiliary dataflow is a regular Spark job that is executed apart from the main dataflow. The auxiliary dataflow consumes tuples annotated with a tag of interest and perform a user-defined analysis with them. Snoopy comes with two default auxiliary dataflows: the (i) *monitoring* and (ii) *logging* dataflows.

(i) The monitoring dataflow allows the GUI to consume tuples from the ElasticStreamSource component. It provides the GUI with online notifications of the user's dataflow instrumentation, e. g., alerts and crash traces. In detail, the monitoring dataflow consumes all tuples with the alert, breakpoint, crash, display, and trace tags. In the GUI, the user can then visualize, log, modify and re-inject tuples back to the main dataflow as well as pause and resume the job execution. For instance, assume an incoming tuple tagged as "display". The monitoring dataflow would pull such a tuple and immediately display it to the user in the GUI. Besides, this dataflow is in charge of providing *simulated* breakpoints, such as in [12]. A simulated breakpoint virtually pauses the job execution. For this, the monitoring dataflow checks if the tuple has the "breakpoint" tag before sending it to the GUI. If so, it buffers the tuple together with the following incoming tuples. Once the job is resumed by the user, it sends all the buffered tuples until the next breakpoint to the GUI.

(ii) The logging dataflow is responsible for storing tuples having the "log" tag for post-hoc debugging purposes (see Section 5). By default, Snoopy stores the logs in HDFS. Investigating more sophisticated methods for storing logs, such as in [14], is out of the scope of this paper.

User interaction. The user can debug tuples either by using the GUI of Snoopy or plugging an auxiliary dataflow to manipulate tuples containing a set of tags she inserted during instrumentation. In the latter case, she can use the convenience methods for post-hoc debugging and provide a job as shown in the examples of Section 5. For simplicity, we assume the user utilizes the GUI when dealing with our three debugging example scenarios. For the data breakpoint scenario, the user receives a message notifying that the system has been paused: she can then analyze the intermediate results produced so far and resume the job by invoking the next_tuple or next_operator method. For the alert and crash culprit scenarios, the user simply receives a warning in the GUI with the tuple that caused such an alert or crash.

7 EVALUATION

The goal of our evaluation is to determine the performance efficiency of Snoopy. In more detail, we carried out our experiments to answer three main questions:

(1) *Performance efficiency*: how obtrusive is Snoopy to the performance of an original dataflow? Although users may be used to high overhead in performance when debugging their dataflow jobs, we argue that it is crucial that a debugging system incurs low overhead. This not only increases productivity, but also makes it possible to debug dataflows in production.

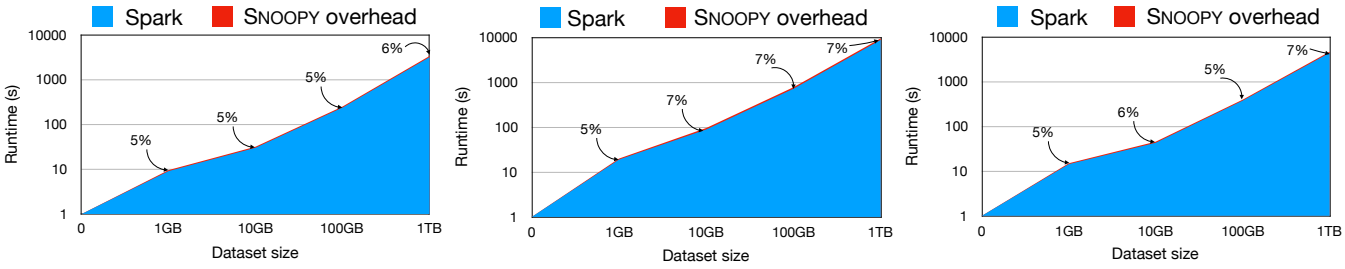
(2) *Scalability*: how well does Snoopy scale to large dataflows and compute nodes? It is crucial that a debugging system scales with the size of the dataflow and number of nodes. That is, increasing the number of operators in a dataflow and of compute nodes in a cluster should not increase the overhead of the system.

(3) *Responsiveness*: how responsive Snoopy is to system events and user instructions? We believe that allowing interactivity with users is not a nice-to-have feature but a must. Users must be able to guide the system in their debugging tasks in a real time fashion, i. e., users should not wait more than a second to see the result (for the system to react) [22].

Setup. We ran all our experiments on a cluster of 10 machines, each with: 2 GHz quad-core CPU, 32GB memory, 1TB storage, 1 Gigabit network, and 64-bit Ubuntu OS. Assuming that the 10-machines cluster is a production cluster, we used a Mac Pro for out-of-place debugging: 2.7 GHz 12-core CPU, 64GB memory, 1TB storage, 1 Gigabit network, and macOS Mojave. We used Java 9, Spark 2.4.0, and HDFS 2.6.5. We used each of these platforms with their default settings and 20 GB of max memory. We only set Spark to run one worker per core, which results into a cluster size of 40 workers for the production cluster and 12 workers for the debugging node. We use Spark as the baseline and consider three different tasks: grep, wordcount, and join (PigMix L2). We use the Wikipedia-abstracts dataset for grep, wordcount and PigMix dataset for join and vary the dataset sizes from 1GB to 1TB. We run each of our experiments 5 times, removed the slowest and fastest results, and report the average time of the remaining 3 executions.

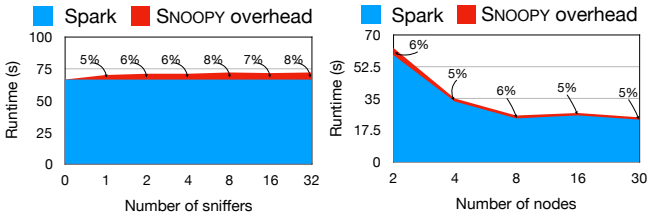
7.1 Performance Efficiency

We first evaluate the overhead that Snoopy incurs on Spark jobs. For this experiment, we considered all three tasks (grep, wordcount, and join) with varying input dataset sizes. We instrumented Snoopy to annotate tuples in order to support all tasks of Table 1. Note that this represents the worst case for Snoopy in terms of performance, because it has to use all tags to annotate a tuple (including tuple



(a) Grep (b) WordCount (c) Join

Figure 2: Overhead when debugging Spark jobs with SNOOPY.



(a) Dataflow size (b) Cluster size

Figure 3: SNOOPY scalability.

and operator identifiers). We measure the runtime of a task running on Spark without any debugging (*Spark*) and the runtime of the task with debugging. We then report the difference (*overhead*).

Figure 2 illustrates the results of this set of experiments. Overall, we observe that SNOOPY incurs a very low overhead of only 5-7% regardless of the task or dataset size. This is because the out-of-place debugger absorbs most of the overhead. Recall that the actual debugging, e. g., fixing or formatting malformed tuples, is done apart from the cluster running the main dataflow. Thus, the overhead depends mostly on the socket connections that the out-of-place debugger receives and hence on the number of sniffers. The number of sniffers in these three tasks is small: it ranges from 2 to 7. This is why SNOOPY has a similar incurred overhead for all three tasks. In the following, we shall show the scalability of SNOOPY in terms of number of sniffers. These results show the high performance efficiency of SNOOPY, which leads to retaining the Spark execution time almost intact. This means one can use SNOOPY in production. In fact, it incurs 10x less overhead than BigDebug [12] for grep and wordcount.

7.2 Scalability

We now evaluate the scalability of SNOOPY in terms of dataflow and cluster size.

Increasing dataflow size. For this experiment, we composed synthetic Spark jobs with varying number of operators, from 2 to 33, and instrumented them with one sniffer per operator pair connection. We considered the Wikipedia-abstracts dataset with a size of 10 GB. Figure 3(a) shows the results. As we noted earlier, the overhead depends on the number of sniffers instrumented in the dataflow. We observe how the overhead incurred by SNOOPY increases slightly as the number of sniffers increases. This is because the out-of-the place debugger has to deal with more connected sniffers. Still, we observe that SNOOPY scales gracefully to large dataflows. For instance, for a dataflow of 33 operators (i. e., 32 sniffers), which is already a relatively large dataflow in practice, the overhead is only 8%.

Increasing number of compute nodes. The goal of this experiment is to determine if SNOOPY allows Spark to retain its node scalability. For this experiment, we used the grep task with an input dataset size of 10 GB and varied the number of compute nodes. Figure 3(b) illustrates the results. We observe that SNOOPY incurs an almost constant overhead: it ranges from 5% to 6%. This minor difference is mainly due to the cluster variance.

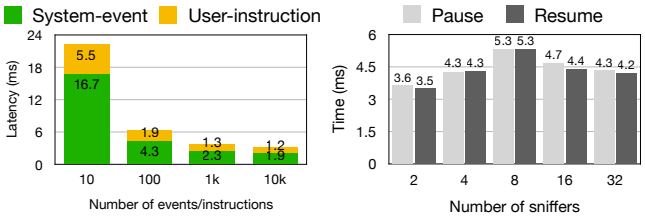
Therefore, the above results allow us to conclude that SNOOPY scales gracefully with the dataflow size and along with Spark for increasingly larger clusters.

7.3 Responsiveness

We end our evaluation with a set of experiments to evaluate how responsive SNOOPY is to system events (e. g., alerts and crashes) as well as user instructions (e. g., system pauses and tuple re-injections). For this, we consider two debugging scenarios for the join task on 100GB: *crash culprit* and *online debugging*.

Crash culprit. Recall a crash culprit is a tuple that causes the dataflow to fail. We constructed this scenario by inserting a varying number of tuples with null values into the input dataset, which causes the join task to fail. SNOOPY catches such exceptions and sends them to the out-of-place debugger for user inspection (*system-event*). The user inspects and fixes the failing tuple (*tuple-repair*). Then, she re-inserts the fixed tuple into the main dataflow (*user-instruction*). We report the average times only for the system-event and user-instruction, because the tuple-repair time depends on several external factors.

Figure 4(a) shows the results of this experiment. The x-axis shows the total number of system events and, thus, user interactions (i. e., the number of crash culprits). We observe that SNOOPY ensures an immediate response to system events as well as to user instructions. From the moment a crash occurs, in the worst case (when having 10 failing tuples), it will take (i) 16.7 ms to send the crash culprit (the failing tuple with its metadata) to the user for



(a) Interaction time (b) Time to pause or resume

Figure 4: Snoopy **responsiveness.**

out-of-place debugging and (ii) 5.5 ms to receive back the fixed tuple. Surprisingly, we see that the responsiveness of Snoopy increases with the number of events and instructions. This is because the TCP/IP protocol buffers up data until there is a decent amount of data to send. Thus, when having more events/instructions, this waiting time goes down significantly: it takes 3.1 ms for sending $10k$ crash culprits and receiving back the fixed tuples.

Online debugging. Here, we assume the user decides to pause the system to inspect how tuples are being transformed by the dataflow (via the next_tuple and next_operator convenience methods). For this, Snoopy must "instantly" react so that she can inspect a particular observed behavior. We measured the time Snoopy takes to pause the entire dataflow from the moment the user instructs it and the time to resume the computation. We varied the number of sniffers instrumented in the dataflow. Figure 4(b) shows these results. We observe that Snoopy takes $4-5$ms on average to either pause or resume the entire dataflow, regardless of the number of sniffers. This is because they ping to each other directly without any server among them (and hence no possible bottleneck).

These results show the high responsiveness of our system, which makes it a truly interactive debugging system.

8 RELATED WORK

The popularity of dataflow based systems such as Hadoop and Spark has created the need for more advanced big data debugging tools. Almost all of the prior research work has focused on simplifying a specific debugging scenario. In contrast, TagSniff provides a holistic approach based on powerful primitives that could be used to implement almost all of the common debugging tasks. Additionally, Snoopy supports both online and post-hoc debugging.

The closest to our work is Inspector Gadget (IG) [18] which provides a programming framework for coding any monitoring or debugging task. IG provides a set of Java APIs that a user needs to implement for enabling a given debug task. Although we share many of IG's goals, we have some core differences: (i) our primitives require less coding effort from the users, (ii) our architecture employs an out-of-core debugging mechanism which significantly decreases the overhead, and (iii) we support any post-hoc analysis while IG supports only forward tracing. BigDebug [12] provides support for simulated watchpoints and breakpoints for debugging Spark jobs, but requires the modification of the underlying dataflow engine. The primitives proposed in BigDebug are more coarse-grained and do not support post-hoc solutions. Arthur [10] supports selective replay based post-hoc debugging scenarios but not online debugging. Graft [20] is a post-hoc debugger for Apache Giraph [2] where users can replay their graph dataflows. Daphne [15] is a visualization and debug tool for DryadLINQ jobs. It is based on an abstraction of different sources of information about a job (e. g., job plan, runtime logs, application logs, input/output). While debugging, a user has to select the vertex that caused the failure and re-execute it via the Dryad runtime. BigShift [11] automates a specific task by combining data provenance and delta debugging [21]: given a test function, the system automatically finds a minimum set of fault-inducing input data responsible for a faulty output. However, this technique is used for debugging anomalous results and requires as input a user-defined test function.

There are also many prior works that seek to support provenance in dataflow engines as it could be used for many debugging tasks, such as forward/backward tracing. Some works [9, 24] are based on offline provenance and require to re-execute the job to record any required information. Although this approach occurs no extra overhead on the main dataflow, it requires much time to capture the provenance, especially for long running jobs. On the contrary, we instrument the job and gather the information needed on the fly with a minimum overhead. Similarly to our approach, RAMP [13] and Newt [16] use wrapper functions to capture tuple-level lineage on dataflow engines. RAMP records provenance for forward and backward tracing in MapReduce workflows but can incur an overhead of up to 76%. Newt records the arrival and departure of each tuple to an operator and can reconstruct the lineage offline. It incurs a lower overhead of $12-49\%$, but does not support debugging itself: one has to query the provenance stored in the system and implement her debugging tasks. Titian [14] extends Spark's RDD to capture lineage information and stores it using Spark's internal structures. However, this requires a modified version of Spark, which may not always be feasible, e. g., in large companies that require SLAs.

9 CONCLUSION

We made two major contributions that could dramatically simplify big data debugging. First, we introduced the TagSniff model, which is based on two powerful primitives. We demonstrated via numerous examples how they could concisely specify almost all of the common debugging scenarios. Second, we described Snoopy, an efficient implementation of TagSniff built on top of Spark. Snoopy makes a number of innovative architectural choices that allows it to support both online and post-hoc debugging with an average overhead of 6%. Furthermore, it does not need any modification to the underlying system thereby making it portable. We believe that both TagSniff and Snoopy have the potential to reduce the tedious and time consuming task of big data debugging.

ACKNOWLEDGMENTS

This work has been supported by the German Ministry for Education and Research as BBDC 2 (ref. 01IS18025A) and BZML (ref. 01IS18037A).

REFERENCES

[1] Apache Flink. https://flink.apache.org/.
[2] Apache Giraph. https://giraph.apache.org/.
[3] Apache Hadoop. https://hadoop.apache.org/.
[4] Apache Spark. https://spark.apache.org/.
[5] D. Agrawal, M. L. Ba, L. Berti-Équille, S. Chawla, A. K. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki. Rheem: Enabling Multi-Platform Task Execution. In *SIGMOD*, pages 2069–2072, 2016.
[6] D. Agrawal, S. Chawla, A. Elmagarmid, Z. Kaoudi, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and M. J. Zaki. Road to freedom in big data analytics. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 479–484, 2016.
[7] D. Agrawal, B. Corteras-Rojas, S. Chawla, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and A. Troudi. Rheem: Enabling Cross-Platform Data Processing – May The Big Data Be With You! *PVLDB*, 11(11):1414–1427, 2018.
[8] T. Britton, L. Jeng, G. Carver, and P. Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". 2013.
[9] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. *The VLDB Journal*, 12(1):41–58, 2003.
[10] A. Dave, M. Zaharia, S. Shenker, and I. Stoica. Arthur: Rich Post-Facto Debugging for Production Analytics Applications. Technical report, University of California, Berkeley, 2013.
[11] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. Automated Debugging in Data-intensive Scalable Computing. In *SoCC*, pages 520–534, 2017.
[12] 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 *ICSE*, pages 784–795, 2016.
[13] R. Ikeda, H. Park, and J. Widom. Provenance for Generalized Map and Reduce Workflows. In *CIDR*, 2011.
[14] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data Provenance Support in Spark. *PVLDB*, 9(3):216–227, Nov. 2015.
[15] V. Jagannath, Z. Yin, and M. Budiu. Monitoring and Debugging DryadLINQ Applications with Daphne. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1266–1273, 2011.
[16] D. Logothetis, S. De, and K. Yocum. Scalable Lineage Capture for Debugging DISC Analytics. In *SOCC*, pages 17:1–17:15, 2013.
[17] D. H. O'Dell. The Debugging Mindset. *ACM Queue*, 15(1):50:71–50:90, 2017.
[18] C. Olston and B. Reed. Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows. *PVLDB*, 4(12):1237–1248, 2011.
[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.
[20] S. Salihoglu, J. Shin, V. Khanna, B. Q. Truong, and J. Widom. Graft: A Debugging Tool For Apache Giraph. In *SIGMOD*, pages 1403–1408, 2015.
[21] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
[22] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. How progressive visualizations affect exploratory analysis. *IEEE Trans. Vis. Comput. Graph.*, 23(8):1977–1987, 2017.
[23] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin. An empirical study on quality issues of production big data platform. In *ICSE*, pages 17–26, 2015.
[24] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *SOSP*, pages 295–310, 2011.