HAL
open science

Practical Online Debugging of Spark-like applications

Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix

To cite this version:

Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix. Practical Online Debugging of Spark-like applications. IEEE QRS 2021 : International Conference on Software Security and Reliability, Dec 2021, Hainan Island, China. hal-03398772

HAL Id: hal-03398772

https://hal.science/hal-03398772v1

Submitted on 23 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Practical Online Debugging of Spark-like applications

Matteo Marra
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
mmarra@vub.be

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
guillermo.polito@univ-lille.fr

Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

***Abstract*—Apache Spark is a framework widely used for writing Big Data analytics applications that offers a scalable and fault-tolerant model based on rescheduling failing tasks on other nodes. While this is well-suited for hardware and infrastructure errors, it is not for application errors as they will reappear in the rescheduled tasks. As a result, applications are killed, losing all the progress and forcing developers to restart them from scratch. Despite the popularity of such a failure-recovery model, understanding and debugging Spark-like applications remain challenging. When an error occurs, developers need to analyze huge log files or undergo time-consuming replays to find the bug. To address these concerns, we present an *online debugging* approach tailored to Big Data analytics applications. Our approach includes local debugging of remote parallel exceptions through dynamic local checkpoints, extended with domain-specific debugging operations and live code updating functionality. To deal with data-cleaning errors, we extend our model to easily allow developers to automatically ignore exceptions that happen at runtime. We validate our solution through performance benchmarks that show how our debugging approach is comparable or better than state-of-the-art debugging solutions for Big Data. Furthermore, we conduct a user study to compare our approach with another state-of-the-art debugging approach, and results show a lower time to find the solution to a bug using our approach, as well as a generally good perception of the features of the debugger.**

I. Introduction

Big Data frameworks such as Hadoop Map/Reduce and Apache Spark (Spark in short) are extremely popular and widely used in industry for writing data analytics applications due to their ability to scale to large amounts of data and their resilience. They provide a fairly simple programming model which exploits both the convenience of function calls to mask their parallelization and fault-tolerance mechanisms aimed at the not always reliable execution on a cluster. When a program fails during a parallel execution, the framework re-schedules the failed computation in another node. Applications then either terminate without errors or, if the failure was caused by a bug in the code, they stop after a number of re-scheduling attempts. While such a failure-recover model gracefully deals with hardware and infrastructure errors, it is not suitable for application errors as they reoccur even when they are re-scheduled. As such, applications are eventually stopped when reaching the rescheduling limit, forcing developers to redeploy them, i.e. restarting them from scratch in the cluster.

Whenever errors occur, log files are generated and progress results are discarded. Unfortunately, extracting enough information from these logs to understand production failures is known to be difficult [1]. Alternatively, a recorded (or partially recorded) execution could be replayed using replay and check-point-based debuggers [2], [3]. Replaying requires long debugging times especially when data analytics applications run for many hours or even days. Checkpoint-based debuggers can potentially alleviate that issue. But current solutions for Big Data frameworks, like BigDebug [3] for Spark, rely on developers placing the right checkpoints beforehand, which does not guarantee to avoid a total replay. In fact, developers usually avoid non-necessary checkpoints due to their performance costs. More recently, data provenance [4] has been explored to backtrack exceptions during the execution. This approach still requires a post-mortem analysis of the execution, backtracking, and at least partial replays. Full online debugging support as found in mainstream IDEs remains unexplored for Big Data frameworks.

In this work, we propose the use of an always-active debugging infrastructure to interactively debug Spark-like [1] applications. We argue that Big Data frameworks should incorporate debugging support which (1) reduces replay times upon an error to the minimum, (2) provides all the contextual information necessary to discover the bug, (3) avoids full redeployment of applications for application errors when possible, (4) and gives the possibility of automatically ignoring errors related to data that would anyways be discarded. This paper presents a practical online approach specially designed to offer such debugging support. Our approach avoids replays by transferring the context of a failing execution to a debugger as soon as a failure occurs. The debugger offers full debugging capabilities on a remote exception similar to what mainstream IDEs offer for regular applications including the possibility of inspecting all variables, stepping into/over the execution of expressions, and augmented with advanced stepping operations targeted to the Spark-like model (such as

Identify applicable funding agency here. If none, delete this.

[1] We define *Spark-like* applications those written with a programming based on a distributed data structure akin to Spark's RDD [5].

step to next transformation, step to next element, and step to action result). To ease the development process, our approach features live code updating capabilities that avoid redeploying the whole application upon any code change, as well as a built-in ignoring functionality to automatically ignore failures in certain parts of the program.

We validate our approach with both performance benchmarks and a user study. On the one hand, we show that our debugging approach is practical by measuring its overhead on the execution when multiple failures happen in parallel; on the other hand, we show through a within-participant strong experimental user study with 17 participants that the time to find a bug was on average lower using our debugger, and our live code updating capabilities allowed participants to minimize application re-deployments. Overall, the participants rated positively to very positively Spa's debugging features when compared to a debugger similar to Big Debug.

The key contributions of this paper are:

- An online debugging model for Spark-like applications based on *dynamic local checkpoints*, that enables local debugging of remote parallel exceptions and incorporates domain-specific debugging features such as partition windowing and composite exceptions.
- Built-in support to *automatically ignore exceptions* caused by records that should be removed anyways before an analysis through a data cleaning phase.
- An implementation of our approach in Pharo Smalltalk, including debugging through dynamic local checkpoints and live code-updating of applications, that we use for both our performance benchmark and our user study.
- We validate the overhead of our approach through performance benchmarks, and the usability of our debugger and its features through a user study.

II. Background and Motivation

This section introduces the context and motivation of our work. We first discuss Spa, the system in which we built our online debugging model and ignore support. Spa is a Big Data framework for Pharo Smalltalk, that features a programming and computational model akin to Spark [6], i.e. Spark-like. Like Spark, Spa is based on the Master/Worker model [7] in which a number of workers nodes execute operations on data partitions, or simply *partitions*, and a master node assigns the execution of operations (*i.e.* tasks) to the workers and coordinates their results. In what follows, we use Spa to introduce the key concepts of our Spark-like model.

A. Big Data Programming with Distributed Data Structures

Spa's programming model is based on the concept of a distributed data structure (DDD), akin to Spark's RDD [5]. Using DDDs, applications are expressed in terms of functional transformations on data structures that are eventually executed in parallel by the infrastructure. Developers create a DDD by distributing a data source such as a collection or a file and apply operations on it. DDD operations are divided into two groups: *transformations* and *actions*.

In detail, *transformations* do not alter the structure of the data, but just (potentially) its contents and are executed in a lazy fashion. Common transformations include `map:`, `filter:`, and `flatMap:`. Each transformation returns a new DDD: if more than one transformation is called in sequence, all these transformations are pipelined. *Actions* are operations that alter the structure and optionally the contents of the data structure and are executed eagerly, immediately triggering the execution of the operation pipeline. Common actions include `reduce:`, `reduceByKey:`, `groupByKey:`, `aggregate:with:`, `sorted:` and `count:` operations. The code snippet below illustrates how these are put together to make up a classical word count application in Spa using two transformations and one action. [2]

```
fileDDD := spa distributeFromFile: '/path/to/file/or/dir'
flatDDD := fileDDD flatMap: [:line | (line substrings: ' ') ].
pairDDD := flatDDD map: [ :word | word -> 1 ].
result := pairDDD reduceByKey: [:v1 : v2 | v1 + v2 ].
```

B. Persistence

By default, actions do not store intermediate results: it is up to developers to decide when to persist data in the execution pipeline. The `execute` action stores in a new DDD the contents of the current DDD after applying all of the transformations in the pipeline, making data available across subsequent operations.

Persisting data in a Spark-like model implies high performance costs as, besides its memory footprint, it requires coordination between the workers and the master. This forces developers to carefully choose at what point in the pipeline data is stored so that the overhead of persisting is lower than the overhead of redoing that part of the pipeline.

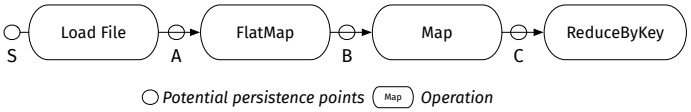

Fig. 1. The pipeline of a wordcount and its potential persistence points.

To explain this trade-off, consider again the wordcount application explained above. Figure 1 shows the execution pipeline of the application. The circles represent the different points in which a developer may introduce a persistence point: S (start), A, B, and C. Persisting at point A (after loading the file), will store the content of the file in memory, so developers usually insert it if those contents are used more than once. Persisting at point B would break the pipelining of the flatMap and the map, requiring a synchronization in the middle, so developers rarely insert it. Persisting at point C will store the filtered and mapped data, so developers will insert it if that data is used again in some other computation after the `reduceByKey` operation.

[2] All code snippets in the article are written in Pharo Smalltalk. In Pharo Smalltalk syntax, method invocations are expressed with keywords, block closures are delimited by square brackets and dots ("`.`") are used as statement separators. "`spa distribute: data`" is equivalent to "`spa.distribute(data)`" in canonical syntax.

C. Debugging Support

Let us consider an exception during a parallel execution causing the application to terminate with an error. Upon the error, the developer gets a log including a stack-trace and information about the exception (i.e. its type). Unfortunately, debugging using logs is very difficult because they miss important *contextual information* relevant to determine the root cause of the bug [1]. The contextual information is typically limited to what developers anticipated beforehand, usually a stack-trace, and the values of particular variables that they printed to the standard output.

Recently, several offline debuggers have been proposed for Big Data frameworks [2], [3], [29]. A replay debugger such as Arthur [2] allows developers to replay a previously recorded execution. In our example, it will need to replay from S to the point where the exception is raised. Due to the amount of data that is analyzed, these replay times can be very long. Alternatively, a checkpoint-based debugger such as BigDebug [3] exploits the points at which an application persists data to make a checkpoint. In particular, BigDebug will replay from the latest persisted point that is closest to a breakpoint, potentially reducing the replay time. However, since developers avoid checkpointing due to their performance costs, there is no guarantee that a checkpoint was placed when a failure actually appears. This leads to at least a partial replay of the application to find the place where to checkpoint, with a full replay in the worse case when no checkpoints are available. Concerning contextual information, those debuggers do not offer much more information than log-based solutions. For instance, BigDebug only exposes the value/record passed as a parameter to the current transformation, but it is not possible to check the values of other variables nor to evaluate code in the context of the debugged execution.

We should also consider that not all errors are harmful: there are situations where ignoring certain application errors has little or no impact on the final result of the application. For instance, a recent study has shown that developers spend hours debugging data-cleaning errors [8], finding that a handful of fail-inducing data records are often traduced in hours of lost computations. In Hadoop Map/Reduce a parameter can be set in the configuration files to disregard a certain number of failed tasks. However, this is not easily adaptable since requires a full restart of the framework when changed, and is very corse-grained as it applies to *all* tasks for all applications. While it is possible to ignore exceptions manually, this involves adding boilerplate code to correctly handle and return the ignored records.

Besides finding the root cause of a bug, a crucial aspect of the debugging cycle is to update the program with a corrected version. The most common approach is to recompile the application to redeploy it in the cluster. In Spark, redeployment to a cluster implies re-packaging the application, uploading it to the cluster, and re-submitting it to the framework instance. Since this process is time-costly, debuggers such as BigDebug allow updating the code of an application, but in a limited way: in BigDebug developers may update only the code applied by a transformation. Broader changes to the code such as modification of the pipeline or application methods and classes need a standard redeployment.

D. Problem Statement

The existing debugging support for Spark-like applications discussed above suffers from some issues that motivate our work:

1) They rely either on replay or on memory and costly checkpointing. In contrast, an always-on online debugger could enable immediate access to the execution and interactive debugging through stepping operations.
2) They offer very limited contextual information on a breakpoint or failed execution. For instance, they lack the possibility to inspect all the variables in the context of the execution, as well as the ability to evaluate expressions in the same context.
3) They do not support fine-grained automatic ignoring of exceptions to neutralize data-cleaning errors customizable per application or task.
4) They offer very limited support for updating an application without fully re-deploying it.

III. Online debugging of Big Data applications

In this paper, we explore always-active debugging support specially designed to deal with the characteristics of Spark-like applications. We propose a novel debugging approach that combines traditional online debugging with *dynamic local checkpoints*, i.e. checkpoints which are created dynamically when an exception happens. The checkpoint captures all the contextual information about the exception and propagates it to the developer's machine, opening a debugging session recreating the runtime environment of the bug. The benefits of this approach are two-fold. First, dynamic checkpointing removes from the user the burden of deciding where to checkpoint, as it is automatically created upon exception. Second, developers proceed to debug locally using a traditional online debugger that exposes all contextual information and step operations. To deal with Spark-like computation, our approach offers dedicated stepwise operations,*e.g.,* step to next transformation, incorporates live code-updating capabilities to avoid full redeployments and support to automatically ignore exceptions. In what follows we describe in more detail the key concepts of our approach.

A. Overview of our Online Debugging Architecture

Figure 2 provides illustrates of our debugging infrastructure, in yellow, on top of a Spark-like runtime in blue (Spa in our prototype implementation). In this example, it is deployed on a cluster hosting one master and two different workers nodes. The developer's machine is external to the cluster, and runs the monitoring and debugging UI in white. Our infrastructure augments the Spark-like runtime with a Debugger Monitor at the master node in charge of capturing the dynamic checkpoints and send them to a Debugger Manager at the developer's

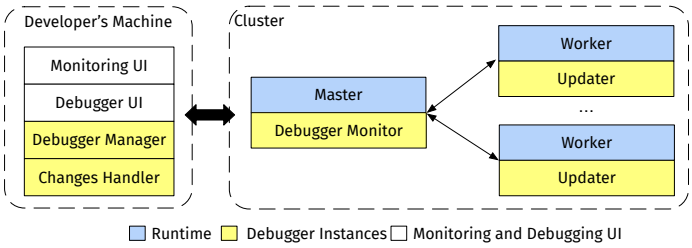

Fig. 2. Architecture of our debugging model.

machine. Our approach supports live code updating by introducing a Changes Handler within the developer's machine that packages the code changes done during a debugging session in a *patch* and sends them to the Updater in the cluster worker nodes. The Updater applies the code patches and resume the execution of the affected tasks.

Our debugging architecture is based on prior work on *out-of-place debugging* [11], [12]. In contrast to prior work, this work introduces (1) dynamic local checkpoints, (2) automatic ignoring of exceptions, and (3) domain-specific debugging operations for stepwise execution of Spark-like applications. Finally, it also revisits the original live code-updating support.

B. Practical Online Debugging with Dynamic Local Checkpoints

Inspired by out-of-place debugging, when an exception happens in the cluster, the entire execution environment (i.e. the execution stack with its associated data) is transferred to the developer's machine to enable local debugging of the remote computation. This allows developers to debug errors in isolation and avoid replays. However, in Spark-like applications, the associated data may be very large as it includes the failure-inducing record as well as the whole partition that includes it. In order to scale to large amounts of data processed by Spark-like applications, we incorporated two features: (a) partition windowing and (b) composite exceptions.

a) Partition windowing: Instead of transferring the entire execution stack, our approach (i) removes all the frames from the stack related to framework-related method calls on the worker, and (ii) trims the data of the failing partition around the failure-inducing record. For example, if the runtime was applying a `map:` over a partition of 3000 elements, and the element at index 1234 fails, the failing partition will be cut to contain only a window of N elements, with indexes $[1234-(N/2), 1234+(N/2)]$, adjusted to respect the bounds of the partition. Partition windowing allows developers to debug a failure-inducing record in context with its neighboring records without having to transfer large amounts of data. By customizing the window size (i.e N), developers make a trade-off between the debugging overhead and the contextual information accessible during debugging.

b) Composite exceptions: To further reduce network overhead, our approach adapts the concept of *composite exceptions*, originally devised for Map/Reduce [12] to DDDs. Composite exceptions aggregate many similar exceptions that occur in the same parallel execution regardless of their data partition or worker nodes. We consider two exceptions similar if they have the same type and they happened at the same point of the execution; i.e. same call stack, same program counter.

In order for composite exception to work with DDDs we employ delta stacks [13]: we extract the execution stacks of two similar exceptions and for each pair of stack frames, we compute their differences. We then serialize only one of the stacks of the two exceptions (previously shrunk with partition windowing), and for the other one only the differences. If there is more than two similar exceptions, the serialization of all other ones will include only the differences with the first one.

C. Domain-specific Debugging Operations

Crucial to our approach is that developers debug Spark-like applications by means of a *local* online debugger on a breakpoint or exception hit on a *remote* worker. Once the debugging session is opened at the developer's machine, the debugger provides classical online features like a view of the state of all the variables for each frame of the call-stack, the possibility to inspect and evaluate code in the debugged context, and operations to step into and over the execution of methods, none of which is possible in current debugging approaches such as BigDebug.

To ease the debugging of Spark-like applications, we propose three coarse-grained stepping operations designed for a Spark-like execution model which complement the stepping operations from classic online debuggers:

Step to next record. Steps to the next execution of the same transformation, i.e. to the next record.

Step to next transformation. Steps to the first execution of the next transformation (akin to BigDebug step over).

Step to action result. Steps until the point in which the next action is applied (locally). At that point, the developer inspects the result to evaluate (i) if the execution finished correctly and (ii) the result of such an action

Combined with dynamic local checkpoints and partition windowing, the developer is then able to debug the execution (i) locally without network overheads and (ii) on a subset of the original partition, and not only on the failure-inducing record, and (iii) with the desired stepping granularity.

D. Live Code Updating

In order to avoid costly redeployments of Spark-like applications, our approach captures any code change made by the developer during a debugging session to then propagate the updated code to the cluster. Code changes captured include addition, modification, and removal of any method or class. Once satisfied with the code changes she made, the developer commits them to update the application code in master and workers. This happens live, without having to forcedly re-initialize master and workers, and without re-deploying the application. Code updates are treated as an operation, i.e. executed from the worker as a task, when the worker is finished executing the current task if any to prevent concurrency issues.

Once a code change was committed, the developer decides to either (i) restart the failed partitions, (ii) restart the whole operation, or (iii) restart from a previous persisted point. The first option will restart the particular failed operation on the partitions that caused the previous error, using the updated code. The second will restart the failed operation on all of the partitions, including the ones that did not cause an error. The last will restart the execution from a previous point, re-executing all the transformations thereafter. It is always possible for the developers to restart the whole application by re-executing it on the same workers if necessary.

E. Ignoring Exceptions

Debugging a Big Data application does not necessarily entail looking at the code: this is the case for instance of errors generated by dirty data, often the reason for much time loss [8]. To tackle this problem, we propose to build in our debugging infrastructure support to automatically ignore exceptions. To this end, we build on the ideas of acceptability-oriented computing [14] and *relaxed computations* [15] which incorporate a failure model where errors are accepted automatically by the runtime. We believe that such ignoring support can help developers in avoiding tedious debugging time for errors that would be discarded anyway, as is the case for many data-cleaning errors [8].

While in Map/Reduce ignoring can happen only by means of discarding a percentage of failing tasks, in our approach developers decide to ignore exceptions globally for all actions, or locally to a particular action and its related transformations in the pipeline, and with different percentage thresholds in terms of records and not of tasks. The global ignore mode and its threshold are set through the debugger's API or from the UI, in a similar way as launching an application in *debug* mode in a classic IDE. The local ignore mode is set programmatically with a new action: `ignoreExceptions`. This action forces the ignoring of exceptions of its *preceding* pipeline and returns a new DDD with all the records that have not failed.

When an error is ignored, the debugging infrastructure filters out the failure-inducing record so that the resulting data structure does not include it (or a transformation of such record). Developers still gain access to ignored records to verify the proper execution of the application through the debugger UI.

We also propose an `ignoreExceptions:` action variant where the ignore threshold is specified as a percentage of the size entire data set. By default, calling `ignoreExceptions` without parameters is equivalent to calling `ignoreExceptions:100`. Each action (and related transformations' pipeline) has one local handler with one ignore counter associated, cumulative across the different transformations. Different actions can have different local handlers (with different thresholds). A local handler always overrides the global handler.

IV. Implementation

We have implemented the proposed online debugging approach in Spa. Our prototype is built on Pharo Smalltalk (v8) [**?**] and uses several Pharo libraries to manage the execution and debugging. Zinc [17] is used for HTTP network communication, Fuel [18] for object serialization, and Epicea [19] for detecting code changes at the developer's machine and applying code patches in the workers at the cluster. In the following, we describe some of the implementation choices made for ignoring exceptions, and for optimizing online debugging. Finally, we present the main parts of Spa's UI added to Pharo's IDE that allows developers to code, run and debug their Spark-like applications.

A. Handling the Call stack

As detailed in Section III, the size of transferred execution stacks is reduced using partition windowing. By transferring a window of records instead of only the failing record, developers can further debug using the real data surrounding the failure-inducing record. It is important to notice that the proper implementation of partition windowing requires a careful traversal of the execution stack to apply windowing to any data structure held by the stack, especially when they reference the original partition or a subset of it. Collections found to reference the original partition are cut using the same strategy as partition windowing. Streams and iterators need not only to be cut, but also the elements currently being iterated should be set as the pivot of the stream, and the stream's position, read limit, and write limit need to be adapted. While this approach drastically reduces the amount of serialized data, we are looking into using a custom communication protocol (instead of HTTP) to further optimize network overhead.

B. Debugger Front End

The front-end of Spa is integrated with the Pharo IDE. Spa's GUI allows developers to choose a global failure model (Debug, Ignore, or Ignore and Debug) and to set the ignoring threshold. It also allows developers to specify three application deployment modes: (i) local, where master and workers are deployed on the current machine; (ii) standalone, where master and workers are deployed manually and the front end connects directly to the master by its IP; (iii) on Hadoop Yarn, in which containers running master and workers are automatically deployed by Yarn on a (cluster of) machine(s). Finally, it offers an extension of the Pharo IDE Playground, called the *Spa evaluator* supporting asynchronous execution of code, so that the Pharo image remains responsive while the code is running on the cluster. Due to space restrictions, we describe and show different screenshots of the debugger' UI in appendix A.

V. Validation

To evaluate our approach we conduct both a quantitative and qualitative study. For the quantitative study, we measure the overhead and performance of our approach through performance benchmarks. For the qualitative study, we conduct a user study using a mixed-method experimental design [20]

where participants solve different debugging tasks using two different debuggers: ours and a re-implementation of BigDebug [3]. We compare both approaches and evaluate the features of our debugger.

A. Quantitative Evaluation

We conducted several performance benchmarks that aim to answer the following questions:

- RQ 1. How does our debugging approach scale to Big Data?
- RQ 2. How much time does online debugging save in comparison to replay and checkpoint-based debugging?
- RQ 3. What is the overhead of ignoring exceptions, and how does it scale to a big number of exceptions?

1) Setup: We run our experiments on a cluster composed of one *root* node and eight identical *slave* nodes. Each node presents an Intel Xeon CPU E3-1240 @ 3.50GHz, 32 GB of RAM, and 200 GB of SSD Storage. Nodes are connected via a 1 Gb/s local network.

For all the benchmarks, we deploy Spa on the cluster using Hadoop Yarn, and we use 1 single-core master, and 20 single-core workers. Hadoop Yarn takes care of the allocation of the master and workers on the cluster. The cluster runs Pharo 8.0.0 (x64) on a Pharo 8.3.0 Headless VM. We control the execution from a machine running an Intel Core i7-7567U @ 3.5GHz CPU, 16 GB of RAM, and 500 GB of SSD storage. This machine uses SSH tunneling to communicate to the cluster.

2) The benchmark suite: We run our benchmarks on three different Big Data applications, commonly used to test the performance of Big Data frameworks [21]: distributed grep, wordcount and K-means. In the distributed grep, each line of the dataset is loaded by the workers in different partitions, and only the lines containing a certain string are extracted. In the wordcount, tweets are parsed from a file, and the application counts how many times each of the words is found in the text. In K-means, tweets are parsed from a file, and, for a number of hashtags, tweets containing those single hashtags are clustered according to how many times they were liked. K-means is also a representative application to evaluate the ignoring of exceptions, as it is not heavily impacted in the results by a lower accuracy [22]. Furthermore, the dataset used for this application is naturally "dirty", so ideal for this experiment.

3) Dataset: We run all the applications against a subset of a dataset containing 100 GB of tweets (represented in JSON), coming from a recording of the Twitter live stream. The great majority of the tweets in the dataset are valid (i.e they have an id, a text, etc.) and are thus correctly parsed. However, since the dataset was recorded from a live stream, it also presents around 17% of tweets that were malformed or miss information that causes the application to fail if they are not filtered out. In our experiments, when we say that we inject failures, it means that we leave some of those tweets in the dataset which generate exceptions during the execution.

4) RQ 1: Does online debugging scale to Big Data?: This section validates whether our online debugging approach scales when having big amounts of failure-inducing records. To this end, we run the K-means application and inject failures in the execution to assess the overhead of generating and transferring a debugging session.

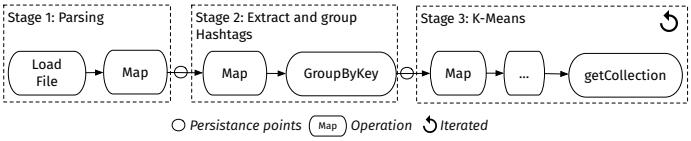

Fig. 3. Overview of the different stages of the K-means application.

Figure 3 shows the different stages in the K-means application that conceptually belong together and that are grouped in pipelined operations in the execution framework [3]. These stages are (i) parsing the JSON dataset in `Tweet` objects (ii) extracting the hashtags and related likes and group the tweets by hashtags, and (iii) running the K-means algorithm. Note that the K-means stage will be iterated several times for each of the different hashtags, as indicated by the arrow in the figure.

For this benchmark, we inject a failure-inducing record at the beginning of stage 2 of each partition, i.e. all workers fail just after the tweets are parsed. This allows us to measure the time needed to handle an exception and generate an online debugging session. We run the same experiment for datasets of different sizes, starting with a set of 5 GB up to 50 GB. We run the execution 20 times for each of the portions of the dataset, and each amount of parallel exceptions.

Results: Figure 4 shows the average execution time of the failing stage two, including the time to generate a debugging session, for each different size of the dataset (shown in different colors). The results show that the execution time of the failing stage is related to how many exceptions happen in parallel, not to the data size. It is slightly higher for bigger data sizes, but still in the order of hundreds of milliseconds. Thanks to the use of partition windowing and composite exceptions, our approach limits to the minimum the amount of data that needs to be transferred. This experiment shows that our debugging approach scales to applications that analyze big amounts of data, and to different exceptions happening simultaneously in the parallel execution.

5) RQ 2: How much time does online debugging save in comparison to replay and checkpoint-based debugging?: This section assesses the benefits of our debugging approach in comparison to replay and checkpoint-based debugging, using the results of the same benchmark used for RQ1.

Comparison to replay debugging: To compare our approach to replay debugging, we employ the same experiment as the previous section. Figure 4 showed that the execution time of stage 2 including the time of creating a dynamic local checkpoint, is lower than 400 milliseconds. As such, we deduce that the overhead of the debugging infrastructure on the execution is constant around hundreds of milliseconds and independent of the amount of data or the number of parallel exceptions. In a replay debugger, an error at the beginning

[3]Due to space restrictions, some operations happening during K-means have been omitted in the figure, as they are not relevant to our experiments.

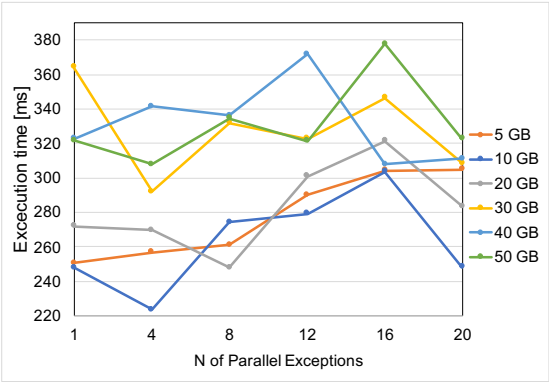

Fig. 4. Execution time of stage 2 when increasing the number of exceptions happening in parallel, for different sizes of the dataset.

of stage 2 requires to replay the whole execution from stage 1. The execution time of stage 1 in the previous experiment amounts to 46 seconds for the 5 GB file and up to 448 seconds for the 50 GB file. These results show that generating a dynamic local checkpoint is convenient in terms of runtime, since running again stage 1 takes 100 times more for the 5 GB file and 1000 times more for the 50 GB file.

Comparison to checkpoint-based debugging: To compare our approach to checkpoint-based debugging, we should first note that the experiment from previous section represents their best-case scenario: the developer manually has set a checkpoint at the beginning of stage 2, so a checkpoint-based debugger does not need to replay any execution. However, this quickly changes when an error occurs in the middle or at the end of a long stage, as explicit checkpoints are normally not created in the middle of a stage for performance reasons. In those cases, a checkpoint-based debugger would have to replay part of the stage, which is a more realistic case for the comparison. We thus run an experiment in which we inject the failure in the middle of stage 1 by removing an if-test for null.

In this experiment, each worker reads a part of the dataset and immediately starts parsing tweets, and iterates the execution 20 times. Since the previous benchmark showed that the size of the file does not impact the time to create a dynanmic local checkpoint, we use a 20 GB dataset.

The results show that each of the workers takes on average between 48 and 54 seconds to read its portion of the file from disk. The execution then fails, on average, between 100 and 500 milliseconds after starting the actual parsing, and the execution terminates after an average of 55.7 seconds of total execution time. When using a checkpoint-based solution, this time would need to be replayed to get to the error, since there is no persisting operation between the reading and the parsing. Furthermore, our solution does not require to manually add breakpoints or checkpoints to trigger the debugging. In comparison, as shown in Figure 4, the overhead of getting a debugging session with our approach amounts to hundreds of milliseconds.

From the experiments conducted for RQ2 we conclude that our approach provides *faster* access to a debugger by avoiding replay operations that both replay and (partially) checkpoint-based debugging approaches for Big Data rely on.

6) RQ 3: What is the overhead of ignoring exceptions, and how does it scale to a big number of exceptions?: This section first assesses the overhead of the infrastructure for ignoring exceptions on a failure-free execution, and then evaluates the scalability of ignoring an increasing number of exceptions.

a) Assessing the overhead: We assess the overhead of ignoring exceptions by turning the "ignore" mode on, but letting the applications complete their execution without errors. In this way, we simulate a setup in which the developer correctly filtered the initial dataset, but still wants to leave the "ignore" or "ignore and debug" modes on, in case something was missed in the data-cleaning process. We run this benchmark on the three applications using three portions of the database of increasing size (15GB, 30GB, and 45 GB), using as baseline the execution when the ignoring and debugging support is switched off, and comparing it to the execution with it turned on. Each application is executed at least 10 times, and a maximum of 25 times (depending on run-time).

App	$T_{baseline}$	T_{ignore}	Δ (%)
Grep 15G	0,77 ± *0,35*	0,76 ± *0,30*	**-0,60**
Grep 30G	1,09 ± *0,13*	1,16 ± *0,28*	**6,52**
Grep 45G	1,54 ± *0,15*	1,60 ± *0,11*	**3,92**
WC 15G	82,96 ± *5,42*	83,99 ± *3,56*	**1,24**
WC 30G	170,38 ± *8,98*	172,86 ± *10,55*	**1,46**
WC 45G	446,56 ± *18,90*	433,38 ± *22,72*	**-2,95**
KM 15G	171,15 ± *1,08*	180,11 ± *0,69*	**5,24**
KM 30G	263,44 ± *0,84*	273,91 ± *1,06*	**3,97**
KM 45G	385,62 ± *0,93*	392,44 ± *1,43*	**1,77**

Fig. 5. Runtime (in seconds) and overhead of running Grep (G), WordCount (WC), and K-means (KM) applications with and without ignore mode active.

Results: Figure 5 displays the runtime of the different applications for different sizes of the datasets, with the ignore mode turned off ($T_{baseline}$) and on turned on (T_{ignore}). We report the average time of the multiple iterations with the standard error. The last column presents the overhead of the ignore mode in terms of percentage of ($T_{baseline}$).

The table shows that the overhead varies between -2,95% to +6,52%. Looking more in detail at each application, we can see that Grep, the fastest benchmark, shows a big variation among the different data sizes, starting negative, then growing to 6%, and then lowering to 3,9%. The confidence intervals, however, overlap, which makes us conclude that there is no significant performance degradation. On the other hand, a more consistent overhead pattern appears in both WordCount and Kmeans: the overhead seems to lower when increasing the amount of data.

All in all, the activation of the ignore mode always introduces overhead when starting to execute a particular action, since it needs to initialize a replicated counter to count the possible amounts of exceptions. However, we observe that the higher the runtime of an operation, the lower the overhead.

b) Performance impact of ignoring an increasing number of exceptions: We now assess the performance impact

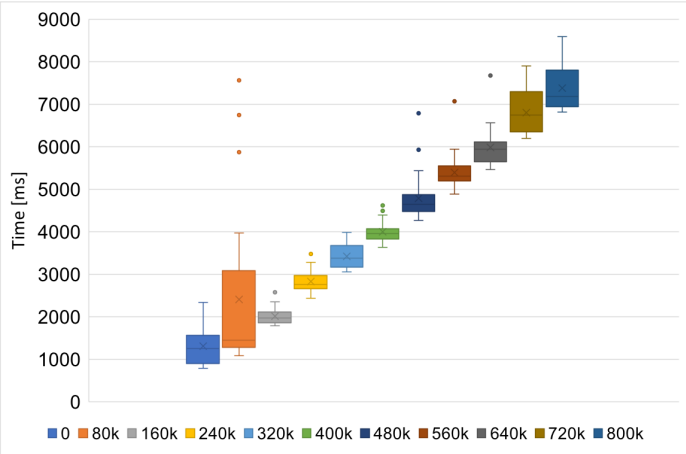

Fig. 6. Run-time when ignoring an increasing amount of exceptions.

of ignoring exceptions, when exceptions raised during the execution are ignored. To do so, we use the first phase of the K-means application, in which the tweets are parsed and then grouped by their hashtag. We gradually leave invalid tweets in the dataset to cause an exception when extracting the hashtags. It has to be noted that, we did not inject more invalid tweets in the dataset than the ones already present.

We distribute a fixed portion of the dataset among the workers and filter out all the failure-inducing records (*a.k.a.* `nil` records). We test using a dataset of 4.003.315 valid non-deleted tweets (roughly 20 GB of data), and we inject at every iteration 80.000 failures (corresponding again roughly to 2% of the original records). Since the original dataset presents 17% of failure-inducing records, we set the ignoring threshold to 50%, so we are sure to ignore all errors. We test the K-means application and measure the total time to completion.

Figure 6 displays the results of the experiment. Every execution was run 50 times, and the measure represents the amount of time in milliseconds it takes to parse and group tweets with the K-means application while injecting an increasing number of failure-inducing records. The results show that the runtime increases linearly with the number of ignored exceptions, with the overhead of ignoring a single exception ranging between 3 and 7 microseconds. We believe that this is an acceptable overhead when compared to re-executing the whole application possibly multiple times.

B. Qualitative Evaluation

In the qualitative evaluation, we assess the usability of our debugger in a study with 17 subjects that tested our debugger and a reimplementation of the closest related work, namely BigDebug [3], on top of the Spa runtime[4]. The user study measures both the time subjects take to solve the assignments with each debugger, and the impact of the features of both debuggers on the subject perception of debugger helpfulness, debugging difficulty, and number of redeployments. In the remainder of this section, we first describe the methodology of the study and then discuss different research questions.

1) Methodology: For this user study, we use a strong experimental design [23] to isolate and test the effect of the independent variable on the dependent variable. In particular, we use a within-participant design [23]–[25] in which all participants are exposed to each experimental condition, and fill in a post-test after being exposed to each of the conditions.

Experimental conditions: Each of the participants was exposed to two experimental conditions:

- Solve a debugging assignment using the Spa Debugger
- Solve a debugging assignment using BigDebug

To keep under control different external validity threats we randomize the order in which the participants are exposed to the two experimental conditions. We create two groups, a first group at first exposed to debugging with Spa, and later with BigDebug, and a second group exposed to the debuggers in the opposite order. We then assign randomly each participant to one group, trying to balance the size of the groups.

Throughout the study, we use two debugging assignments, always in the same order for both groups. Each debugging assignment presents two bugs in one application; the first bug is related to the formatting of the input data, and the second one to a logical error in the application (*i.e.* the final result is not correct).

2) Experiment setup: We conduct the study with 17 volunteer subjects (9 master students and 8 researchers), randomly split into the two different groups. Each participant executes the two assignments with both debuggers, but in a different order. The master students at least had followed (and passed) two master-level courses in which they had to program a project in Pharo Smalltalk and in Spark, respectively. All researchers had experience using Pharo Smalltalk, and knowledge of Spark.

a) The debugging tools: Participants employed the same IDE, i.e. the Pharo Smalltalk environment including Spa, but they interacted with the two different front-end debuggers corresponding to the Spa Debugger presented in this paper, and a reconstructed version of Big Debug on top of the Spa runtime. To keep the participants as unbiased as possible, the two debuggers are randomly renamed Debugger A and B, respectively, in all the presentations and in the debugger's UI.

The Spa Debugger offers all of the features described in this paper, but the ignore mode is enabled exclusively in global mode (to limit the explanations to a similar length than BigDebug's explanations). The BigDebug reimplementation provides the key features of the original work [3], namely guarded watchpoints, simulated breakpoints, stepping and resuming, record skipping, record substitution, code patching, and tracing to input. However, instead of being a separated web GUI as in [3], we integrate it in the Big Data framework's IDE, i.e Pharo Smalltalk. Please refer to Appendix B for screenshots of the UI.

b) Experimental material: Before doing a particular assignment, all the participants are handed a pdf describing the application and the bugs they will be looking for, as well as a cheat sheet containing information about the debugger,

[4]In this section, Spa Debugger refers to our debugger, and BigDebug refers to the reimplementation of BigDebug [3] in Spa.

the framework's API, and the language syntax. Furthermore, they also have access to the pdf of the presentation, including screenshots from the demo[5].

c) Structure of the study: Due to COVID-19 restrictions, we run the study in timed sessions in a campus room or a virtual one. The study, in both online and on-campus versions, has the same structure: First, a 15-minutes presentation and hands-on demo about programming with the Spa's model (cf. Section II-A) without debugging. Before each assignment, there is a 20-minutes presentation about the debugger they will use and a hands-on demo of debugging an application. At this point, the participants are handed the material and have 45 minutes to complete each assignment. Upon completion of the first assignment, they fill in the first part of the post-test that includes general questions about their experience in debugging and developing software, and questions about the debugger they have used. Then, after a small break, we repeat the same process for the second experiment (presentation of the second debugger, second assignment, post-test). Throughout the study, the host of the experiment is present in the (campus or virtual) room, answering privately to questions about the debuggers, as well as helping with technical issues.

d) Debugging assignments: In each assignment, participants have to solve two bugs: a first one causing an exception, a second one producing wrong results. Before tackling the second bug, they have to solve the first one. The system gives them feedback when they have solved the first and the second bug. Furthermore, the system automatically logs the activity of the participants while debugging, as well as the final code of the applications when they finish the experiment.

The first assignment is the debugging of the Twitter K-means application described in Section V-A2. For the first bug, we previously deleted the code that cleaned the data, which leads to data-cleaning errors because of deleted tweets that missed the hashtags. For the second error, we previously introduced a bug in the application's logic: the original dataset included tweets with non-ASCII characters that could not be displayed correctly within the final result. Those tweets have to be identified and removed.

The second assignment involves debugging an implementation of the popular ID3 decision tree algorithm [26] that analyzes a set of Amazon Reviews to find out which of the features of the review (*e.g.,* length of the text, amount stars, etc.) makes the review the most helpful (*i.e.* produced the highest count of the `helpfulVotes` feature). For the first bug, we modified the dataset by removing some of the features for some of the records, so that a data-cleaning error would appear when the application is first run. For the second bug, a wrong classification of the number of stars makes the algorithm return a final decision tree that is not correct. The wrong classification code has to be identified and corrected.

3) User study: The conducted user study aims to answer the following questions:

[5]All the experimental material is available at https://soft.vub.ac.be/~mmarra/userStudy/UserStudyMaterial.zip

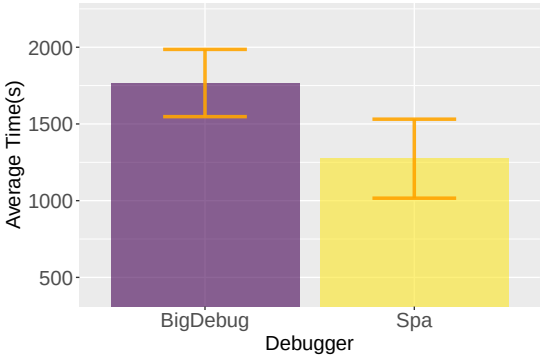

Fig. 7. Time to find the first bug with both debuggers (C.I.:80%).

RQ 4. Does the debugger impact the time to first bug?
RQ 5. Is there a difference in the overall debugging experience?
RQ 6. Did the code-updating capabilities influence the number of re-deployments?
RQ 7. How the features of the Spa Debugger were valued?

We now elaborate on the answers to different questions asked in the post-test to address our research questions.

a) RQ 4: Does the debugger impact time to first bug?: To answer this question, we automatically measured the time that each of the participants took to correctly identify and resolve the first bug. We did this for both applications and both debuggers. Figure 7 shows the average time to find the first bug, aggregated across the two applications and the two debuggers. The results show that the average time to find the first bug is lower for the Spa Debugger by 493 seconds (around 7 minutes). The confidence interval, calculated with an 80% of confidence, also shows that the Spa Debugger did reduce the time to find and solve the first bug.

Regarding the second bug, we do not report the full numbers, since the success rate was low for both debuggers, making the sample too small to draw conclusions. Only 6/17 participants found the second bug across the two applications while using the Spa Debugger, and 4/17 while using BigDebug. This may indicate that we underestimated the complexity of the second bug and that the 45 assigned minutes were not enough to solve both bugs.

b) RQ 5: Is there a difference in the overall debugging experience?: To answer this question, we asked in our post-test how much each debugger helped them in identifying the cause of the bugs, and whether debugging that particular application was difficult. Both could be answered using a Likert scale from 1 to 5 (1=Not at all, ..., 5=Very much).

How much did the debuggers help in solving the bugs?: Figure 8 presents a violin plot that shows that for BigDebug answers cluster around 2, while they cluster around 4 for Spa. The average, shown by the dot, is around 2.5 for BigDebug, and just above 4 for Spa. From these results, we conclude that the participants perceived as advantageous the features of the Spa Debugger, while they did so less for BigDebug.

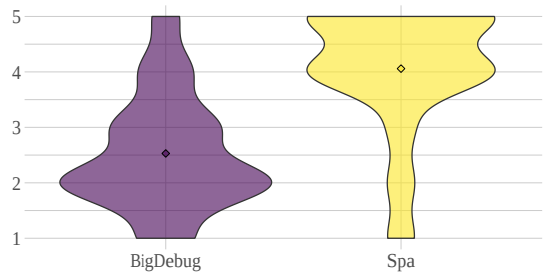

Fig. 8. Violin plot of the answers to "How much did the debugger help you in finding the bugs?", where 1 is *not at all* and 5 is *very much*.

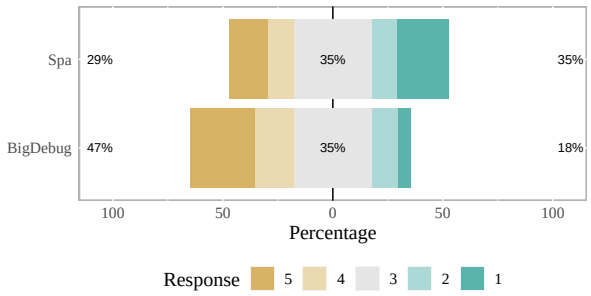

Fig. 9. Likert plot with the answers to "Debugging the application was difficult", where 1 is *very easy* and 5 is *very difficult*.

How difficult was debugging each application?: Figure 9 displays the answers of the participants to this question. When debugging with our debugger, 29.4% of participants declared that debugging the application was difficult/very difficult, against 47% for BigDebug. Accordingly, 35.6% of the participants declared that debugging the application with the Spa Debugger was easy/very easy, against 17.7% for BigDebug.

These results, together with the ones of the previous question, make us believe that debugging with our approach was perceived as easier and more helpful than with BigDebug.

c) RQ 6: Does the debugger influence the number of redeployments?: First of all, it is important to mention that the UI of both debuggers featured a button "Redeploy" which allowed participants to restart master and workers using the updated code-base, i.e perform a full redeployment of the application. In the post-test, we asked the participants how many times they had used the redeploy button.

Figure 10 shows a violin plot with the answers. We observe a clear difference between the two debuggers: with the Spa Debugger, on average participants had to redeploy 1 time, with most of them being clustered between 0 and 3. For BigDebug, participants re-deployed on average 5 times, being clustered between 4 and more than 5 times. Note that the average of BigDebug might be even higher because the questionnaire only allowed "more than 5" as the maximum value.

The number of lower redeployments for the Spa Debugger can be attributed to the live code-updating functionality, which allows to apply code patches without requiring a full redeployment. We discuss the general appreciation of the live code updating functionality in the next research question.

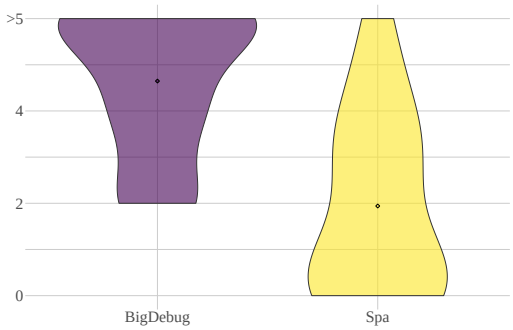

Fig. 10. Boxplot of redeployment count across the two applications with the two different debuggers.

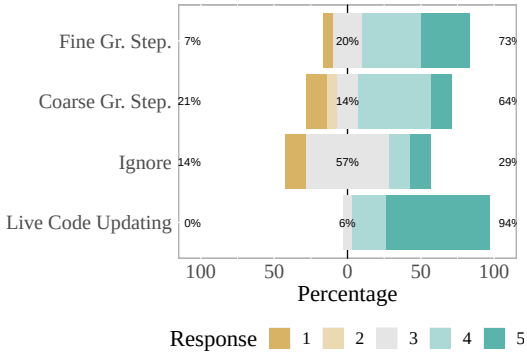

Fig. 11. Likert scale showing how useful each feature of the Spa Debugger was rated by the participants.

d) RQ 7: How the features of the Spa Debugger were valued?: To answer this question, we analyze the results of two post-test questions: one that asked directly how useful they would rate the debugging functionalities of the Spa Debugger, and one to check which feature of each debugger participants missed when using the other one.

How useful are the advanced debugging functionalities of the Spa Debugger?: To answer this question, participants were asked to rate how useful the debugger's features were using a Likert scale from 1 (not at all) to 5 (very useful). Particularly, we asked them how useful they found fine-grained stepping, coarse-grained stepping, ignore mode, and live code updating.

Figure 11 shows the answers for each feature. We observe that, except for the ignore feature, the majority of people gave a 4 or 5 rating to both kinds of stepping, as well as to live code updating. Actually, live code updating is the feature best rated (as it did not received any rating below 3). On the other hand, the ignore functionality was considered neutrally useful (with a rating of 3) by the majority of the participants.

Is there a feature of a debugger that you missed when using the other debugger?: This question was asked for both

debuggers after the participant had completed both the assignments. While no participant indicated a feature of BigDebug that was missed in our debugger, different participants indicated several features of our debugger as missing in BigDebug.

Feature	N of participants
Local debugging	11/17
Live code updating	6/17
Full breakpoints	3/17
Abstractions over the exceptions	1/17

TABLE I
FEATURES OF THE SPA DEBUGGER THAT WERE MISSING IN BIGDEBUG

Table I shows how many participants indicated in their answer one of the features of the Spa Debugger that they missed when debugging the assignment with BigDebug. Examples of answers that were classified as the "local debugging" feature are: "the interactive debug session" and "Normal stepping, easy browsing and editing code". The other classifications took into consideration whether they explicitly mentioned the feature in their comment. Interestingly, one participant explicitly answered: "Abstractions over the exceptions" as a feature they missed in BigDebug.

From the results in this question, we conclude that the participants generally appreciated the features of the Spa Debugger. This is also confirmed by the answers to the question "Which feature of Spa did you find useful?", in which 16 out of 17 participants selected "Live Code Updating", 15 selected "Debugging locally a remote exception in an interactive way", and 9 selected "Breakpoints on parallel execution" and "Advanced Stepping Operations".

4) Threats to validity: There are different factors that may have influenced the results of our user study:

a) Number of participants and participant profiles: Since we required a particular profile of participants and we had a limited time frame, we were only able to recruit 17 participants. For this reason, we opted for a within-participants study, which allowed us to maximize the data points. This approach led to the presence of more explanation of the tools, since the computational model and both debuggers were explained to the participants. Furthermore, by having to use both debuggers, participants may have been affected by fatigue more than if we had run an inter-participant study.

b) Bugs representativity and difficulty: The assignments were both based on popular algorithms for data analysis, although not all our participants were knowledgeable of them. The bugs, however, were not taken by common reports but from personal experience in debugging the applications. This may have led to too difficult bugs to be solved in the time frame of the assignments; while all of the participants except one solved the first bug in both applications, only 6 and 4 participants solved correctly the second bug in the first and second assignment, respectively.

c) BigDebug reproduction: Participants used a reimplementation of BigDebug since using two different runtimes and languages would have made both the setup and experiments more complex. To reproduce BigDebug, we used the information available on the original paper [3], the project website and demo videos publicly available, as we did not manage to successfully build BigDebug from its public repository and test it.

VI. RELATED WORK

This section discusses related work on debuggers for Big Data frameworks, and on acceptability-oriented computing.

a) Big Data debuggers.: Most of the debugging solutions for Big Data consist in replay debuggers [27], such as Arthur [2] and Graft [28], that record and store events of one execution to then replay them later for debugging. These solutions, however, require tedious replay times and advanced knowledge of which executions to record. More recently, Daphne [29] and BigDebug [3] combined replay debugging with some simil-online debugging capabilities: Daphne introduces breakpoints for DryadLINQ [30] queries, allowing developers to debug remotely a certain node or to replay locally a certain execution. BigDebug introduces *simulated breakpoints* to add some online debugging primitives (such as stepping), by partially replaying the execution from the last checkpoint with respect to where the breakpoint was placed. As discussed in Section V-A4, this may still introduce relevant replaying times. BigDebug also offers some post-mortem debugging capabilities, such as crash analysis through data provenance, to detect which part of the execution failed, that has been further studied in BigSift [4]. Both the solutions, however, are post-mortem and require replaying different parts of the execution. Finally, IDRA_{MR} [12] is a debugger for Map/Reduce applications. Compared to our approach, it is not generalized to work with a broad Spark-like API, does not present partition windowing, and does not support debugging a pipeline of operations on a distributed data structure.

b) Acceptability-oriented computing applied to Big Data.: Acceptability-oriented computing was first defined by Rinard [14] as a failure-oblivious system that describes the properties that state and behaviour that a system must preserve for a program's execution to be acceptable, and that then monitors, and enforces these acceptability properties and eventual violations. Later, Carbin et al. [15] further define *relaxed programs* as programs that "have been extended with additional nondeterminism to relax their semantics and enable grater flexibility to the execution". The concept of relaxed programs and failure-oblivious systems were later used to prevent buffer overflows in C [10], and to test the reliability of Java programs [9].

To the best of our knowledge, such an approach was never applied as such to Big Data applications. We have found however in Hadoop Map/Reduce a static parameter allows developers to define a certain percentage of tasks that are allowed to fail. However, this parameter is very coarse grained: it refers to tasks, as opposed to records as in our approach. Furthermore, this relates to all jobs that will be executed, and cannot be changed, added or removed at runtime. Our approach, instead, allows developers to set a threshold to ignore exceptions in specific parts of the execution, that can be set dynamically at run time.

VII. Conclusion

In this paper we explored and evaluated an online debugging approach tailored to Spark-like applications. We propose a practical always-active debugging approach, that combines ideas of out-of-place debugging [11] and dynamic local checkpoints to allow online debugging of Spark-like applications. Debugging of a remote exception/breakpoint happens locally, and is augmented with advanced stepping operations to improve the debugging experience. To ease the re-deployment of the application, the debugger supports live code updating in the cluster. Furthermore, our debugging approach features built-in support for automatic ignoring of exceptions, that developers can use to avoid tedious data cleaning errors.

We show using several benchmarks that the debugging overhead is comparable or better to replay and checkpoint-based debugging and that our debugging approach scales well when increasing the size of the analyzed data and the number of parallel failures. Moreover, we show that the ignoring exceptions infrastructure introduces negligible overhead on a non-failing program, and that the overhead of ignoring failures grows linearly with respect to the number of ignored failures.

We complement our quantitative evaluation with a user study with 17 subjects, in which we (i) compare our approach to the closest related work and (ii) assess the usefulness of the introduced features of our debugging approach. The results show that our approach improves the time to find the solution to a bug, made finding the bug easier, and reduced the amount of full re-deployments of the applications. Furthermore, participants generally evaluated positively the features of our debugger, especially live code updating and the possibility to do fine-grained stepping while debugging a failed execution. We conclude that the proposed online debugging approach presents an improvement for debugging Spark-like applications.

References

[1] D. Pacheco, "Postmortem Debugging in Dynamic Environments," *Commun. ACM*, vol. 54, no. 12, pp. 44–51, 2011.

[2] A. Dave, M. Zaharia, S. Shenker, and I. Stoica, "Arthur: Rich post-facto debugging for production analytics applications," *Technical report, University of California*, 2013.

[3] 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*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 784–795.

[4] M. Gulzar, S. Wang, and M. Kim, "Bigsift: automated debugging of big data analytics in data-intensive scalable computing," 10 2018, pp. 863–866.

[5] 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*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[6] Apache, "Apache spark," http://spark.apache.org/, 2020, accessed: 2020-05-12.

[7] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth, "Master–Worker: An Enabling Framework for Applications on the Computational Grid," *Cluster Computing*, vol. 4, no. 1, pp. 63–70, 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011416310759

[8] M. Muller, I. Lange, D. Wang, D. Piorkowski, J. Tsay, Q. V. Liao, C. Dugan, and T. Erickson, "How data science workers work with data: Discovery, capture, curation, design, creation," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 115. [Online]. Available: https://doi.org/10.1145/3290605.3300356

[9] L. Zhang and M. Monperrus, "Tripleagent: Monitoring, perturbation and failure-obliviousness for automated resilience improvement in java applications," 10 2019, pp. 116–127.

[10] M. Rigger, D. Pekarek, and H. Mössenböck, "Context-aware failure-oblivious computing as a means of preventing buffer overflows," in *Network and System Security*, M. H. Au, S. M. Yiu, J. Li, X. Luo, C. Wang, A. Castiglione, and K. Kluczniak, Eds. Cham: Springer International Publishing, 2018, pp. 376–390.

[11] M. Marra, G. Polito, and E. Gonzalez Boix, "Out-of-place debugging: a debugging architecture to reduce debugging interference," *The Art, Science and Engineering of Programming*, vol. 3, no. 2, pp. pp. 3:1–3:29, October 2018.

[12] M. Marra, G. Polito, and E. G. Boix, "A debugging approach for live big data applications," *Science of Computer Programming*, vol. 194, p. 102460, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642320300708

[13] M. Marra, G. Polito, and E. Gonzalez Boix, "Framework-aware debugging with stack tailoring," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 7184. [Online]. Available: https://doi.org/10.1145/3426422.3426982

[14] M. Rinard, "Acceptability-oriented computing," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 03. New York, NY, USA: Association for Computing Machinery, 2003, p. 221239. [Online]. Available: https://doi.org/10.1145/949344.949402

[15] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 12. New York, NY, USA: Association for Computing Machinery, 2012, p. 169180. [Online]. Available: https://doi.org/10.1145/2254064.2254086

[16] Q. Shi, H. Hoffmann, and O. Khan, "A hw-sw multicore architecture to tradeoff program accuracy and resilience overheads," *IEEE Computer Architecture Letters*, vol. 14, pp. 1–1, 10 2014.

[17] Zinc, "Zinc," http://zn.stfx.eu/zn/index.html, 2020, accessed: 2020-05-15.

[18] M. Dias, M. M. Peck, S. Ducasse, and G. Arévalo, "Clustered serialization with fuel," in *Proceedings of the International Workshop on Smalltalk Technologies*, ser. IWST '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:13.

[19] M. Dias, D. Cassou, and S. Ducasse, "Representing code history with development environment events," *CoRR*, vol. abs/1309.4334, 2013. [Online]. Available: http://arxiv.org/abs/1309.4334

[20] J. W. Creswell and V. L. P. Clark, *Designing and Conducting Mixed Methods Research*. SAGE Publications, 2017.

[21] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 488–499.

[22] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–9.

[23] L. B. . Christensen, *Research methods, design, and analysis*, 12th ed., B. Johnson, Ed. England: Pearson Education Limited, 2015. [Online]. Available: https://opac.uitm.edu.my/opac/detailsPage/detailsHome.jsp?detailLinking=true&_rqst=undefined&db=null&tid=564937

[24] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Rand McNally College Publishing, 1963. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike-20&path=ASIN/B000BWLQBE

[25] W. Shadish, T. Cook, and D. Campbell, *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Unknown Publisher, 2002.

[26] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986. [Online]. Available: https://doi.org/10.1007/BF00116251
[27] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593–622, Dec. 1989.
[28] S. Salihoglu, J. Shin, V. Khanna, B. Q. Truong, and J. Widom, "Graft: A debugging tool for apache giraph," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1403–1408. [Online]. Available: http://doi.acm.org/10.1145/2723372.2735353
[29] 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*, Anchorage, AK, USA, May 2011, pp. 1266–1273.
[30] Microsoft, "Dryadlinq," https://www.microsoft.com/en-us/research/project/dryadlinq/, accessed: 2017-05-10.

APPENDIX

A. Spa UI

In this section, we show and describe the most important views and details of our debugger. Due to space restrictions, we did not include these screenshot in the main text of this paper. Doing so, would have meant showing only some of the screenshots and in a lower resolution.

These screenshots are the ones presented to the participants of the user study to explain how to use the tool, and are the ones the participants actively used during the interactive demos.

a) The debugger tab: Right in the centre of the Spa UI, three tabs display the current state of the execution, the debugger (currently selected), and the interface code changes handler to propagate updates to the remote nodes. In Figure 12, you can see the Debugger tab. The left panel of the debugger tab displays the list of the composite exceptions that happened. In this case, one `ZeroDivide` exception happened during a `map:`. The right panel shows the stack of the exception, as well as the list of records that caused the exception and the lineage of the operation. Finally, on the bottom right, different buttons are presented to start the online debugging session of the selected exception and data.

b) Debugging an execution: When debugging a Spa execution (because of a breakpoint or an exception), the developer is presented with the view presented in Figure 13.

This is a classical Pharo debugger, that displays in the top view the call-stack, and in the bottom view the source of the selected method. In the bottom part, not visible in this screenshot, the Pharo debugger also shows the state of the variables of the selected stack frame, as well as an evaluator

c) The code manager tab: Figure 14 shows a screenshot of the code manager tab. In here the left panel shows a list of exceptions, while the right panel shows on top a list of code changes and in the bottom the diff in source code of the selected code change. Finally, the bottom left panel presents two buttons, one to reset the changes set (i.e. to delete the current changes from the ones that must be propagated at the next commit); the second button is the button to commit the changes to the workers.

d) Changing execution mode: Finally, the execution mode can be changed using the combobox displayed in Figure 15.

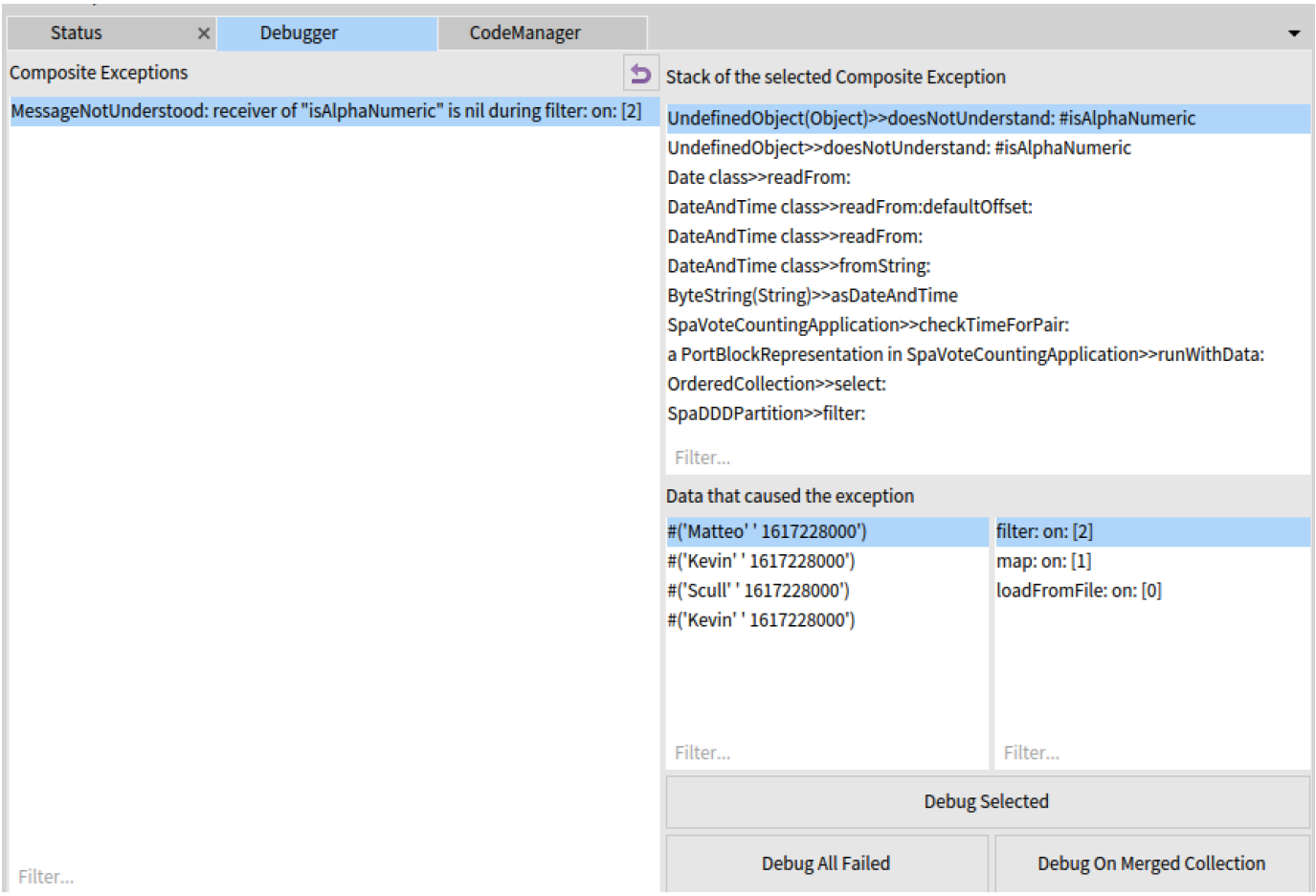

Fig. 12. The debugger tab.

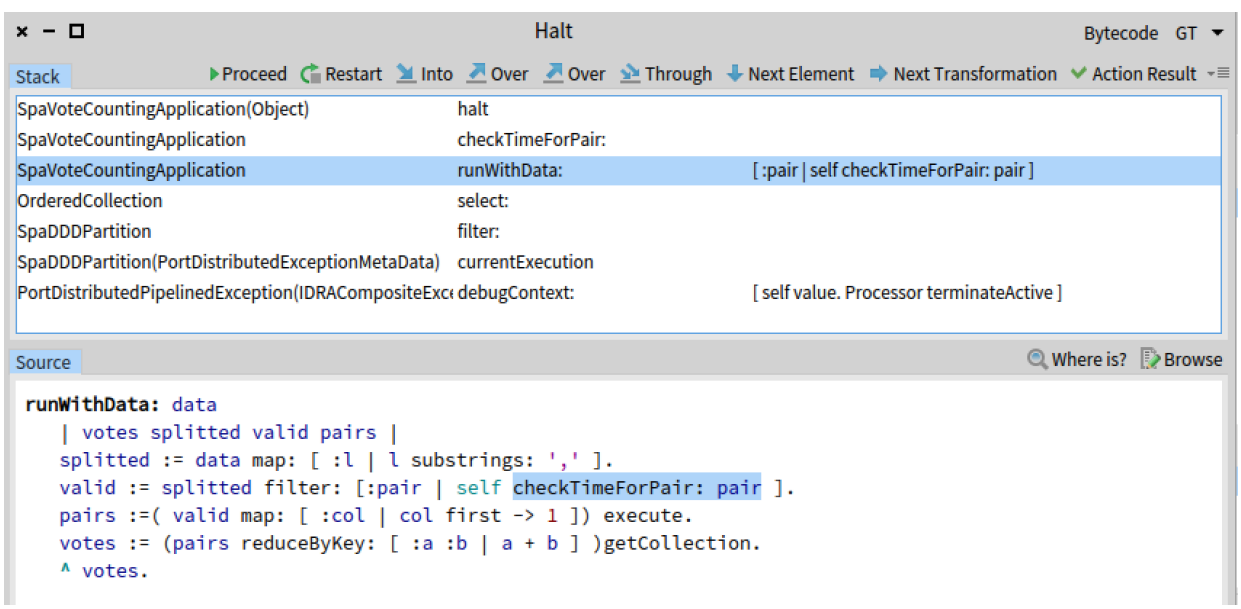

Fig. 13. Spa Debugger view.

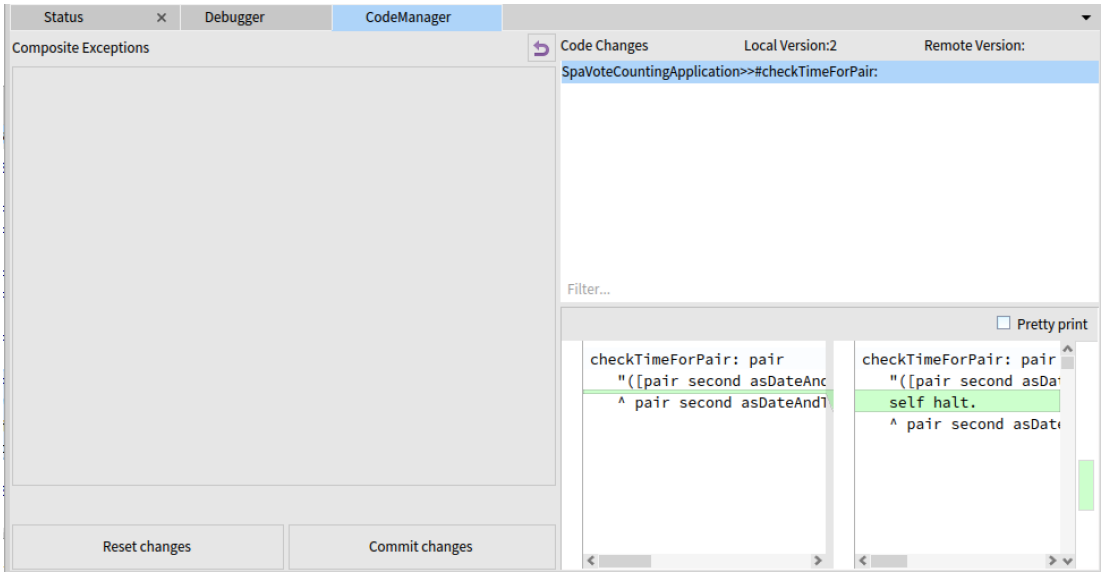

Fig. 14. The code updating tab of Spa.

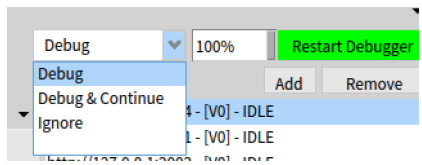

Fig. 15. The execution mode selection view of Spa.

B. BigDebug UI

As we did for Spa in the previous section, in this section we show and describe the UI of our implementation of BigDebug, that was used by the participants in the user study

a) The debugger tab: As for Spa, the participants could access the data of an exception or watchpoint in the debugger tab, displayed in Figure 16. On the left side, you can see the execution pipeline, in which the filter is marked with a red exclamation mark, indicating that the error is there. On the right side, under watchpoint, you can see all the records that caused the exception. By selecting one record, the rightmost view updates showing the textual form of the record. The three buttons in the bottom right allow to modify the record (upon changing its textual form), skip the record, or execute the tracing to input. Finally, the two buttons in the bottom left side allow to step (over) a transformation, and to resume the execution.

b) The stack trace: As in the original BigDebug, the stacktrace is shown in the Pharo Transcript (a sort of standard output). Figure 17 displays an example of such a stack trace.

c) Code patching: Finally, patching the code executed by a certain transformation happens in a similar way to the original BigDebug. By double clicking on one of the transformations in the left panel of Figure 16, the dialog shown in Figure 18 would pop up, giving the participant the possibility to fill in a closure to be applied instead of the original one.

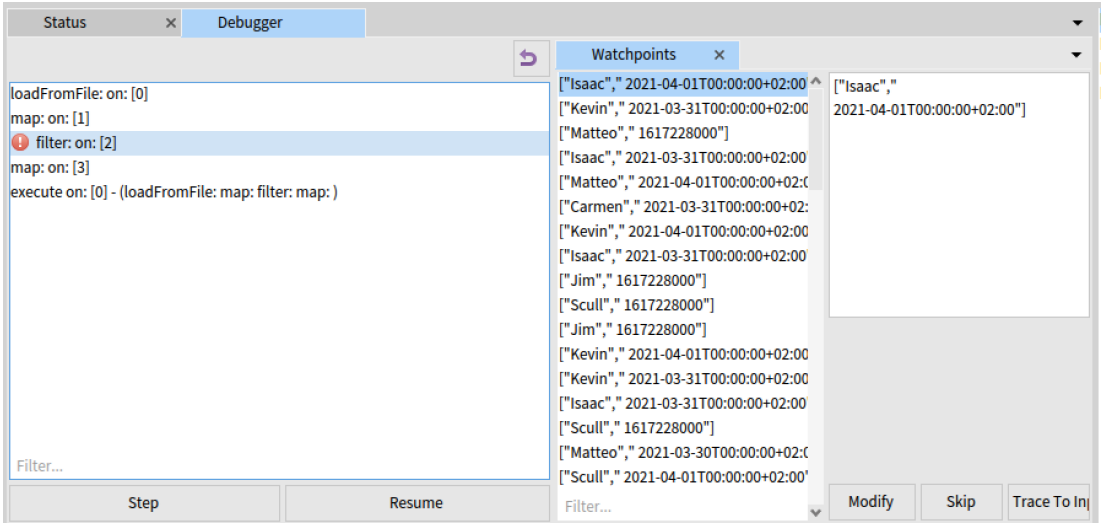

Fig. 16. The debugger tab of BigDebug.

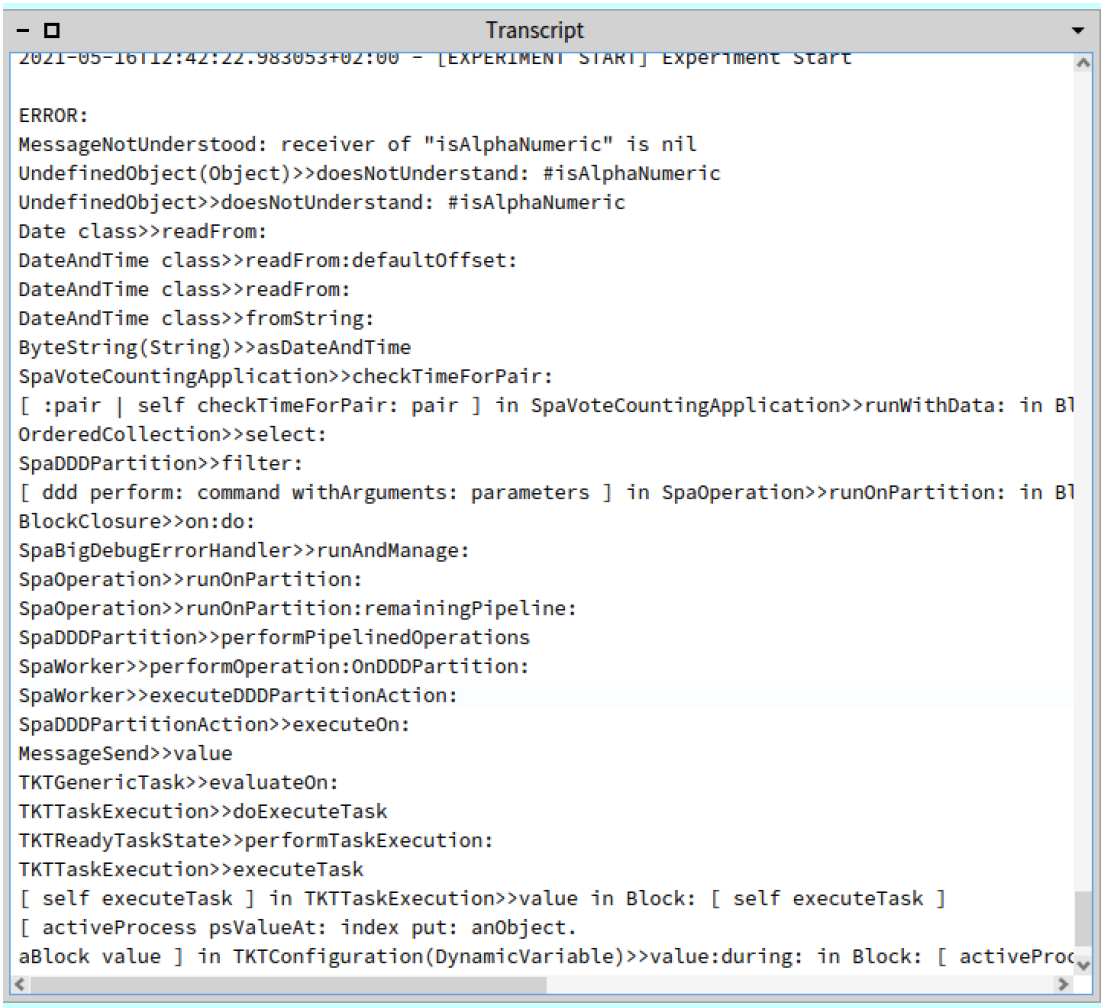

Fig. 17. Stacktrace of an exception as shown in BigDebug.

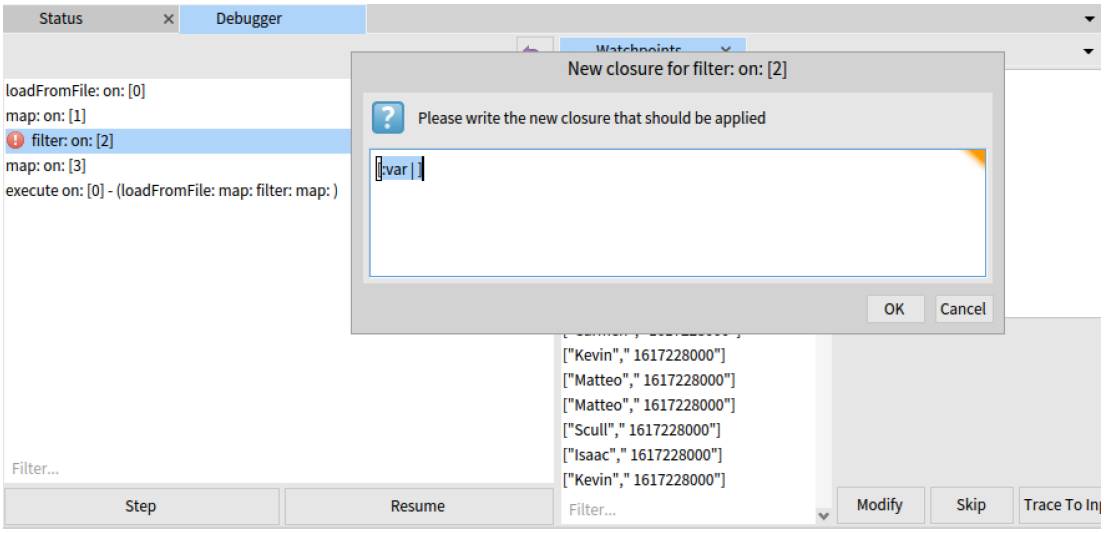

Fig. 18. Code patching in BigDebug.