



MASTER IN
COMPUTER
SCIENCE

Debugging Spark Applications

A Study on Debugging Techniques of Spark Developers

Master Thesis

Melike Gecer
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

May 2020

Prof. Dr. Oscar Nierstrasz
Dr. Haidar Osman

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Debugging is the main activity to investigate software failures, identify their root causes, and eventually fix them. Debugging distributed systems in particular is burdensome, due to the challenges of managing numerous devices and concurrent operations, detecting the problematic node, lengthy log files, and real-world data being inconsistent.

Apache Spark is a distributed framework which is used to run analyses on large-scale data. Debugging Apache Spark applications is difficult as no tool, apart from log files, is available on the market. However, an application may produce a lengthy log file, which is challenging to examine.

In this thesis, we aim to investigate various techniques used by developers on a distributed system. In order to achieve that, we interviewed Spark application developers, presented them with buggy applications, and observed their debugging behaviors. We found that most of the time, they formulate hypotheses to allay their suspicions and check the log files as the first thing to do after obtaining an exception message. Afterwards, we use these findings to compose a debugging flow that can help us to understand the way developers debug a project.

Contents

1	Introduction	1
2	Technical Background	4
2.1	Spark Architecture	4
2.1.1	How Does Spark Run an Application on the Cluster?	5
2.1.2	Spark with YARN	6
2.1.3	HDFS	6
2.2	Our Experimental Spark Cluster	6
3	Experimental Setup	7
3.1	Debugging Stories	7
3.2	Reproducing Bugs	8
3.2.1	The First Bug: Task Not Serializable	8
3.2.2	The Second Bug: 64 KB JVM Bytecode Limit Problem	10
3.2.3	The Third Bug: Corrupted Data	10
3.2.4	The Fourth Bug: Excess Transformations Problem	10
3.3	Developer Interviews	11
3.3.1	Professional Information Questionnaire	11
4	Discovering Strategies	13
4.1	Correctness and Speed	13
4.2	Answering RQ1: Discovering Strategies	14
4.2.1	Patterns	14
4.2.2	Hidden Markov Model	17
4.2.3	Debugging Flow	19
5	Challenges and Missing Tooling	23
5.1	Post-Mortem Questionnaire	23
5.2	Answering RQ2: Requirement Collection	24
5.3	Answering RQ3: Missing Tools	25
6	Related Work	27
7	Conclusion and Future Work	32
A	Setting Up an Apache Spark Cluster	34
A.1	Upgrading Operating Systems	34
A.2	Installing Java and Scala	35
A.3	Installing and Setting Up an Apache Hadoop Cluster	36
A.4	Installing Apache Spark	38

B	Implementation of Bugs	43
B.1	Implementation of the First Bug: Task Not Serializable	43
B.2	Implementation of the Second Bug: Stack Overflow	45
B.3	Implementation of the Third Bug: Corrupted Data	47
B.4	Implementation of the Fourth Bug: Stack Overflow	48
C	Debugging Flow	51

1

Introduction

A bug is an error in a system that results in a faulty operation or a system failure. Bugs can arise due to various reasons, such as carelessly designed logical structures and typos. Software failures caused \$1.7 trillion in financial losses in 2018 [12]. Software bugs are the most frequent cause of such failures [12].

Developers use debugging tools to analyze programs and investigate bugs in their code. Such tools allow them to step in and step out of code to investigate its behavior, or to track value changes in variables. Most integrated development environments (IDE), like IntelliJ IDEA or NetBeans IDE, come with their own debugging tools.^{1 2}

In 1997, Lieberman [15] claimed that many programmers are still using “print” statements to debug their code despite the existence of debugging tools. 14 years after Lieberman’s paper was published, Yin *et al.* [25] mention that we are facing “the next generation of debugging scandal” and debuggers are not able to keep pace with the development of programming languages. 6 years later, Beller *et al.* [5] conduct a survey and observe that 8 out of 15 programmers do not use the IDE-provided debugging infrastructure and prefer using “print” statements instead.

There are several factors that make debugging a difficult task. First of all, the complexity of a program plays an important role. As complexity increases, bugs become trickier to solve. Secondly, while it might be easy to observe the consequences of a bug, it is often no small matter to identify its cause. If the root of a problem is unknown, then it is harder to understand and reproduce than solve. Finally, problems might occur not only in the program itself, but also in any of the dependencies used to implement it. When a program is modified on a regular basis, it can become difficult to track which dependencies do not work anymore and require updating. If a dependency is a crucial component of a system, its malfunction can render the entire program useless.

In distributed systems, Debugging is even more challenging for four primary reasons:

1. In distributed systems, operations are executed concurrently. Deadlocks and race conditions might occur while waiting for other members of the system to operate or to share resources [6, 14, 18]. Debugging becomes difficult due to the nondeterministic behavior of systems and the difficulty to reproduce failures.

¹<https://www.jetbrains.com/help/idea/debugging-code.html>

²<https://netbeans.org/features/java/debugger.html>

2. By definition, distributed systems consist of many devices of different kinds (*e.g.*, mobile devices, servers, home electronics) which have different software and hardware properties [6]. It is challenging to design a system that can communicate with numerous different devices [2, 6, 16, 18, 20, 21]. A node crash is confusing to tackle and detect in a distributed system.
3. Logs are the only source of information allowing developers to solve problems that occur in their application. However, logs can be quite lengthy, making it difficult for developers to investigate.
4. Real-world data is always “dirty”, *i.e.*, it contains errors and inconsistencies. Developers of big data applications prefer using distributed systems to process and analyze large-scale data. Thus they need to handle such errors and inconsistencies in data.

In this thesis, we seek to understand the current state of debugging distributed systems and the tools used in the process. More specifically, we want to investigate how distributed systems developers tackle problems in their applications and to shed light on their bug-fixing techniques. As a subject distributed system for our investigation, we choose Apache Spark for three main reasons:

1. It is one of the most widely-used distributed computing frameworks.³
2. Apache Spark provides support for a wide range of Apache solutions. In the scope of this thesis, we used Apache Spark, Apache Yet Another Resource Negotiator (YARN), and Apache Hadoop Distributed File System (HDFS).⁴
3. Apache Spark supports many programming languages (*e.g.*, Scala, Java, and Python).⁵

As an experimental environment, we set up an Apache Spark cluster, identified and reproduced the four most common problems that developers encounter. Then, we held interviews with seven professional Apache Spark developers in order to observe their debugging methodologies. With their consent, the screen and audio were recorded during the interviews. We went through the recordings, analyzed the steps taken to debug an application, and found patterns which we used to form a debugging workflow. Later, we describe the tool ideas that are suggested by interviewees.

After analyzing the results, we observe that the most common steps to debug an application are making hypotheses, reading exception messages, and checking logs. We observe that when the problem is related to data:

- Three out of seven interviewees check the file on the cluster;
- Two out of seven interviewees check the file on the local machine.

Furthermore, six out of seven interviewees check the class (if the exception message explicitly states its name), *i.e.*, they find the line where the exception is thrown, trace the method call and try to identify the problem.

Using these findings, we design a unified debugging flow and explain how developers debug their applications. The flow consists of two main parts, namely: finding the root of the problem and solving the problem. All of our interviewees have a way (*e.g.*, using orchestration frameworks) to be acknowledged that a job has failed. However, as supported by our findings in the Markov model and the most frequent steps previously mentioned, developers find it challenging to read the logs and solve the problem. We find that developers start with checking the logs and coming up with a hypothesis. There are also common

³https://github.com/thedataincubator/data-science-blogs/blob/master/output/DC_packages_final_Rankings.csv

⁴<https://hadoop.apache.org/old/>

⁵<https://spark.apache.org/docs/2.2.0/>

ways to solve problems. For instance, if the problem is related to the code, they try to solve the problem using a local debugger; or they check the file properties when the problem is related to data.

In order to understand the debugging methodology of Spark applications, we pose the following research questions (RQ) to guide our study:

RQ1: How do developers debug Spark applications?

We learn that developers prefer to debug their codes on a local environment whenever possible. To achieve this, they use the IDE debugger. In addition, they prefer to use different interactive environments such as Jupyter Notebooks, in which they can check code one line at a time.⁶ They use logs to detect problems by locating the exception messages.

RQ2: What are the challenges that developers face when they debug such applications?

We observe that developers agree that logs are not always enough to gain insight into the problem. They believe that some problems occur only on the server cluster, thus debugging on a local environment does not help.

RQ3: How can we overcome these challenges?

We enquire interviewees about the tools they need the most. They recommended tools that can be split into three categories: code experience enhancements, log enhancements, and dashboard implementation.

We find it interesting that our interviewees use external tools to schedule jobs and to be notified regarding submitted jobs. In the same way, we would expect them to use some debugging tools that are developed by researchers. Instead of using such tools, our interviewees use various methods, such as print statements, local tests and debuggers. This may be due to a lack of commercial debugging tools.

Outline

- In *Chapter 2*, we give insights about internal parts of Apache Spark and our experimental Spark environment.
- In *Chapter 3*, we explain the four main bugs we chose and how we chose them. We introduce our interviewees and display statistics based on the first questionnaire.
- In *Chapter 4*, we answer RQ1 by presenting the results, the Hidden Markov Model, and the unified debugging model.
- In *Chapter 5*, we answer RQ2 and RQ3 by presenting difficulties our interviewees encountered as well as their suggestions for tools that would make their life easier.
- In *Chapter 6*, we present and discuss various distributed debuggers that were implemented in related works.
- In *Chapter 7*, we conclude the thesis and present future works.

⁶<https://jupyter-notebook.readthedocs.io/en/stable/>

2

Technical Background

2.1 Spark Architecture

Spark is an open-source cluster-computing system.^{1 2} It is widely used in industry because of its scalability and speed. It works in-memory whenever possible, which is faster than disks and thus represents a great advantage for algorithms requiring intensive computation (*e.g.*, machine learning algorithms). Spark supports the Scala, Python, Java, and R languages.^{3 4 5 6} Moreover, Spark has high-level libraries that can be used on top, such as the Machine Learning library (MLlib) and Spark Streaming.^{7 8} For these reasons, we chose Spark as our subject distributed system to study.

Spark is based on the Resilient Distributed Dataset (RDD) abstraction. RDD is a collection of immutable objects, which means the records cannot be changed. In Spark we can perform transformations and actions on data. A transformation produces a new RDD from the existing RDDs after applying operations such as `map()`, `filter()`, or `union()`. An action, on the other hand, can be performed on the actual dataset, *i.e.*, directly on the files themselves (with operations like `count()`, `collect()` or `saveAsTextFile(path)`).

As can be seen in Figure 2.1, a Spark cluster consists of a master node and worker nodes. The master node controls the cluster and assigns tasks to workers. Worker nodes execute tasks on partitioned RDDs and return the results to Spark Context. Spark Context is a significant part of any Spark application because it serves as a connection between the application and the Spark cluster.

Spark provides event logs for its applications, which developers can review through a web User Interface (UI). These logs are the only source of information to find exception messages.⁹ If YARN (which will be introduced shortly hereafter) and Spark are configured together, YARN's Resource Manager, as

¹<http://spark.apache.org/docs/latest/index.html>

²For the sake of simplicity, we chose to refer to Apache Spark as Spark.

³<https://www.scala-lang.org/>

⁴<https://www.python.org/>

⁵https://www.java.com/en/download/whatis_java.jsp

⁶<https://www.r-project.org/>

⁷<http://spark.apache.org/docs/latest/ml-guide.html>

⁸<http://spark.apache.org/docs/latest/streaming-programming-guide.html>

⁹<https://spark.apache.org/docs/latest/monitoring.html>

seen in Figure 2.2, can be used to go through the application logs.¹⁰

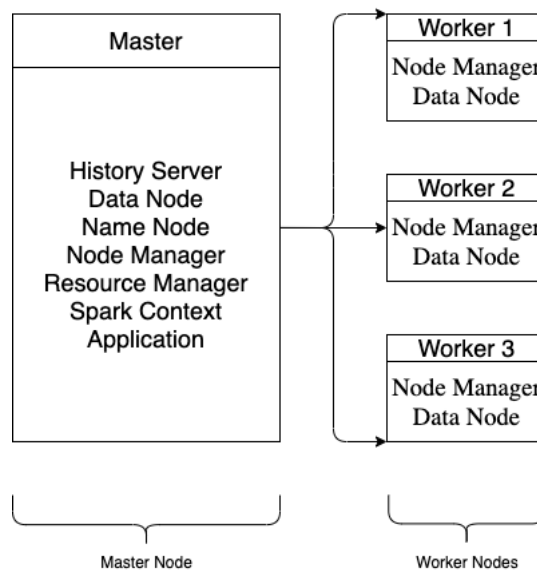


Figure 2.1: A basic Spark cluster architecture

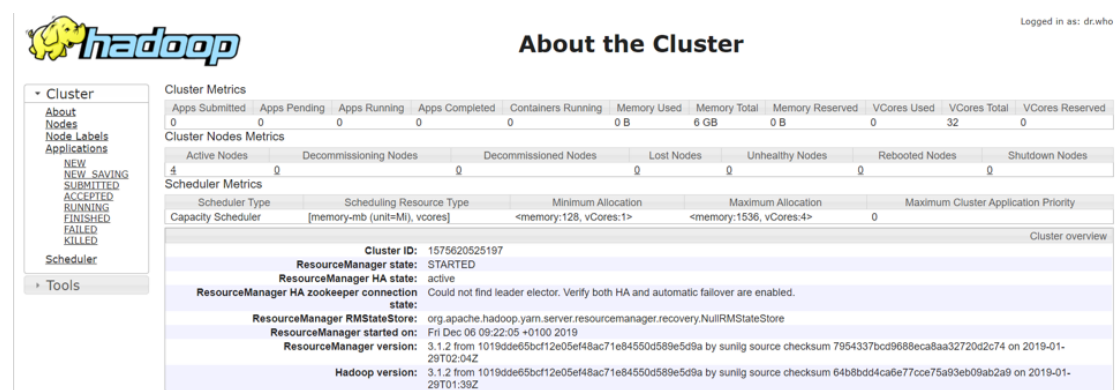


Figure 2.2: Resource Manager UI of Hadoop. Using the dropdown menu on the left, we can check the cluster specs as well as the list of applications and their status. When we click on an application ID, we can observe that application’s logs, DAG, history, and output (if any).

2.1.1 How Does Spark Run an Application on the Cluster?

When a job is submitted to a Spark cluster, Spark creates a Spark context to serve as a gateway, as well as a Directed Acyclic Graph (DAG), which is a logical graph representation of the Spark job. Spark then applies some optimizations (*e.g.*, a pipeline of transformations) and creates small execution plans known as “tasks”. The tasks are sent to the Spark Cluster, which forwards them to the Driver Program. The driver

¹⁰For the sake of simplicity, we chose to refer to Apache Hadoop YARN as YARN, and Apache Hadoop as Hadoop.

program manages resources and interacts with the Cluster Manager. Finally, the driver program launches executors on worker nodes to perform the tasks. Once executions are completed, the results are sent to the Cluster Manager, which forwards the results to the Driver Program. When the driver program exits, it releases all the resources and terminates the executors.

2.1.2 Spark with YARN

YARN is a system designed to manage clusters, which includes job scheduling and resource management. It has two essential components: a resource manager and a node manager. The resource manager is the master node that communicates with YARN workers, manages cluster resources, orchestrates tasks, and coordinates the application master, which is used for requesting resources, with the node manager. Node managers are responsible for managing their own resources; they launch and keep track of processes assigned to them.

2.1.3 HDFS

HDFS is a highly fault-tolerant distributed file system that can handle large data sets. It has two essential components, namely the name node and data nodes. The name node is the master that manages the file system and regulates access to files, *e.g.*, to open a file in the system. Data nodes, on the other hand, manage data storage on worker nodes.

2.2 Our Experimental Spark Cluster

Our Spark cluster is set up using four servers, which run on Debian 9.7, Ubuntu 18.08 and Ubuntu 16.04.

Spark works with Java, Scala, and Python. We choose to use Scala with Spark. Thus, we need to install Scala 2.11, which also requires JDK 8. Appendix A.2 contains a brief tutorial on how to install both Java and Scala.

The next step is to install Hadoop and Spark, which are configured to work together. We have Spark 2.0.2 installed on all servers. We describe the step-by-step installation and the configuration of a Spark cluster with Hadoop from scratch in Appendices.

Besides a node manager and a data node, the master node contains a history server to the display status of Spark jobs, a name node, and a resource manager. A presentation of the cluster can be seen in Figure 2.1. The most important configuration files of the Hadoop system are *core-site.xml*, *hdfs-site.xml*, *mapred-site.xml*, and *yarn-site.xml*. As for the Spark environment, the *spark-defaults.conf* file is essential. These configuration files allow Hadoop and Spark to communicate with each other and with the operating systems.

3

Experimental Setup

3.1 Debugging Stories

In order to observe the way a developer debugs a project, we need a set of simple applications that produce failures. Thus, we search for the most frequently encountered problems in Spark that we can reproduce during the interviews. To that end, we turn to Stack Overflow, one of the largest communities where developers can discuss their questions and help each other.¹

On Stack Overflow, there are more than 19 million questions. Since we want to obtain insight about the most recurrent issues Spark developers face, we add a filter showing only questions tagged as *[apache-spark]*. After the first filtering criteria, we end up with 57,494 questions, which we sort by the number of votes in descending order. We go through the first 50 pages of 15 questions per page, which is 750 questions in total. We disregard “How to ...” and “What is ...” questions such as “What is the difference between map and flatMap?”, because we are not interested in tutorial-type questions. After selecting four questions, we talk to a Spark application developer to get their feedback; they recommend adding the *java.lang.StackOverflowError* exception. We select an initial batch of questions regarding the following common exceptions:

1. *java.io.NotSerializableException*: 118 questions asked on Stack Overflow, six of which are in the top 750 questions.
2. *java.lang.OutOfMemoryError*: 150 questions asked on Stack Overflow, four of which are in the top 750 questions.
3. *java.lang.ClassNotFoundException*: 220 questions asked on Stack Overflow, three of which are in the top 750 questions.
4. *java.net.ConnectException*: 31 questions asked on Stack Overflow, two of which are in the top 750 questions.
5. *java.lang.StackOverflowError*, which is recommended by a Spark application developer as one of the hardest errors to debug.

¹<https://stackoverflow.com/>

A pilot interview with another Spark application developer reveals two important aspects about our selection:

- Some exceptions are too simple, meaning that a developer who is new to the Spark environment is likely to encounter them. For instance, the *java.lang.ClassNotFoundException* means that the application cannot find the class given.
- Each code should be accompanied by a small scenario, meaning that we should tell an implementation story to a Spark application developer to make the purpose of the code more meaningful. The variable should be explicitly named so that they fit the implementation story.

After reworking our examples, we keep the following four bugs:

1. Serialization problem, *i.e.*, *java.io.NotSerializableException*: encountered when a class cannot be serialized.
2. 64 KB JVM bytecode limit problem: encountered when a case class, which is used to model immutable data, has too many fields and excessive nesting. This situation results in exceeding the 64 KB bytecode JVM limit and throws a *java.lang.StackOverflowError* exception.
3. Data problem, *i.e.*, *java.lang.NumberFormatException*: encountered when part of the data is corrupted or changed.
4. Excess transformations problem, *i.e.*, *java.lang.StackOverflowError*: encountered when the stack runs out of memory because too many operations are queued and Spark expects an action to be performed.

3.2 Reproducing Bugs

Here, we explain the bugs we reproduce, the implementation of which can be found in Appendix B. We briefly explain the simple story behind the implementation, why a bug occurs, where it occurs in the code, and the possible solutions for it.

3.2.1 The First Bug: Task Not Serializable

Implementation Story: The application, adapted from a Stack Overflow post², is called *SalesReportApp*, and its purpose is to calculate the mean of high sales. Then, the application prints the result.

This application throws *java.io.NotSerializable* exception whenever a method of a non-serialized class is called.

Bug Definition: In Spark, classes are expected to be serializable. As such, when the workers invoke methods on a non-serializable class, then the application throws *java.io.NotSerializable*.

Possible Solutions: There are three solutions to solve this exception:

- In Scala, unlike a *class*, an *object* is serializable by default. Therefore, the first solution is to add “extends *Serializable*” after the class definition as seen in the Listing 2. Thus, the class can be serialized, and the bug can be solved.

²<https://stackoverflow.com/questions/22592811/task%2Dnot%2Dserializable%2Djava%2Dio%2Dnotserializableexception%2Dwhen%2Dcalling%2Dfunction%2Dou>

```

1  ...
2  class SalesCalculator(sales: RDD[Int]) {
3  ...
4  }
5

```

Listing 1: Old implementation of SalesCalculator.scala

```

1  ...
2  class SalesCalculator(sales: RDD[Int]) extends Serializable {
3  ...
4  }
5

```

Listing 2: New implementation of SalesCalculator.scala

- In Scala, a method is a part of a class. A function, on the other hand, is a complete object. Thus, changing the problematic method to a function solves this problem. The change can be observed in listings 3 and 4. However, this solution depends on our business solution. If it is not convenient, we should not change the definition and keep it as a method.

```

1  ...
2  def meanOfHighSales(): Double = {
3    highSales().mean()
4  }
5  ...
6

```

Listing 3: Old implementation of meanOfHighSales() in SalesCalculator.scala

```

1  ...
2  val meanOfHighSales(): Double => {
3    highSales().mean()
4  }
5  ...
6

```

Listing 4: New implementation of meanOfHighSales() in SalesCalculator.scala

- The last solution is explained by the fifth interviewee “*Spark tries to pull all the dependencies, which is also the class in this case. I can use a conversion. I copy the value to a local variable, so that it won’t pull the whole class.*” See the changes in listings 5 and 6.

```

1  ...
2  class SalesCalculator(sales: RDD[Int]) {
3    val highSalesThreshold = 75
4    def highSales(): RDD[Int] = {
5      sales.filter(entry => entry >= highSalesThreshold)
6    }
7    def meanOfHighSales(): Double = {
8      highSales().mean()
9    }
10  ...
11  }
12

```

Listing 5: Old implementation of variables in SalesCalculator.scala

```

1  ...
2  class SalesCalculator(sales: RDD[Int]) {
3      val highSalesThreshold = 75
4      def highSales(): RDD[Int] = {
5          val tmp = highSalesThreshold
6          sales.filter(entry => entry >= tmp)
7      }
8      def meanOfHighSales(): Double = {
9          highSales().mean()
10     }
11     ...
12 }
13

```

Listing 6: New implementation of variables in SalesCalculator.scala (see the difference on line 5)

3.2.2 The Second Bug: 64 KB JVM Bytecode Limit Problem

Implementation Story: The application, adapted from an issue on Apache Spark’s Jira³, is called *GradeApp*, and its purpose is to create grade reports. After reading the grades from a file on HDFS, the application converts them to a Dataset, which is a type secure data collection in Spark, groups them by report, and counts the reports.

We receive a *java.lang.StackOverflowError* exception when creating a nested structure and applying some operations on it.

Bug Definition: The *java.lang.StackOverflowError* occurs because of too many variables and nesting of constructors, which results in exceeding the 64 KB JVM bytecode limit.

Possible Solutions: There are two possible solutions to this problem. The first solution is to decrease the number of fields and the second solution is to remove the nesting. The choice between both solutions depends on our business problem and how we would like to solve it.

3.2.3 The Third Bug: Corrupted Data

Implementation Story: The application is called *FindEmployeeApp*, and the aim is to find the number of employees in a department whose ID value is given.

In this bug, we simulate data corruption that occurs when files are sent to the cluster. When the unit test is run on the local file, we do not encounter any problem, but when reading the distant file on the HDFS, the exception *java.lang.NumberFormatException* is thrown.

Bug Definition: *java.lang.NumberFormatException* occurs when the type of the expected value does not match the type of the current value. In this example code, a string value is found when an integer value was expected.

Possible Solutions: Once the corrupted data is found, it can be either removed, changed or discarded. This choice depends on the application developers’ access, authority on the data, and the business logic.

3.2.4 The Fourth Bug: Excess Transformations Problem

Implementation Story: The application, adapted from an issue on Apache Spark’s Jira⁴, is called *DanceChoreography*. We read a file where each line contains a pair of dancing partners. The program

³<https://issues.apache.org/jira/browse/SPARK-22523>

⁴<https://issues.apache.org/jira/browse/SPARK-5499?jql=project%20%3D%20SPARK%20AND%20text%20~%20%20%20java.lang.StackOverflowError>

simulates a dance choreography where a woman and a man switch places in every iteration of the loop. For instance, if a pair is `woman1-man1`, then on the next iteration of the loop, the pair will be `man1-woman1`.

Our second `java.lang.StackOverflowError` exception is thrown during a loop of 1000 iterations.

Bug Definition: In Spark, transformations are lazily evaluated. Lazy evaluation means that transformations are stored temporarily and not applied until an action is performed. After the action is performed, Spark applies all transformations which might fill up the Java stack memory and cause the application to throw a `java.lang.StackOverflowError`.

Possible Solutions: This exception can be solved in three different ways. The first of them is to increase the stack memory so that it will not fill up. Secondly, it is possible to avoid this problem by regularly caching the transformation results in the in-memory cache. Lastly, once in a while, we can use a checkpoint to store an intermediate version of the RDD to the HDFS before applying subsequent transformations.

3.3 Developer Interviews

After finishing the implementation, we prepare simple data files for each application and copy them to the cluster. Also, we use Simple Build Tool (SBT) to create Java ARchive (JAR) files of each application and upload them to the cluster and the IntelliJ IDE in order to show the code to the interviewees.⁵ We then schedule interviews with seven professionals who are knowledgeable Spark application developers.

Before every interview, we explain to the participants the goal and steps of our research. With their consent, both audio and screen are recorded for the duration of the interview. The interviews are semi-structured, which means we are open to deviation from the interview plan. There are three people involved in the interview: an interviewee, an interviewer, and an observer. An observer should follow the interview, but can also ask and discuss questions.

The first part of the interview consists of a questionnaire about the interviewee's professional experience, which is explained in Section 3.3.1. Then, we run the sample applications one at a time. Finally, we conduct a post-mortem survey, which is explained in Section 5.1.

3.3.1 Professional Information Questionnaire

The first four questions are aimed at gathering information about our interviewees, while the remaining two relate to their experience while working on Spark applications:

1. How long is your professional experience in industry?
2. How long is your experience in Scala?
3. How long is your experience in Spark?
4. How many “different” projects that use Spark did you work on in your experience?
5. When you encounter a Spark job failure, what are your first steps to investigate?

We want to learn what our interviewees do right after they learn that their application has failed and how they search for the problem.

6. What are your tools for debugging failed Spark jobs?

If they know there is a problem, how do they find it? Do they use any software applications?

⁵<https://www.scala-sbt.org/>

In Figure 3.1, we see that the interviewees have at least three years of experience and are notably well-versed in industry, Scala, and Spark. Also, they use Spark to develop applications in various major projects.

	I1	I2	I3	I4	I5	I6	I7
How long is your professional experience in Industry?	3+	3+	3+	3+	1-3	3+	3+
How long is your experience in Scala?	3+	1-3	<1	3+	3+	3+	3+
How long is your experience in Spark?	3+	3+	<1	3+	1-3	3+	1-3
How many "different" projects that use Spark did you work on in your experience?	1	3+	2	3	3	3	3+

Figure 3.1: Experience of our interviewees in industry, Scala, and Spark; number of “different” projects that the interviewees worked on. Each column represents an interviewee. Each colored cell represents the number of years or, in the case of the last row, the number of projects (red: one or less; orange: between one and three; yellow: three; green three and above).

4

Discovering Strategies

In this chapter, we present the results of the interviews and address the first question (*RQ1: How do developers debug Spark applications?*) introduced in Chapter 1. The findings discussed in this chapter are based on our analysis of the collected data and notes during our interviews with Spark developers.

4.1 Correctness and Speed

We calculate how long each developer takes to fix each bug using the length of the recordings as in Figure 4.1. The durations vary from 2 minutes 26 seconds to 20 minutes 23 seconds. On average, interviewees need 8 minutes 12 seconds to complete each challenge, which is quite reasonable in real-world applications. We do not interfere with the interviewees as they work on the problems; however, we help them when they need it.

	I1	I2	I3	I4	I5	I6	I7	AVG
B1	07:27	04:45	15:48	11:42	07:02	03:52	12:17	08:59
B2	09:55	07:45	10:00	11:56	12:00	06:17	20:23	11:11
B3	03:11	03:10	02:50	06:21	04:15	06:46	02:56	04:13
B4	14:43	02:26	08:45	12:13	05:52	05:45	09:05	08:24
AVG	08:49	04:31	09:21	10:33	07:17	05:40	11:10	08:12

Figure 4.1: Durations to solve bugs. Each row and column represent bugs and interviewees respectively. The color code refers to cases where we explain the problem and its possible solutions (red) and those where developers figure it out on their own (green). Each cell represents the duration of an interviewee to solve a bug. In addition, we show average duration for each interviewee and for each bug as well as the global average.

We introduce the notion of *step*, *i.e.*, an action taken by the developer to solve a bug, and that belongs to a specific category. For instance, when a developer runs a test before making any changes, sees an

exception, makes changes in his code, then runs the test again with the new changes, we consider this sequence to comprise four steps. In Figure 4.2, we count the number of steps, 3 being the lowest and 13 the highest. We also include the average number of steps per interviewee in order to derive the global average (6.39).

	I1	I2	I3	I4	I5	I6	I7	AVG
B1	10	9	10	13	5	5	10	8.86
B2	7	6	8	8	5	4	7	6.43
B3	3	5	4	8	7	4	4	5.00
B4	5	5	6	7	4	5	5	5.29
AVG	6.25	6.25	7	9	5.25	4.5	6.5	6.39

Figure 4.2: Number of steps to solve a bug. Each row and column represent bugs and interviewees respectively. The color code in this table refers to the scale of steps, from low (dark green) to high numbers (red). Each cell contains the number of steps an interviewee takes to solve a bug. In addition, we show the average number of steps for each interviewee and each bug as well as the global average.

In Figure 4.3, we compare the number of steps with the debugging duration per bug. We find out that, in general, interviewees are faster in solving the third bug, *java.lang.NumberFormatException*, and quite slow in solving the second bug (*java.lang.StackOverflowError*). We observe that we cannot generalize the relationship between steps and duration. For instance, the seventh interviewee solves the second bug in 20 minutes 23 seconds (the highest duration) in only 7 steps, which is close to the average. On the other hand, it takes 11 minutes 42 seconds for the fourth interviewee to solve the first bug, but they do so in 13 steps (the highest number of steps).

4.2 Answering RQ1: Discovering Strategies

In this section, we address our first research question (*RQ1: How do developers debug Spark applications?*) by investigating our interviewees' strategies as they debug the sample applications. More specifically, we look for common patterns in their strategies in order to build a model, which ultimately allows us to propose a debugging workflow.

4.2.1 Patterns

Based on the debugging steps that we identify manually with the help of the recordings, we create a classification comprising of the following steps:

1. **Checks the logs on the console (via terminal or command line):** Tells us the interviewee uses the console to see if anything has gone wrong.
2. **Sees the exception message:** An exception message refers to a message generated by the Java Virtual Machine (JVM), such as, *java.lang.StackOverflowError*. The interviewee might see the exception message either on the console or on the Resource Manager.
3. **Checks the logs on the Resource Manager:** The interviewee uses the Resource Manager interface to find the exception message.
4. **Formulates a hypothesis:** Once the bug is found, the interviewee formulates a hypothesis and tries to find the cause of the problem and its possible solutions.

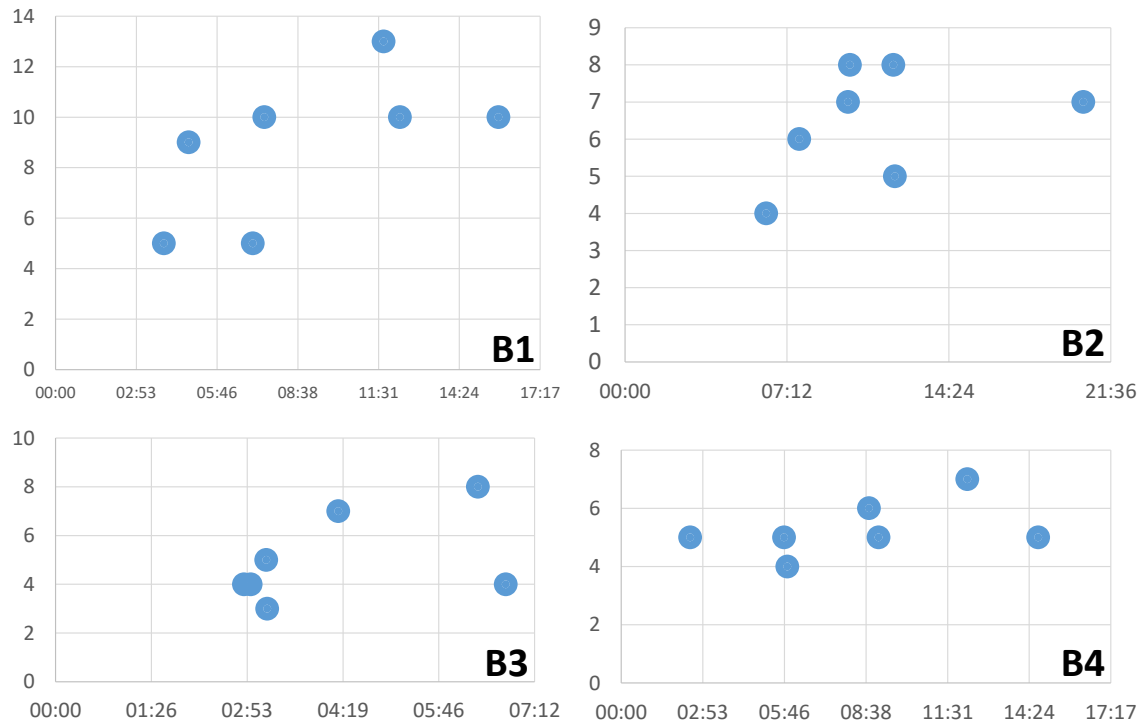


Figure 4.3: Number of steps vs. time to solve a bug (per bug)

5. **Inspects the class that threw the exception:** The interviewee finds a class name in the log file and checks the class.
6. **Suggests to do something:** After making a hypothesis, the interviewee suggests to make changes, *e.g.*, in the code.
7. **Runs the test without making any changes in the code:** The interviewee prefers to run a test (if there is any) without making any changes.
8. **Makes changes in the code:** This category refers to any kind of changes done to the code.
9. **Runs the test with the new changes:** The interviewee runs the test after having made changes in the code.
10. **Runs the code with the new changes on the cluster:** The interviewee runs the application on the cluster after having made changes in the code.
11. **Searches the exception on the internet:** The interviewee searches for possible causes on the internet if they cannot think of any.
12. **Checks the size of the input file on HDFS:** The interviewee wants to see the size of the input file on the cluster.
13. **Checks the input file on the cluster:** The interviewee takes a look at the contents of the remote input file.

14. **Inspects the deployment script:** The interviewee investigates the script that is used to upload the application to the cluster.
15. **Checks the local input file:** The interviewee takes a look at the contents of the local input file.
16. **Writes tests:** In the absence of tests, the interviewee prefers to write their own to verify whether the application runs as expected locally.
17. **Runs their own tests:** The interviewee runs the test that they have written.
18. **Changes the script:** The interviewee changes the script that is used to run the application on the cluster.

Each category is identified by a letter, ranging from A to R (see Figure 4.4). These letters and their respective colors serve as the basis for the definition of patterns.

A	Checks the logs on the console
B	Sees the exception message
C	Checks the logs on the Resource Manager
D	Formulates a hypothesis
E	Inspects the class that threw the exception
F	Suggests to do something
G	Runs the test without any changes in the code
H	Makes changes in the code
I	Runs the test with the new changes
J	Runs the code with the new changes on the cluster
K	Searches the exception on the internet
L	Checks the size of the input file on HDFS
M	Checks the input file on cluster
N	Inspects the deployment script
O	Checks the local input file
P	Writes tests
Q	Runs their own tests
R	Changes the script

Figure 4.4: The guideline for patterns

In Figure 4.5, we lay out the debugging steps followed by interviewees for each bug. By looking at the raw count for each category, we observe that some steps, such as checking the logs for exception messages (A, B, C), modifying the code (H) or running tests (G, I), are more common than others. This is not surprising because they are fundamental to application development and debugging. Another recurring step is interviewees reporting that they suspect something (D); this is explained by the fact that we ask them to describe their thought process out loud during the interviews.

While looking for patterns in the developers' initial approach to tackling a problem, we notice that their first intention is to find the exception (B) message in the logs. The way they do this varies from one developer to another. Some are quite determined to always check the console first (A) and then the Resource Manager (C), while others do the opposite or check only one source of information. The choice between one tool or the other mostly depends on the length of the log file (it is easier to read long logs on the Resource Manager).

What they do next depends on the nature of the problem. In the case of the first bug, the class name is explicitly given in the logs. This leads interviewees to check the line where the exception is thrown. In the third bug, which is data-related, the logs explicitly indicate which data is corrupted. This results in developers trying to locate it both in the local and remote files. For the second and last bugs, a *java.lang.StackOverflow* exception is raised. This means no clear course of action is prompted by the logs. This is highlighted by the fact that developers make several hypotheses (D) as to what the root of the problem could be.

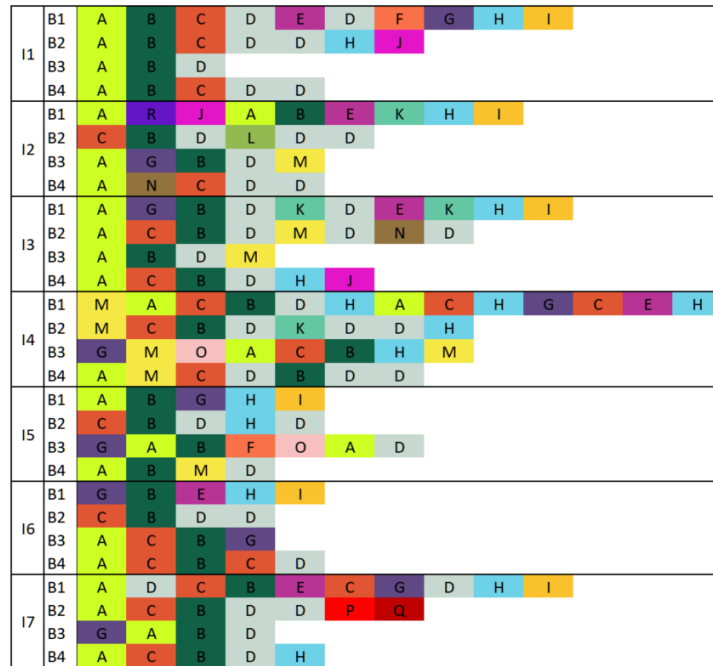


Figure 4.5: Debugging patterns grouped by each interviewee

In contrast to our previous findings, we observe that some steps are done only by a few interviewees most likely because of personal preferences and debugging style. For instance, the last interviewee writes their own test (P) and runs it (Q) because no tests are provided for that bug.

We are surprised to see that some operations are not more frequent. One example concerns checking the file size (L). Since file size can cause the problems encountered in the sample applications related to stack size, we would expect more developers to undertake this step. In addition to this, stack size can be modified by editing the Spark submit script (R); it is therefore surprising not to see more developers try this solution.

4.2.2 Hidden Markov Model

A Markov Model represents a stochastic process where each state depends only on the previous state. A Hidden Markov Model (HMM) is a Markov model with probabilities of sequences of observable events. However, some of the events might be hidden and thus cannot be observed directly. HMMs can capture hidden information in a model using observable information. HMMs are used in many fields such as computational biology, recommender systems, and handwriting. Although there are observable states, such as reading the logs, making changes in the code and running tests, debugging is a dynamic process

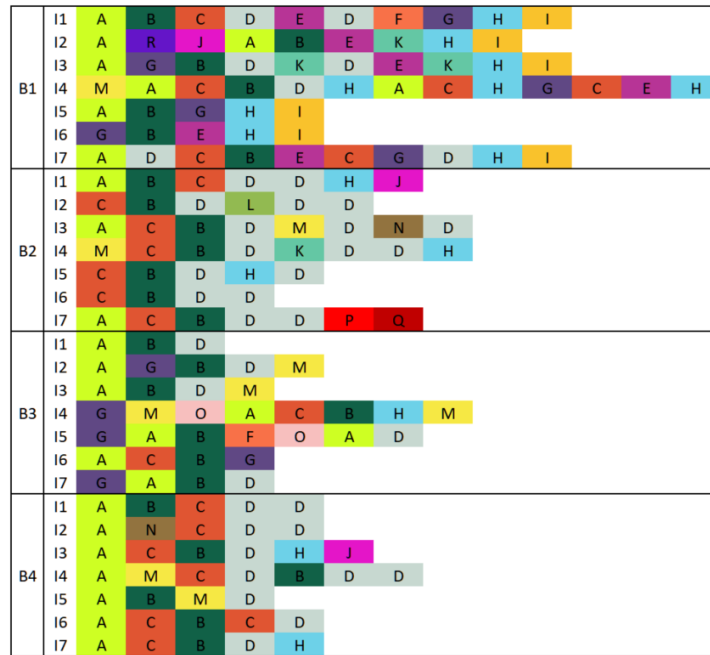


Figure 4.6: Debugging patterns grouped by each bug

and unpredictable events may occur. Thus, we believe that HMMs are a suitable way of presenting how our interviewees debug thanks to their descriptive strength.

START-A	0.68	B-D	0.56	D-C	0.04	E-H	0.33	H-G	0.08	N-C	0.50
START-C	0.11	B-E	0.11	D-D	0.29	E-K	0.33	H-I	0.50	N-D	0.50
START-G	0.14	B-F	0.04	D-E	0.07	F-G	0.50	H-J	0.17	O-A	1.00
START-M	0.07	B-G	0.07	D-F	0.04	F-O	0.50	H-M	0.08	P-Q	1.00
A-B	0.38	B-H	0.04	D-H	0.25	G-A	0.20	J-A	1.00	R-J	1.00
A-C	0.35	B-M	0.04	D-K	0.07	G-B	0.30	K-D	0.50	D-END	0.43
A-D	0.08	C-B	0.59	D-L	0.04	G-C	0.10	K-H	0.50	G-END	0.04
A-G	0.08	C-D	0.27	D-M	0.11	G-D	0.10	L-D	1.00	H-END	0.11
A-M	0.04	C-E	0.05	D-N	0.04	G-H	0.2	M-A	0.17	I-END	0.21
A-N	0.04	C-G	0.05	D-P	0.04	G-M	0.10	M-C	0.33	J-END	0.07
A-R	0.04	C-H	0.05	E-C	0.17	H-A	0.08	M-D	0.33	M-END	0.11
B-C	0.15	D-B	0.04	E-D	0.17	H-D	0.08	M-O	0.17	Q-END	0.04

Table 4.1: Frequency table. Pairs are on the left-side whereas their probabilities are on the right-side.

Every developer has their own way of solving problems. Sometimes, they perform an operation which is not preferred by others. In our HMM, we see that some pairs of steps have a probability of 1 (J-A, L-D, P-Q and R-J). However, since they occur only once or twice, we cannot infer any general debugging behavior from these observations.

Often, however, developers follow similar approaches to solve their problems. Here, we investigate the next most frequent pairs, namely START-A (0.68), C-B (0.59), and B-D (0.56). The only source of information provided by Spark is the logs. Therefore, we expect to see the interviewees choose to start their

debugging by checking the logs on the console (START-A), which is the case in 19 out of 28 debugging cases. Another way of locating the exception message is to check the Resource Manager, which explains the relatively high probability of the C-B pair. Lastly, the B-D pair indicates that interviewees start making hypotheses as soon as they have acknowledged the exception message.

Next, we observe that interviewees use the following pairs of operations with a 0.50 probability:

- The H-I pair (running the test cases after making changes) is one of the most common ways to see if one's code works or not.
- The K-D and K-H pairs (after checking an online source, a developer either comes up with a possible solution or gains knowledge about the problem and makes a hypothesis). This finding is not interesting to us because they are the only two possible outcomes after looking for information on the internet.
- The other three pairs are used to solve specific problems:
 - The F-G pair is useful if the problem is related to the code;
 - The N-D pair can be used for configuration-related issues (*e.g.*, stack memory size)
 - The F-O pair can be seen when fixing data-related bugs.

Lastly, we notice in our HMM that the D-D pair is also frequent because developers tend to make several hypotheses rather than just one. Although we only look for pairs using HMM, we should also highlight the D-x-D triad, where x can be any step except D. It shows that the developer makes changes to their initial hypothesis after having tried an operation. For instance, the D-H-D triad tells us that the changes done by the interviewee were unsuccessful.

4.2.3 Debugging Flow

Using the results from sections 4.2.1 and 4.2.2, we generalize the Markov model to a debugging flow, which presents one possible approach to debugging Spark applications. This overview can be especially helpful for new developers or for those who are interested in improving debuggers. The complete graph can be found in Appendix C. In this section, we focus on each part of the workflow and explain them.

Finding the Root of the Problem

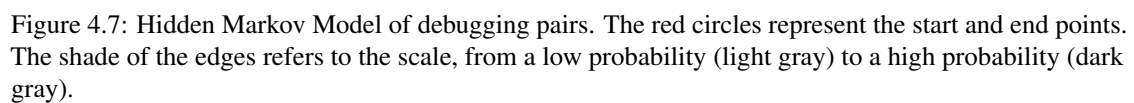
The debugging adventure starts once the developer learns the job has failed. Then, the developer examines the logs to find the exception message. In Figure 4.8, we present the beginning of debugging. We introduce two subparts, learning the job has failed and checking the logs.

Learning the Job Has Failed: There are many ways to be informed about the status of jobs. Most interviewees mention that they have a “monitoring system,” such as Grafana and Prometheus.^{1 2} In such systems, a notification is sent to the developer whenever something happens, whether it is a piece of bad news or good news from the job. Some of the interviewees also report using some “orchestration frameworks,” such as Apache Airflow, which displays scheduled jobs and their DAGs, as well as their status.³ In addition to these methods, most of them also said they have “scripts” that they write to check their jobs. Others still prefer to check the job manually by inspecting the logs. The most unusual type of the manual inspection we encounter comes from the third interviewee, who said: *“I usually know how much time I would expect for each cell to run in Jupyter. So, I have a kind of gut feeling.”*

¹<https://prometheus.io/>

²<https://grafana.com/>

³<https://airflow.apache.org/>



Checking the Logs: Once the developer learns the job has failed, they move on to the next step, which is checking the logs and tracing the exception message. In Spark, there are three possible ways to find an exception message. In reality, developers prefer to use tools, *e.g.*, schedulers, which also allows them to take a look at the logs. During our interviews, most of the interviewees prefer to use the console as their first step. However, we observe that if the logs are too long, they move on to the Resource Manager instead. Some of them use search functions on a browser to find the exception message more easily. This is especially useful when dealing with lengthy log files. However, if the log file is too long, the browser might crash.

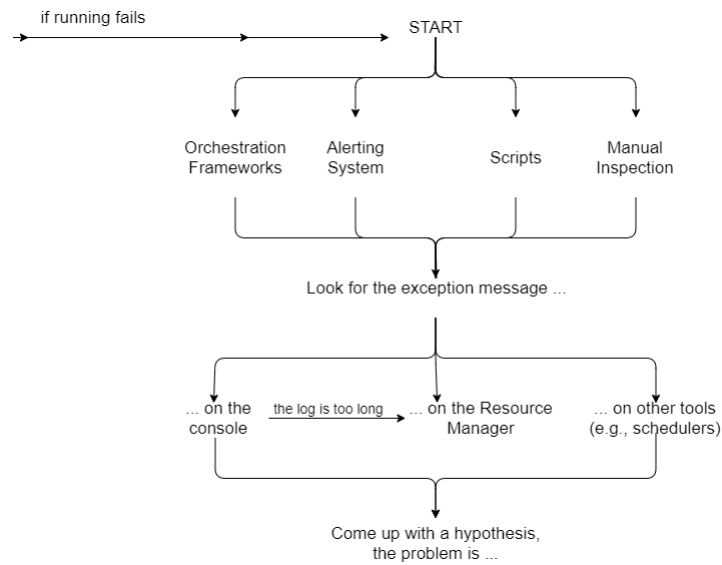


Figure 4.8: Finding the Root of the Problem

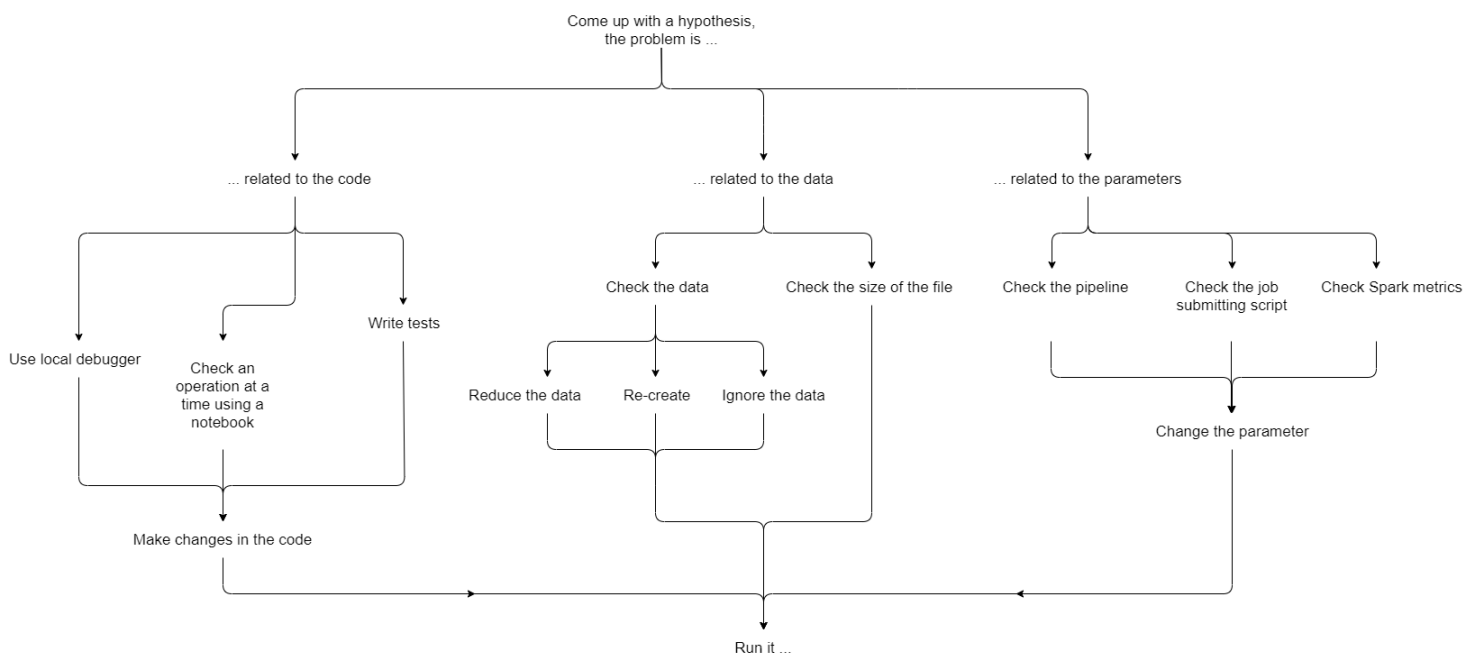


Figure 4.9: Solving the problem

Solving the Problem

In Figure 4.9, we present the flow of solving the problem once a developer comes up with a hypothesis. The problem can be related to one or several of three main categories, namely the code, the data or the configurations used to submit a job. It is not always clear to which category the problem belongs. This

means that developers often come back to this part of the workflow.

Solving Code-Related Problems: If the problem is related to the code, developers prefer to use a local debugger. However, it is not always convenient to use a local debugger since the problem might be particular to the cluster. The second way is to write test cases to see where the application fails, but again testing might not be enough to solve a problem. The second interviewee says: “... *we don’t really have test cases. Except for common functionalities...*” The last way to debug an application, recommended by some interviewees, is to use a notebook (e.g., Jupyter). A notebook consists of cells of code. One can write each line of code to a separate cell then run the notebook one cell at a time to find the problematic line. The second interviewee refers to notebooks as an “*interactive*” way to detect a problem. However, although this approach can help developers to see which line is failing, the root of the problem might be deeper. One creative way of debugging code that we observed in one of the interviewees is to use a binary approach. They check the line in the middle of the code, and if it works, they move to the middle line of the second part and so on. Whatever the approach, developers finally make all necessary changes to the code and re-run the application.

Solving Data-Related Problems: When the problem is related to the data, we observe that our interviewees follow one of two paths. The first one is to check the size of the data file. The second is to check the data itself. For instance, developers use a Jupyter notebook to observe each stage of data transformation. If they believe there is a problem, then they either reduce, re-create or ignore the data. These solutions are also based on the developer’s permissions; they might not have the authorizations to access or make changes to the data. After they make the changes, they re-run the application.

Solving Parameter-Related Problems: If the problem is related to the parameters, we observe that there are three possible ways to find the problem. First, the developer can check the pipeline to see where the problem has occurred. Secondly, one can examine the Spark metrics that contain the properties of the Spark environment, such as driver heap usage, driver Java Virtual Machine (JVM) memory pools, and executor HDFS reads. Lastly, we can investigate the script which is used to submit the job. Usually, the script consists of different parameters such as the number of workers and the deploy mode. After the problem is found, one can change the parameters and run the application again.

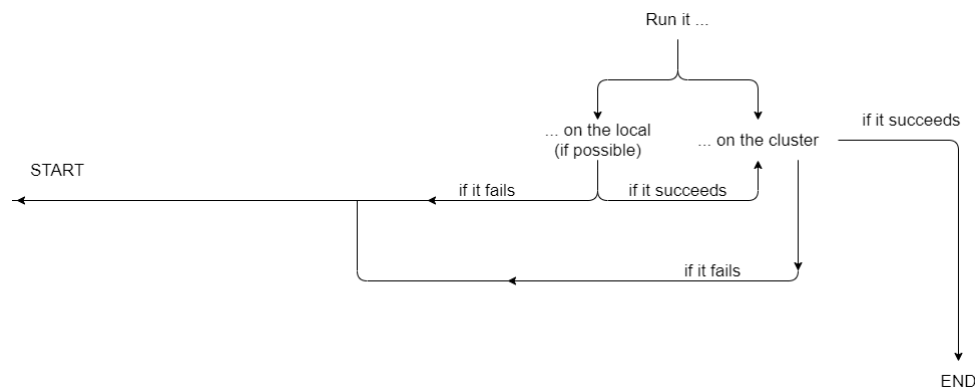


Figure 4.10: Verification

Verification (Figure 4.10): The developer can run the application again either locally (if possible) or on the cluster. If running locally works, then they re-run the application on the cluster. If the application still fails, then one must go back to the beginning of the debugging flow (see Figure 4.8).

5

Challenges and Missing Tooling

In this chapter, we present the post-mortem questionnaire that we ask after coding part of an interview and interviewee's replies. Also, we answer RQ2 and RQ3.

5.1 Post-Mortem Questionnaire

After the coding part of the interviews, we ask our interviewees some questions about their working habits in a Spark environment. The answers to these questions, except for the first one, are discussed throughout the rest of the thesis. The questions are as follows:

1. Have you seen these bugs before?
We would like to learn if the interviewees have experienced the problems we showed them during the coding part of the interviews.
2. What do you do after you see the exception message?
We want to learn the first thing they do once they find the exception message.
3. How do you know if there is a job failure?
In contrast to the previous question, this question aims to discover which tools Spark developers use to be acknowledged when an application finishes with failure.
4. What is the nastiest type of bugs you encountered in Spark applications?
We ask the developers to share the most challenging problems that they had while working on a Spark project.
5. Would you describe a tool that you miss when debugging Spark applications? What does it do? Why do you think it is not there?
We ask the developers to describe a tool that is needed to debug Spark applications and possible challenges to implement such a tool.

Have you seen these bugs before?	I1	I2	I3	I4	I5	I6	I7
B1 (Task Not Serializable)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
B2 (Stack Overflow)	No	No	No	Yes	No	No	No
B3 (Data Corruption)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
B4 (Stack Overflow)	No	No	No	No	No	No	No

Figure 5.1: Experiences of our interviewees with our experimental bugs. Each row and column represent bugs and interviewees respectively. Each cell states whether the interviewee has experienced the bug before.

It is interesting that all of our interviewees except for one report to have never seen *java.lang.StackOverflow* exceptions in a Spark environment. On the other hand, they all remarked that they had data-related problems before, e.g., *java.io.NotSerializable* in our sample bugs. This finding is not surprising because most of the work in Spark revolves around data. *java.io.NotSerializable* is expected since it is among the most popular questions on Stack Overflow.

5.2 Answering RQ2: Requirement Collection

To answer our second research question (*RQ2: What are the challenges that developers face when they debug Spark applications?*), we examine interviewees' answers to the two questionnaires as well as the notes taken during the interviews.

When asked about the “nastiest” types of bugs that they have encountered, most of our interviewees assert that Spark is not hard to deal with. In fact, Spark is difficult only when you are new to Spark. The third interviewee more specifically mentions that “... *everything in Spark took ages to figure out when I was starting with Spark ... I was totally into the blue.*” The fifth interviewee states that “... *the other components, which are used with Spark, are nastier ...*” than Spark. Yet the first interviewee tells us about a case where only half of the executors were used even though the number of partitions matched the number of executors. The sixth interviewee solved a memory overhead problem by allocating more memory to the application (e.g., 60GB per executor), but the parameters showed a memory usage of either 3KB or 6GB.

Judging from the kind of problems encountered by our interviewees, and based on our own experience, we identify four main categories of challenges that developers are faced with when debugging Spark applications. These categories are log file issues, data-related problems, remote-debugging challenges, and access to the production environment.

In Section 4.2.3, we explain that debugging starts by finding the exception message in the logs. However, most interviewees mention it is difficult to use the logs because they contain a lot of information. The sixth interviewee says “... *many output messages are completely useless, but then you also see the useful one, but they are drowning in the sea of messages ...*”. Or, as the sixth interviewee puts it, “... *the logs are too long, the browser can crash ...*”. The other problem of log files is that they lack explicit useful information, and thus can be misleading. Developers complain that, most of the time, logs are not actually difficult to deal with.

The second category refers to challenges related to data. One interviewee points out that “*most problems are coming from the real data.*” Sometimes, the root of the problem is that the data (or information about it) cannot be accessed. As the third interviewee mentions, “*especially, people who read the pipeline struggle in our teams, they don't really see the intermediate steps ...*”. Spark does not show developers how the data looks after each transformation, which makes it difficult to catch the problem either in the data or in the code.

The next category is to have support for remote debugging. Most of our interviewees agree that

debugging locally does not always help to solve problems because some problems only occur on the cluster. The second interviewee mentions, *“In general, it’s difficult to see what’s going on in the executor side.”* This is due to the number of devices in the cluster as well as network limitations. In the words of the first interviewee, when it comes to distributed debugging, it is *“not technically impossible to do it, it is just not physically possible to do it. There is a limit of how many machines you can connect, how much data you can use, how are you going to interact, and so on”*.

The last challenge is pointed out by the seventh interviewee, who says they are unable to access applications in the production environment. This choice to sever a developer from the environment that actually matters may be justifiable from a security and data governance standpoint, but it makes the task of debugging all the more difficult.

5.3 Answering RQ3: Missing Tools

In this section, we answer our third research question (*RQ3: How can we overcome these challenges?*) by examining interviewees’ answers to the questionnaires, and discussing possible solutions for the debugging problems used during the interviews.

Apart from the log files, interviewees mention three tools that they already use alongside Spark to help them in debugging. These tools are Apache Airflow, Prometheus, and Grafana.^{1 2 3} Apache Airflow is used for managing or scheduling the Directed Acyclic Graphs (DAG) of jobs. It allows the developer to define workflows in the code. This tool is useful to inspect the pipeline and get more insights about the submitted job. Also, in the workflow, developers can add or remove a task, which gives them flexibility. They can also write tests in Apache Airflow environment. Prometheus and Grafana, on the other hand, work on metrics like the Spark environment parameters or the job parameters. Grafana enables the analysis, understanding and visualization of metrics in a dashboard, whereas Prometheus merges and stores the metrics in the form of time series.

We classify the many solutions to the challenges faced by our interviewees into three categories, namely coding experience enhancements, log enhancements, and dashboard implementation.

Coding Experience Enhancements Even though many debuggers exist, there is no remote debugger where one can step in and out of the code to investigate its behavior. Interviewees say that they often use local debuggers; however, local debuggers are not sufficient for the detection of bugs that occur only on a cluster due to numerous reasons. For instance, if only one worker node in the cluster out of a thousand causes problems like a memory overload due to array size, it will take a significant amount of time and effort to find the problematic node. In the next chapter, we will discuss several studies that are being done to implement a remote debugger, each of them proposing a different approach and solution.

A tool, such as a plugin, might be implemented to spit out the “last-seen data,” which refers to the last successfully obtained data and its file path before corruption occurs. As the first interviewee suggests “You should have the option in the API to catch the tables”, *e.g.*, it finds the line of a CSV file that causes the problem. Unlike our simple applications, real-world data contains many faults, and data scientists who work on massive data sets struggle more to find corrupted records. Another functionality of this tool could be to display some sample data after each transformation or after a pre-defined number of transformations. Thus, developers can observe their data, its form, and the results.

Log Enhancements During the interviews, we discuss a “log parser” to improve Spark logs. First of all, the parser should be developed using machine learning algorithms using a cluster of logs that corresponds

¹<https://airflow.apache.org/>

²<https://prometheus.io/>

³<https://grafana.com/>

to a specific problem. Then, using the bug patterns learned, the parser should offer solutions based on the exception message. The most challenging part of this approach is to create a beneficial recommendation system that requires hundreds of thousands of logs, since it would take a considerable amount of time for a team to train a cluster. The parser can also parse essential and useless parts of logs and highlight unexpected lines. For instance, when we check the logs on the Resource Manager, we see that there are many lines of messages, however, only few of them are exception messages. Therefore, it is tricky to locate the exception message in the logs. Sometimes, developers use their browser's search function to trace the exception message, but if the log is too long, the browser cannot load it and crashes.

In Spark, exception messages come directly from the JVM and often do not provide enough information about the root of the problem. We believe that Spark bears some responsibilities when the cause is related to Spark. For instance, as mentioned in Section 3.2.4, applying an action after numerous transformations causes a *java.lang.StackOverflowError* exception. Such exceptions are often not clear since the reasons could be manifold. Also, as the second interviewee mentions: *"In Java, you can usually see in which line the code dies, but with Spark, you just have to guess."* Along the same line, another interviewee proposes to have *"a flag that enables logging into certain parts of Spark so that you don't have to wonder which classes you need to check."* The tool that both developers are outlining could track down the classes visited and print a warning when an exception occurs.

The last recommendation is to print log entries after each operation so that each part can be examined instead of the whole application. This solution would aid developers in understanding each part of their code better as well as making it easier to find exception messages.

Dashboard Implementation Some developers work on their client's data and support them with data analytics. They say that they lack the proper authorization to access the production environment and have to work on a simulated environment, which causes problems if the properties of the environments are different. To remediate this, they suggest a dashboard that can compare their environment and their clients' environment, *i.e.*, resources of both environments like memory. Similarly, the sixth interviewee recommends a tool that would display all the configurations of both the cluster and the local environment and compare the environments with the resources used by the application.

6

Related Work

Every debugging tool proposes a different way of helping developers. In this section, we introduce six different approaches, some of which are inspired by the work of Beschastnikh [6] *et al.* and Cheung *et al.* [7], to debug applications: log analysis, model checking, record and replay, predicate testing, tracing, and visualization. We explain each category briefly and introduce some work done in each of them, the results of their respective experiments, and their advantages and drawbacks. Before we go further, we want to highlight that the Theia debugger was especially introduced for the Hadoop environment, which we choose in the scope of this review.

Log Analysis Spark produces logs for every application. Using logs, a developer can reason about many things like the root of the problem, the used resources, *e.g.*, memory, and the run time of the application [6]. Log analysis can parse lengthy logs to ease navigation for developers. There are several log analysis tools such as D3S [16], ShiViz [6], MACEDB [11], Aguilera *et al.* [3], Pip [21], Theia [9], and Recon [13]. In MACEDB, Killian *et al.* filter the logs using regular expressions [11]. In Theia, Garduno *et al.* and in Recon, Lee and Eugster introduce a way to browse and analyze logs [9, 13]. In these works, applications separate important messages in logs or analyze them, which corresponds to some of our interviewees' suggestions.

Model Checking A model checking approach requires producing the model of a system, *e.g.*, a finite-state model, and verifying whether it supports certain properties, *e.g.*, liveness, [6]. There are many model checking tools proposed by researchers such as CMC [19], MACEDB [11], and MODIST [24]. Model checking is not suggested by our interviewees; however, it is useful to analyze code before running it. Model checking requires to write the model, which might be more challenging than writing the code itself.

Record and Replay Record and Replay tools capture and store the execution of a system [6, 7]. Using the recorded capture, a developer can simulate the execution anytime and control it. Friday [10], MACEDB [11], MaceODB [8], Recon [13], and WiDS [17] are examples of record and replay tools proposed in the literature. To the best of our knowledge, our interviewees neither use any record and replay tools nor do they suggest them when asked about any tool that might be useful to them. The Record and Replay

approach is challenging when the number of recordings is either too small or too big. In addition, building a time machine to replay and managing the records might raise issues.

Predicate Testing Testing ensures that a system works as expected using conditions, which are the boolean results of functions or boolean statements known as predicates, *e.g.*, Assert of JUnit [16, 17].¹ Tests can catch exceptions and other faults in a system. These tools use predicate checking to test some properties to catch violations, which is, as our interviewees suggest, a way to test the system. Research works that implement predicate testing tools are: D3S [16], Friday [10], MaceODB [8], and WiDS [17].

Tracing Tracing means tracking the flow of code or data in an application. Using tracing, developers can examine, alter, and control software applications [1, 6, 7, 23]. Tracing tools are also known as step-through debuggers, which allow developers to step in and out of code blocks. The tracing approach is very similar to what interviewees suggest, namely a remote debugger. Although the tools we find are not exactly like local step-through debuggers, a developer can still control the execution. Research works that implement tracing debuggers are: Pharo Promise Debugger [14], EXDAMS [4], GoTcha [1], and P2 [22].

Visualization Visualization tools capture the current resources of a system and display their performance [7]. For instance, using the user interface, a developer can track how much RAM an application uses. ShiViz [6], MACEDB [11], and Theia [9] are examples of visualization tools.

Tracing Tools Leske *et al.* propose a debugger interface that can be used to investigate the call stack when an exception rises [14]. The debugger interface fuses call stacks of two processes, one of which is remote. Their implementation aims to solve debugging problems of remote processes as well as promises. They chose the Pharo language because it is a reflective language and can manipulate processes directly. They encountered two supplementary costs, which are performance and memory overhead. The authors state that generally these costs do not decrease performance.

Balzer develops EXDAMS, which traces the events of execution [4]. EXDAMS fetches and displays information from a history tape, which contains required information of the executions and obtained by monitoring the program. The EXDAMS system has static and motion-picture debugging and monitoring help types. Unlike motion-picture help, static help presents information that does not vary with each execution. For instance, if a piece of information is displayed using motion-picture help, then the information may be different when executed in a different time. Balzer states that EXDAMS is not efficient because replaying records from a history tape requires many I/O operations.

Achar *et al.* implement a browser-based interactive debugger, namely GoTcha [1]. As the authors state, GoTcha is designed for Global Object Tracker programming models, which synchronize object states across distributed nodes in a similar way to Git, a distributed version controlling system. Using GoTcha, a developer can observe state changes and control the flow of execution. GoTcha is tested in a distributed environment on a simple word count application, which outputs wrong count values. GoTcha succeeds when the developer observes the version history and sees the mistake. Achar *et al.* state that GoTcha still needs to improve in terms of scalability of the user interface and the debugger.

Model Checking Tools Musuvathi *et al.* implement a model checker for C and C++ [19]. The model checker performs tests to ensure that an application will work as expected and detect bugs in the implementation. It uses a hash table to record states. Using C Model Checking (CMC), 34 unique bugs out of 40 bugs are detected. Such bugs include memory leaks, assertion failures, and routing loops. Musuvathi *et al.* find that most of the bugs are discovered in a few minutes.

¹<https://junit.org/junit4/javadoc/4.12/org/junit/Assert.html>

Name of the tool	Log Analysis	Model Checking	Record and Replay	Predicate Testing	Tracing	Visualization
Pharo Promise [14] Debugger					X	
EXDAMS [4]					X	
GoTcha [1]					X	
Aguilera et al. [3]	X				X	
Pip [21]	X					
ShiViz [6]	X					X
Theia [9]	X					X
MACEDB [11]	X	X	X			X
D3S [16]	X			X		
Recon [13]	X		X			
Friday [10]			X	X		
WiDS [17]			X	X		
MaceODB [8]			X	X		
MODIST [24]		X				
CMC [19]		X				

Table 6.1: This table categorizes previous works in debugging distributed systems into six main categories. Every line represents a distributed debugger.

Yang *et al.* design a model checker, called MODIST for unmodified distributed systems [24]. MODIST simulates executions and allows state exploration. MODIST is evaluated in three distributed environments, and many bugs are found. When the performance of the system is tested, Yang *et al.* find that even in the worst case, the overhead is still reasonable.

Log Analysis Tool Reynolds *et al.* introduce Pip, which traces performance bugs and unexpected behaviors [21]. Pip provides logging and visualization. Behaviors are represented as a list of path instances and written explicitly by developers. Pip is tested on four different systems, and many bugs are found. Most of the bugs are related to performance, *e.g.*, read latency, whereas other bugs are related to correctness.

Log Analysis and Predicate Testing Tool Liu *et al.* develop a tool for predicate checking, called D3S ([16]). A developer must declare the properties they want to check, and D3S checks the system during the run time. Manually declaring states is a drawback of the system. A developer should order the states in a DAG-form. D3S might have performance overhead due to exposing states, *e.g.*, local states. Liu *et al.* find that the overhead is at the maximum (7-8%) when there are two threads, and exposing frequency is 1000, packet size is 512 or 1024 bytes.

Log Analysis and Visualization Tools Beschastnikh *et al.* introduce ShiViz, a visualization tool to show distributed system executions interactively [6]. ShiVector introduces timestamps that are utilized to construct happens-before relations of events. ShiViz parses logs with ShiVector and displays it to a developer in a logical order not real time.

Garduno *et al.* present Theia, a visualization and log analysis tools especially developed for Hadoop [9]. They introduce the notion of visual signatures that present patterns of unexpected behaviors to detect application-level, infrastructural, and workload problems. Theia performs well when tested on 1373 Hadoop jobs. It finds 192 out of 204 problems; 157 application-level, 2 workload, and 45 infrastructural. Garduno *et al.* state that Theia is not able to identify problems if the cluster is idle or if a node is blacklisted.

Record and Replay and Predicate Testing Tools Geels *et al.* introduce Friday, global predicate monitoring, and a distributed replay system that helps developers to observe and understand states of an application [10]. Friday can capture and replay the execution using a symbolic debugger. Geels *et al.* state that Friday has a few limitations, such as slowing down the replay, starting from a middle point takes time because it requires execution to that point, and predicates require debugging. Geels *et al.* experiment with Friday and find that latency is higher for function calls than other watchpoint events, *i.e.*, writing value. They find their tool scales well as the number and percentage of watchpoints increase.

Dao *et al.* introduce their tool, MaceODB, to aid developers to debug distributed systems [8]. Mace is a language extension of C++, and MaceODB inherits Mace's properties and adds new properties, which are liveness and safety. The tool reports violations to the properties upon execution. The authors describe a debugger that is easy to use, powerful and has low overhead as the epitome of distributed debuggers. Their results show that MaceOBD's performance is lightweight on a decentralized topology, but not on centralized topology. They were able to detect non-trivial errors using their methods.

Liu *et al.* introduce their predicate-based predicate checker, namely WiDS [17]. WiDS gathers logs of execution and deterministically replays them. WiDS is tested on various systems, and different kinds of bugs are found, such as deadlocks and livelocks. The performance of log collection is evaluated, and an overhead of 2% is observed. The performance of the predicate decreases by 4 to 20 times when the checker is used. However, Liu *et al.* state that this inefficiency is acceptable.

Log Analysis and Tracing Tool Aguilera *et al.* use message-level traces for performance debugging of a system that can be used to separate performance issues to increase efficiency [3]. They achieve it

by gathering traces of nodes and combining them. First, they experiment on millions of synthetically generated traces using their trace generator. Then, they experiment on real-world applications. They add delays to their messages to simulate network delays and test their algorithms, a nesting algorithm to inspect traces, and a convolution algorithm to find causal relationships. They find that increasing parallelism and delay variation decrease the performance of the system. Aguilera *et al.* test the accuracy of the convolutional algorithm and find that it can detect frequent paths.

Log Analysis and Record and Replay Tool Lee *et al.* develop Recon, a debugging method that uses logs and deterministic replay [13]. Recon does not use all logs; instead, it uses replay logs of each node. Then, Recon debugs by replaying the logs. It is tested on four different distributed applications, and the authors find that, except for one application, logging and replay overheads are reasonable.

Log Analysis, Model Checking, Record and Replay and Visualization Tool Killian *et al.* present MACEMC, a model checker to detect liveness breaches, and MDB, a debugging tool [11]. MACEMC uses MDB, which has various support for interactive execution, replay, log analysis, and visualization. MACEMC is tested on four different systems, and 52 bugs, 31 liveness issues, and 21 safety issues are found. Killian *et al.* state that using MACEMC and MDB perform better than existing approaches.

7

Conclusion and Future Work

In this section, we conclude our findings and present possible ideas for future work and implementations.

Conclusions

In this study, we examine how Spark developers debug their applications. We set up a Spark cluster, search for the most common Spark exceptions and replicate them. We then hold a series of interviews with seven Spark application developers, which are recorded with their consent.

Using the results of interviews, we observe the interviewees' debugging strategies. We find that some steps are more common than others; such steps are checking the logs for exception messages, modifying the code, and running tests. We use Hidden Markov Models to find recurring debugging pairs, and we find that interviewees most commonly start the debugging process by checking the logs on the console and the Resource Manager to find the exception message, then make a hypothesis right afterward. Then, we model a debugging flow consisting of two main parts; finding the root of the problem and solving the problem. We find that once a developer comes up with a hypothesis about the problem, they perform different operations depending on the cause of the problem (*e.g.*, they use a local debugger if the problem is related to their code). Tracking exception messages is done by checking logs either on the Resource Manager or on other tools. Although there are many distributed debugging tools (see Chapter 6), we see that most of the interviewees do not use such tools. Instead, they solely rely on logs or non-automated tests to solve the problems (*e.g.*, using a notebook to check the code line by line). When they do use external tools, it is usually to receive notifications from submitted jobs, to schedule jobs, to display some properties or the system status.

Future Work

During the interviews, we also ask the interviewees some open-ended questions. The resulting answers point out the problems Spark developers have and indicate suggestions to improving coding experience, improving logs, and implementing a dashboard. We discuss challenges and missing tools in Chapter 5 in detail.

Improving Coding Experience Interviewees state that there is no remote debugger that works like local debuggers. Remote debuggers are difficult to implement and to use due to many reasons, such as the number of nodes in an environment, network problems, and the size limit of data. In addition, interviewees suggest that a plugin can be implemented to display some sample data after each operation. This plugin would also display the last successfully obtained data, *e.g.*, a row in a CSV file.

Improving Logs Interviewees suggest a log analyzer that prints logs after each operation and uses machine learning algorithms to learn problems and suggest appropriate solutions. It is challenging to implement such a system because it requires many logs to train an algorithm. Another type of log tool might give more explicit messages about the exceptions instead of the basic JVM messages, which can sometimes be laconic. This, in turn, would help to separate essential from useless messages.

Dashboard The last suggestion is to design a dashboard that can display an environment's properties and compare them with those of other environments. It is important for developers who work in different environments to be able to compare their properties.



Setting Up an Apache Spark Cluster

A.1 Upgrading Operating Systems

Debian OS

1. Check your current version of Debian OS by typing the command below.

```
cat /etc/debian_version
```

2. Check if the system is up-to-date.

```
sudo apt-get update
```

3. Apply changes if there are any.

```
sudo apt-get upgrade
```

4. Change the `jessie` (the name of the Debian 8) to `stretch` (the name of the Debian 9).

```
sed -i 's/jessie/stretch/g' /etc/apt/sources.list
```

5. Repeat Step 2 and 3, so that the system can find new changes and apply them.
6. The command below will update all the installed packages.

```
sudo apt-get dist-upgrade
```

7. Finally, reboot the system.

Ubuntu OS

1. Check the version of Ubuntu by typing the command below.

```
lsb_release -a
```

2. Check the updates.

```
sudo apt update
```

3. Apply changes if there are any.

```
sudo apt dist-upgrade
```

4. Start upgrading the system.

```
sudo apt install update-manager-core
```

```
do-release-upgrade
```

5. The terminal displays instructions. Follow the instructions to upgrade the OS.
6. Finally, reboot the system.

A.2 Installing Java and Scala

Install Java on both Debian and Ubuntu

1. Check if you have Java and JDK already installed, respectively.

```
java -version  
javac -version
```

2. If you do not have them, install them using the scripts below.

```
sudo apt install default-jre  
sudo apt install default-jdk
```

Install Scala on both Debian and Ubuntu

1. Check if you have Scala already installed.

```
scala -version
```

2. Download Scala package.¹

```
wget -c www.scala-lang.org/files/archive/scala-2.11.7.deb
sudo dpkg -i scala-2.11.7.deb
```

3. Check if there are any updates.

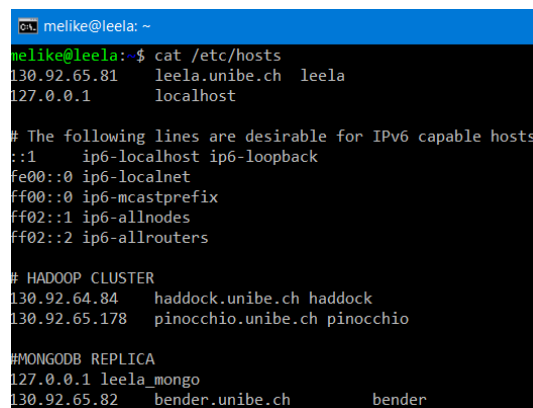
```
sudo apt-get update
```

4. Install Scala.

```
sudo apt-get install scala
```

A.3 Installing and Setting Up an Apache Hadoop Cluster

1. Add IPs of other nodes to /etc/hosts file, which should look like in Figure A.1.



```
melike@leela: ~
melike@leela:~$ cat /etc/hosts
130.92.65.81    leela.unibe.ch  leela
127.0.0.1      localhost

# The following lines are desirable for IPv6 capable hosts
::1           ip6-localhost ip6-loopback
fe00::0       ip6-localnet
ff00::0       ip6-mcastprefix
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters

# HADOOP CLUSTER
130.92.64.84    haddock.unibe.ch haddock
130.92.65.178  pinocchio.unibe.ch pinocchio

#MONGODB REPLICAS
127.0.0.1      leela_mongo
130.92.65.82    bender.unibe.ch  bender
```

Figure A.1: /etc/hosts file

2. The master node needs an SSH connection to connect to worker nodes. Thus, it is essential to set SSH keys.

- (a) Log in to the master node and generate an SSH key: (leave the password blank)

```
ssh-keygen {b 4096
```

- (b) Copy the master node's public key.

¹<https://www.scala-lang.org/files/archive/>

- (c) For each worker node, create a new file, `master.pub`, in the `/home/hadoop/.ssh` directory and paste the master node's public key.
- (d) Add the key to `~/.ssh/authorized_keys` file.

3. On the master node, download Apache Hadoop using the following script:

```
wget http://apache.cs.utah.edu/hadoop/common/current/
hadoop-3.1.2.tar.gz

tar -xzf hadoop-3.1.2.tar.gz

mv hadoop-3.1.2 /opt/hadoop-3.1.2
```

4. Set environment parameters for Apache Hadoop

- (a) Update PATH. Go to `/home/<user_name>/.profile` and add the following line to the file

```
PATH=/home/<user_name>/hadoop/bin:/home/<user_name>/
hadoop/sbin:$PATH
```

- (b) Update PATH for the shell as well. Go to `/home/<user_name>/.bashrc` and add the following line to the file

```
export HADOOP_HOME=/home/<user_name>/hadoop
export PATH=${PATH}:${HADOOP_HOME}/bin:${HADOOP_HOME}/sbin
```

5. NameNode location should look like as shown in Figure A.2, which is under `/opt/hadoop-3.1.2/etc/hadoop/core-site.xml` on the master node.

6. The path of HDFS should look like as shown in Figure A.3 image, which is under `/opt/hadoop-3.1.2/etc/hadoop/hdfs-site.xml` on the master node.

7. YARN settings should look like as shown in Figure A.4, which is under `/opt/hadoop-3.1.2/etc/hadoop/mapred-site.xml` on the master node.

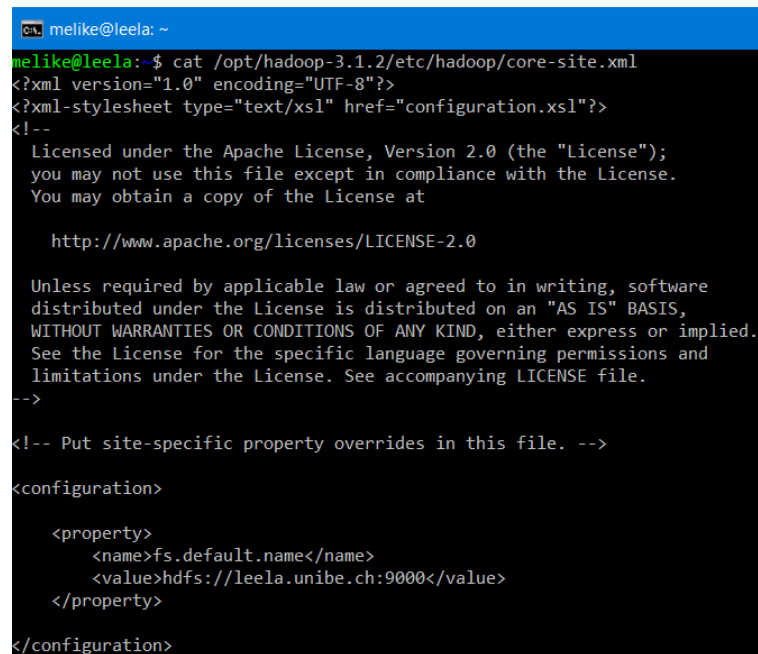
8. YARN settings should look like as shown in Figure A.5, which is under `/opt/hadoop-3.1.2/etc/hadoop/yarn-site.xml` on the master node.

9. You should add the worker nodes as shown in Figure A.6, which is under `/opt/hadoop-3.1.2/etc/hadoop/workers` on the master node.

10. Copy Apache Hadoop directory to other nodes (the workers).

11. Format HDFS on the master node using the following command.

```
hdfs namenode --format
```

A terminal window with a blue title bar showing the command prompt 'melike@leela: ~'. The user has executed 'cat /opt/hadoop-3.1.2/etc/hadoop/core-site.xml'. The output shows the XML content of the file, including the Apache License text and a configuration block for the HDFS default name.

```
melike@leela:~$ cat /opt/hadoop-3.1.2/etc/hadoop/core-site.xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>

  <property>
    <name>fs.default.name</name>
    <value>hdfs://leela.unibe.ch:9000</value>
  </property>

</configuration>
```

Figure A.2: core-site.xml file

A.4 Installing Apache Spark

1. Download Apache Spark

```
wget http://d3kbcqa49mib13.cloudfront.net/spark-2.0.2-bin-hadoop2.7.tgz
```

2. Extract the packages

```
tar -xvf spark-2.0.2-bin-hadoop2.7.tgz
```

3. Copy the packages to a folder of your choice (here, it's /opt/spark)

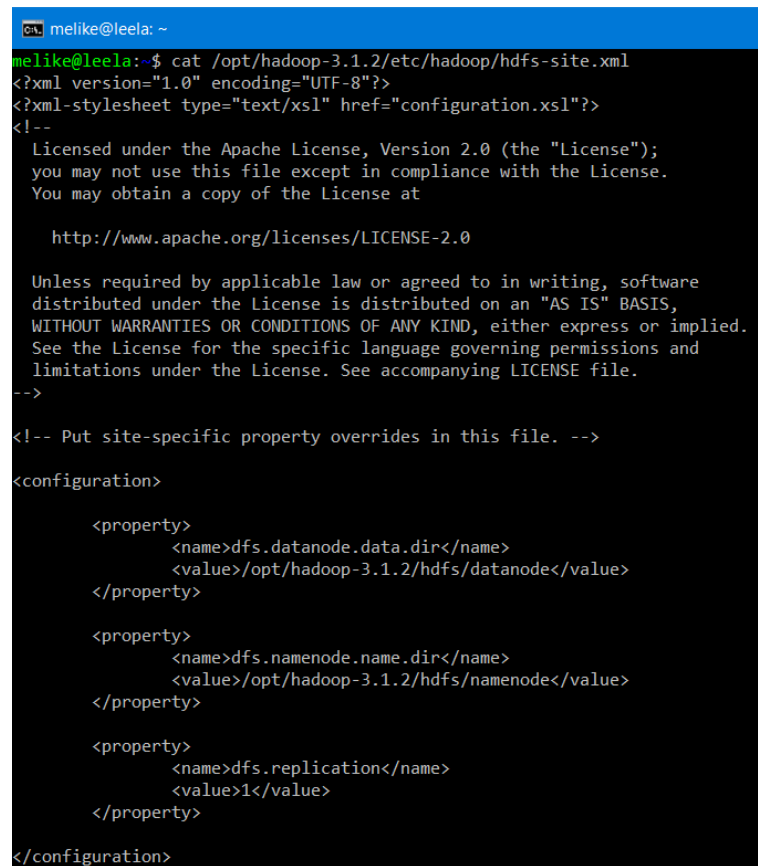
```
cp -rv spark-2.0.2-bin-hadoop2.7/* /opt/spark
```

4. Run Spark-shell to verify installation is successful. (in /opt/spark directory)

```
./bin/spark-shell -- master local[2]
```

5. Unite Apache Spark and Apache YARN Resource Manager

- (a) Set up environment variables.

A terminal window with a blue title bar showing the command to view the hdfs-site.xml file. The output is an XML configuration file for Hadoop Distributed File System (HDFS). It includes a license notice for Apache License 2.0 and a configuration section with three properties: data directory, name directory, and replication factor.

```
melike@leela: ~  
melike@leela:~$ cat /opt/hadoop-3.1.2/etc/hadoop/hdfs-site.xml  
<?xml version="1.0" encoding="UTF-8"?>  
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>  
<!--  
  Licensed under the Apache License, Version 2.0 (the "License");  
  you may not use this file except in compliance with the License.  
  You may obtain a copy of the License at  
  
    http://www.apache.org/licenses/LICENSE-2.0  
  
  Unless required by applicable law or agreed to in writing, software  
  distributed under the License is distributed on an "AS IS" BASIS,  
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
  See the License for the specific language governing permissions and  
  limitations under the License. See accompanying LICENSE file.  
-->  
  
<!-- Put site-specific property overrides in this file. -->  
  
<configuration>  
  <property>  
    <name>dfs.datanode.data.dir</name>  
    <value>/opt/hadoop-3.1.2/hdfs/datanode</value>  
  </property>  
  
  <property>  
    <name>dfs.namenode.name.dir</name>  
    <value>/opt/hadoop-3.1.2/hdfs/namenode</value>  
  </property>  
  
  <property>  
    <name>dfs.replication</name>  
    <value>1</value>  
  </property>  
</configuration>
```

Figure A.3: hdfs-site.xml file

- (b) Rename the `spark-defaults.conf.template` under `/opt/spark/conf` as `spark-defaults.conf`
- (c) Edit the file as shown in the following image.

A terminal window with a blue title bar showing the command to view the mapred-site.xml file. The output is an XML configuration file for Hadoop MapReduce, including a license notice and specific property overrides for YARN and MapReduce memory.

```
melike@leela: ~  
melike@leela:~$ cat /opt/hadoop-3.1.2/etc/hadoop/mapred-site.xml  
<?xml version="1.0"?>  
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>  
<!--  
  Licensed under the Apache License, Version 2.0 (the "License");  
  you may not use this file except in compliance with the License.  
  You may obtain a copy of the License at  
  
    http://www.apache.org/licenses/LICENSE-2.0  
  
  Unless required by applicable law or agreed to in writing, software  
  distributed under the License is distributed on an "AS IS" BASIS,  
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
  See the License for the specific language governing permissions and  
  limitations under the License. See accompanying LICENSE file.  
-->  
  
<!-- Put site-specific property overrides in this file. -->  
  
<configuration>  
  
  <property>  
    <name>mapreduce.framework.name</name>  
    <value>yarn</value>  
  </property>  
  
  <property>  
    <name>yarn.app.mapreduce.am.resource.mb</name>  
    <value>512</value>  
  </property>  
  
  <property>  
    <name>mapreduce.map.memory.mb</name>  
    <value>256</value>  
  </property>  
  
  <property>  
    <name>mapreduce.reduce.memory.mb</name>  
    <value>256</value>  
  </property>  
  
</configuration>
```

Figure A.4: mapred-site.xml file

```
melike@leela: ~  
melike@leela:~$ cat /opt/hadoop-3.1.2/etc/hadoop/yarn-site.xml  
<?xml version="1.0"?>  
<!--  
  Licensed under the Apache License, Version 2.0 (the "License");  
  you may not use this file except in compliance with the License.  
  You may obtain a copy of the License at  
  
    http://www.apache.org/licenses/LICENSE-2.0  
  
  Unless required by applicable law or agreed to in writing, software  
  distributed under the License is distributed on an "AS IS" BASIS,  
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
  See the License for the specific language governing permissions and  
  limitations under the License. See accompanying LICENSE file.  
-->  
<configuration>  
<!-- Site specific YARN configuration properties -->  
  
  <property>  
    <name>yarn.acl.enable</name>  
    <value>0</value>  
  </property>  
  
  <property>  
    <name>yarn.resourcemanager.hostname</name>  
    <value>leela.unibe.ch</value>  
  </property>  
  
  <property>  
    <name>yarn.nodemanager.aux-services</name>  
    <value>mapreduce_shuffle</value>  
  </property>  
  
  <property>  
    <name>yarn.nodemanager.resource.memory-mb</name>  
    <value>1536</value>  
  </property>  
  
  <property>  
    <name>yarn.scheduler.maximum-allocation-mb</name>  
    <value>1536</value>  
  </property>  
  
  <property>  
    <name>yarn.scheduler.minimum-allocation-mb</name>  
    <value>128</value>  
  </property>  
  
  <property>  
    <name>yarn.nodemanager.vmem-check-enabled</name>  
    <value>false</value>  
  </property>  
</configuration>
```

Figure A.5: yarn-site.xml file

```
melike@leela: ~  
melike@leela:~$ cat /opt/hadoop-3.1.2/etc/hadoop/workers  
pinocchio.unibe.ch  
bender.unibe.ch  
haddock.unibe.ch
```

Figure A.6: workers file

```
melike@leela: ~  
melike@leela:~$ cat /opt/spark/conf/spark-defaults.conf  
#  
# Licensed to the Apache Software Foundation (ASF) under one or more  
# contributor license agreements. See the NOTICE file distributed with  
# this work for additional information regarding copyright ownership.  
# The ASF licenses this file to You under the Apache License, Version 2.0  
# (the "License"); you may not use this file except in compliance with  
# the License. You may obtain a copy of the License at  
#  
# http://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.  
#  
# Default system properties included when running spark-submit.  
# This is useful for setting default environmental settings.  
# Example:  
# spark.master spark://master:7077  
# spark.eventLog.enabled true  
# spark.eventLog.dir hdfs://namenode:8021/directory  
# spark.serializer org.apache.spark.serializer.KryoSerializer  
# spark.driver.memory 5g  
# spark.executor.extraJavaOptions -XX:+PrintGCDetails -Dkey=value -Dnumbers="one two three"  
spark.master yarn  
spark.driver.memory 512m  
spark.executor.memory 512m  
# set history-server in hdfs  
spark.yarn.historyServer.address http://leela.unibe.ch:18080  
spark.history.ui.port 18080  
spark.eventLog.dir hdfs:///spark-history  
spark.eventLog.enabled true  
spark.history.fs.logDirectory hdfs:///spark-history
```

Figure A.7: spark-defaults.conf file

B

Implementation of Bugs

B.1 Implementation of the First Bug: Task Not Serializable

```
1 package ch.unibe.scg
2 object MainClass {
3     def main(args: Array[String]) {
4         if (args.length == 1) {
5             Spark.buildContext (Env.cluster)
6             val sales = SalesReader.read(args(0))    //"hdfs://leela.unibe.ch:9000//sales.txt"
7             println(new SalesCalculator(sales).meanOfHighSales())
8             Spark.sc.stop
9         } else {
10            println("There should be only one argument which is the file path.")
11            System.exit(-1)
12        }
13    }
14 }
```

Listing 7: Implementation of MainClass.scala

```
1 package ch.unibe.scg
2 import org.apache.spark.rdd.RDD
3 object SalesReader {
4     def read(filePath: String): RDD[Int] = {
5         Spark.sc.textFile(filePath).map(entry => entry.toInt)
6     }
7 }
```

Listing 8: Implementation of SalesReader.scala

```
1 package ch.unibe.scg
2 import org.apache.spark.rdd.RDD
3 class SalesCalculator(sales: RDD[Int]) {
4     val highSalesThreshold = 75
5     val lowSalesThreshold = 50
6     def highSales(): RDD[Int] = {
```

```

7   sales.filter(entry => entry >= highSalesThreshold)
8   }
9   def mediumSales(): RDD[Int] = {
10    sales.filter(entry => entry >= lowSalesThreshold && entry < highSalesThreshold)
11  }
12  def lowSales(): RDD[Int] = {
13    sales.filter(entry => entry < lowSalesThreshold)
14  }
15  def meanOfHighSales(): Double = {
16    highSales().mean()
17  }
18  def meanOfMediumSales(): Double = {
19    mediumSales().mean()
20  }
21  def meanOfLowSales(): Double = {
22    lowSales().mean()
23  }
24  }

```

Listing 9: Implementation of SalesCalculator.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.{SparkConf, SparkContext}
3 object Spark {
4   private var context: SparkContext =
5   def sc: SparkContext = if (context == null) {
6     throw new RuntimeException("you should build the context before calling it")
7   } else {
8     context
9   }
10  def buildContext(env: Env.Value): SparkContext = {
11    if (context == null) {
12      context = if (env == Env.cluster) {
13        new SparkContext(new SparkConf().set("spark.app.name", "SalesReportApp"))
14      } else {
15        val conf = new SparkConf()
16          .setAppName("SalesReportApp")
17          .setMaster("local[2]")
18        new SparkContext(conf)
19      }
20      context
21    } else {
22      context
23    }
24  }
25 }

```

Listing 10: Implementation of Spark.scala

```

1 package ch.unibe.scg
2 object Env extends Enumeration {
3   type Level = Value
4   val test: Env.Value = Value("local")
5   val cluster: Env.Value = Value("cluster")
6 }

```

Listing 11: Implementation of Env.scala

```

1 import ch.unibe.scg.{Env, SalesCalculator, SalesReader, Spark}
2 import org.apache.spark.rdd.RDD
3 import org.scalatest.{BeforeAndAfterAll, FunSuite, Matchers}

```



```

4 class SalesTest extends FunSuite with BeforeAndAfterAll with Matchers {
5   Spark.buildContext(Env.test)
6   val path: String = getClass.getClassLoader.getResource("sales.txt").getPath
7   val salesRecords: RDD[Int] = SalesReader.read(path)
8   test("Reading the sales file produces the right number of records") {
9     salesRecords.count() shouldEqual 4
10  }
11  test("Average of the high sales should be calculated correctly") {
12    new SalesCalculator(salesRecords).meanOfHighSales() shouldEqual 118.0
13  }
14 }

```

Listing 12: Implementation of SalesTest.scala

B.2 Implementation of the Second Bug: Stack Overflow

```

1 package ch.unibe.scg
2 import org.apache.spark.sql._
3 object MainClass {
4   def main(args: Array[String]) {
5     if (args.length == 1) {
6       val spark: SparkSession = Spark.buildSparkSession(Env.cluster)
7       import spark.implicits._
8       val aJR = GradeFileReader.read(args(0))
9       //"hdfs://leela.unibe.ch:9000//grades.txt"
10      val reports = Seq.fill(10)(GradeReport(GradeReport(aJR)))
11        .toDS().groupByKey(identity).count.map(s => s).collect
12    } else {
13      println("There should be only one argument which is the file path")
14      System.exit(-1)
15    }
16  }
17 }

```

Listing 13: Implementation of MainClass.scala

```

1 package ch.unibe.scg
2 object GradeFileReader {
3   def reader(filePath: String): JuniorReport = {
4     val grades = Spark.sc.textFile(filePath).map(x => x.toInt).toLocalIterator.toArray
5     JuniorReport(grades(0), grades(1), grades(2), grades(3), grades(4), grades(5),
6       grades(6), grades(7), grades(8), grades(9), grades(10), grades(11),
7       grades(12), grades(13), grades(14), grades(15), grades(16), grades(17),
8       grades(18), grades(19), grades(20), grades(21), grades(22), grades(23))
9   }
10 }

```

Listing 14: Implementation of GradeFileReader.scala

```

1 package ch.unibe.scg
2 case class JuniorReport(
3   math01: Int, math02: Int, math03: Int, math04: Int, math05: Int, math06: Int,
4   chem01: Int, chem02: Int, chem03: Int, chem04: Int, chem05: Int, chem06: Int,
5   phys01: Int, phys02: Int, phys03: Int, phys04: Int, phys05: Int, phys06: Int,
6   bio01: Int, bio02: Int, bio03: Int, bio04: Int, bio05: Int, bio06: Int)
7 case class GradeReport[JuniorReport](aJR: JuniorReport, bJR: JuniorReport)
8 object GradeReport {
9   def apply[JuniorReport](aJR: JuniorReport): GradeReport[JuniorReport] =

```

```

10  new GradeReport(aJR, bJR)
11  }

```

Listing 15: Implementation of GradeReport.scala

```

1  package ch.unibe.scg
2  import org.apache.spark.{SparkConf, SparkContext}
3  import org.apache.spark.sql.SparkSession
4  object Spark {
5      private var context: SparkContext = _
6      private var sparkSession: SparkSession = _
7      def sc: SparkContext = if (context == null) {
8          throw new RuntimeException("you should build the context before calling it")
9      } else {
10         context
11     }
12     def buildContext(env: Env.Value): SparkContext = {
13         if (context == null) {
14             context = if (env == Env.cluster) {
15                 new SparkContext(new SparkConf().set("spark.app.name", "GradeApp"))
16             } else {
17                 val conf = new SparkConf()
18                     .setAppName("GradeApp")
19                     .setMaster("local[1]")
20                 new SparkContext(conf)
21             }
22             context
23         } else {
24             context
25         }
26     }
27     def ss: SparkSession = if (sparkSession == null) {
28         throw new RuntimeException("you should build the context before calling it")
29     } else {
30         sparkSession
31     }
32     def buildSparkSession(env: Env.Value): SparkSession = {
33         if (sparkSession == null) {
34             sparkSession = if (env == Env.cluster) {
35                 SparkSession.builder()
36                     .appName("GradeApp")
37                     .getOrCreate()
38             } else {
39                 SparkSession.builder()
40                     .appName("GradeApp")
41                     .master("local[1]")
42                     .getOrCreate()
43             }
44             sparkSession
45         }
46     }
47 }

```

Listing 16: Implementation of Spark.scala

```

1  package ch.unibe.scg
2  object Env extends Enumeration {
3      type Level = Value
4      val test: Env.Value = Value("local")
5      val cluster: Env.Value = Value("cluster")
6  }

```

Listing 17: Implementation of Env.scala

B.3 Implementation of the Third Bug: Corrupted Data

```

1 package ch.unibe.scg
2 object MainClass {
3   def main(args: Array[String]) {
4     if (args.length == 1) {
5       Spark.builderSparkSession(Env.cluster)
6       val filePath = args(0) // "hdfs://leela.unibe.ch:9000//employee_data.csv"
7       val employees = EmployeeReader.read(filePath)
8       println(EmployeeTransactions.findNumberOfEmployeesInDepartment(employees, 30))
9     } else {
10      println("There should be only one argument which is the file path.")
11      System.exit(-1)
12    }
13  }
14 }

```

Listing 18: Implementation of MainClass.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.sql.{DataSet, SparkSession}
3 case class Employee(employeeNo: Int, employeeName: String, designation: String,
4   manager: Int, hireDate: String, salary: Int, departmentNo: Int)
5 object EmployeeReader {
6   def read(filePath: String): Dataset[Employee] = {
7     val spark: SparkSession = Spark.buildSparkSession(Env.cluster)
8     import spark.implicits._
9     val employees = spark.sparkContext.textFile(filePath).map(_.split(","))
10      .map(emp => new Employee(emp(0).toInt, emp(1), emp(2), emp(3).toInt, emp(4),
11        emp(5).toInt, emp(6).toInt)).toDS()
12     employees
13   }
14 }

```

Listing 19: Implementation of EmployeeReader.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.sql.Dataset
3 object EmployeeTransactions {
4   def findNumberOfEmployeesInADepartment(employees: Dataset[Employee], dept: Int): Long = {
5     employees.filter(employees("departmentNo") == dept).count()
6   }
7 }

```

Listing 20: Implementation of EmployeeTransactions.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.sql.SparkSession
3 object Spark {
4   private var session: SparkSession = _
5   def ss: SparkSession = if (sparkSession == null) {
6     throw new RuntimeException("you should build the context before calling it")
7   } else {
8     sparkSession
9   }

```

```

9   }
10  def buildSparkSession(env: Env.Value): SparkSession = {
11    if (session == null) {
12      session = if (env == Env.cluster) {
13        SparkSession
14          .builder()
15          .appName("FindEmployeeApp")
16          .getOrCreate()
17      } else {
18        SparkSession
19          .builder()
20          .master("local")
21          .appName("FindEmployeeApp")
22          .getOrCreate()
23      }
24      session
25    } else {
26      session
27    }
28  }
29 }

```

Listing 21: Implementation of Spark.scala

```

1 package ch.unibe.scg
2 object Env extends Enumeration {
3   type Level = Value
4   val test: Env.Value = Value("local")
5   val cluster: Env.Value = Value("cluster")
6 }

```

Listing 22: Implementation of Env.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.sql.Dataset
3 import org.scalatest.{BeforeAndAfterAll, FunSuite, Matchers}
4 class EmployeeFinderTest extends FunSuite with BeforeAndAfterAll with Matchers {
5   Spark.buildSparkSession(Env.test)
6   val path: String = getClass.getClassLoader.getResource("employee_data.csv").getPath
7   val employees: Dataset[Employee] = EmployeeReader.read(path)
8   test("Reading the employees file produces the right number of records") {
9     employees.count() shouldEqual 14
10  }
11  test("Number of employees in department 30 should be 6") {
12    EmployeeTransactions.findNumberOfEmployeesInADepartment(employees, 30) shouldEqual 6
13  }
14 }

```

Listing 23: Implementation of EmployeeFinderTest.scala

B.4 Implementation of the Fourth Bug: Stack Overflow

```

1 package ch.unibe.scg
2 object MainClass {
3   def main(args: Array[String]): Unit = {
4     if (args.length == 1) {
5       Spark.buildContext(Env.cluster)
6       var listOfDancers = DanceChoreography.start(args(0), Env.cluster)

```

7000	SMITH	CLERK	7902	12/17/1980	800	70
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	30
7566	TURNER	MANAGER	7839	4/2/1981	2975	20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	30
7698	MILLER	MANAGER	7839	5/1/1981	1000	30
7782	CLARK	MANAGER	7839	6/9/1981	2450	10
7788	SCOTT	ANALYST	7566	12/9/1982	3000	20
7839	KING	PRESIDENT	2000	11/17/1981	5000	10
7844	TURNER	SALESMAN	7698	9/8/1981	1500	30
7876	ADAMS	CLERK	7788	1/12/1983	1100	20
7900	JAMES	CLERK	7698	12/3/1981	950	30
7902	FORD	ANALYST	7566	12/3/1981	3000	20
7934	MILLER	CLERK	7782	1/23/1982	1300	10

Table B.1: Data file (employee_data.csv) on the local machine

7000	SMITH	CLERK	7902	12/17/1980	800	70
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	30
7566	TURNER	MANAGER	7839	4/2/1981	2975	20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	30
7698	MILLER	MANAGER	7839	5/1/1981	thousand	30
7782	CLARK	MANAGER	7839	6/9/1981	2450	10
7788	SCOTT	ANALYST	7566	12/9/1982	3000	20
7839	KING	PRESIDENT	2000	11/17/1981	5000	10
7844	TURNER	SALESMAN	7698	9/8/1981	1500	30
7876	ADAMS	CLERK	7788	1/12/1983	1100	20
7900	JAMES	CLERK	7698	12/3/1981	950	30
7902	FORD	ANALYST	7566	12/3/1981	3000	20
7934	MILLER	CLERK	7782	1/23/1982	1300	10

Table B.2: Data file (employee_data.csv) on the cluster

```

7 // "hdfs://leela.unibe.ch:9000//dancersList.csv"
8 println("Count = " + listOfDancers.count())
9 Spark.sc.stop
10 } else {
11     println("There should be only one argument which is the file path")
12     System.exit(-1)
13 }
14 }
15 }

```

Listing 24: Implementation of MainClass.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.rdd.RDD
3 object DancersReader {
4     def read(filePath: String, env: Env.Value, RDD[(String, String)] = {
5         Spark.buildContext(env).textFile(filePath)
6         .map(aLine => aLine.split(", "))
7         .map(aPair => (aPair(0), aPair(1)))

```

```

8   }
9 }

```

Listing 25: Implementation of DancersReader.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.rdd.RDD
3 object DanceChoreography {
4   def start(filepath: String, env: Env.Value): RDD[(String, String)] = {
5     var listOfDancers: RDD[(String, String)] = DancersReader.read(filepath, env)
6     var swappedDancersList: RDD[(String, String)] = null
7     for (i <- 1 to 1000) {
8       swappedDancersList = listOfDancers.map(_._swap)
9       listOfDancers = swappedDancersList
10    }
11    listOfDancers
12  }
13 }

```

Listing 26: Implementation of DanceChoreography.scala

```

1 package ch.unibe.scg
2 import org.apache.spark.{SparkConf, SparkContext}
3 object Spark {
4   private var context: SparkContext = _
5   def sc: SparkContext = if (context == null) {
6     throw new RuntimeException("you should build the context before calling it")
7   } else {
8     context
9   }
10  def buildContext(env: Env.Value): SparkContext = {
11    if (context == null) {
12      context = if (env == Env.cluster) {
13        new SparkContext(new SparkConf().set("spark.app.name", "DanceChoreography"))
14      } else {
15        val conf = new SparkConf()
16          .setAppName("DanceChoreography")
17          .setMaster("local[1]")
18        new SparkContext(conf)
19      }
20      context
21    } else {
22      context
23    }
24  }
25 }

```

Listing 27: Implementation of Spark.scala

```

1 package ch.unibe.scg
2 object Env extends Enumeration {
3   type Level = Value
4   val test: Env.Value = Value("local")
5   val cluster: Env.Value = Value("cluster")
6 }

```

Listing 28: Implementation of Env.scala



Debugging Flow

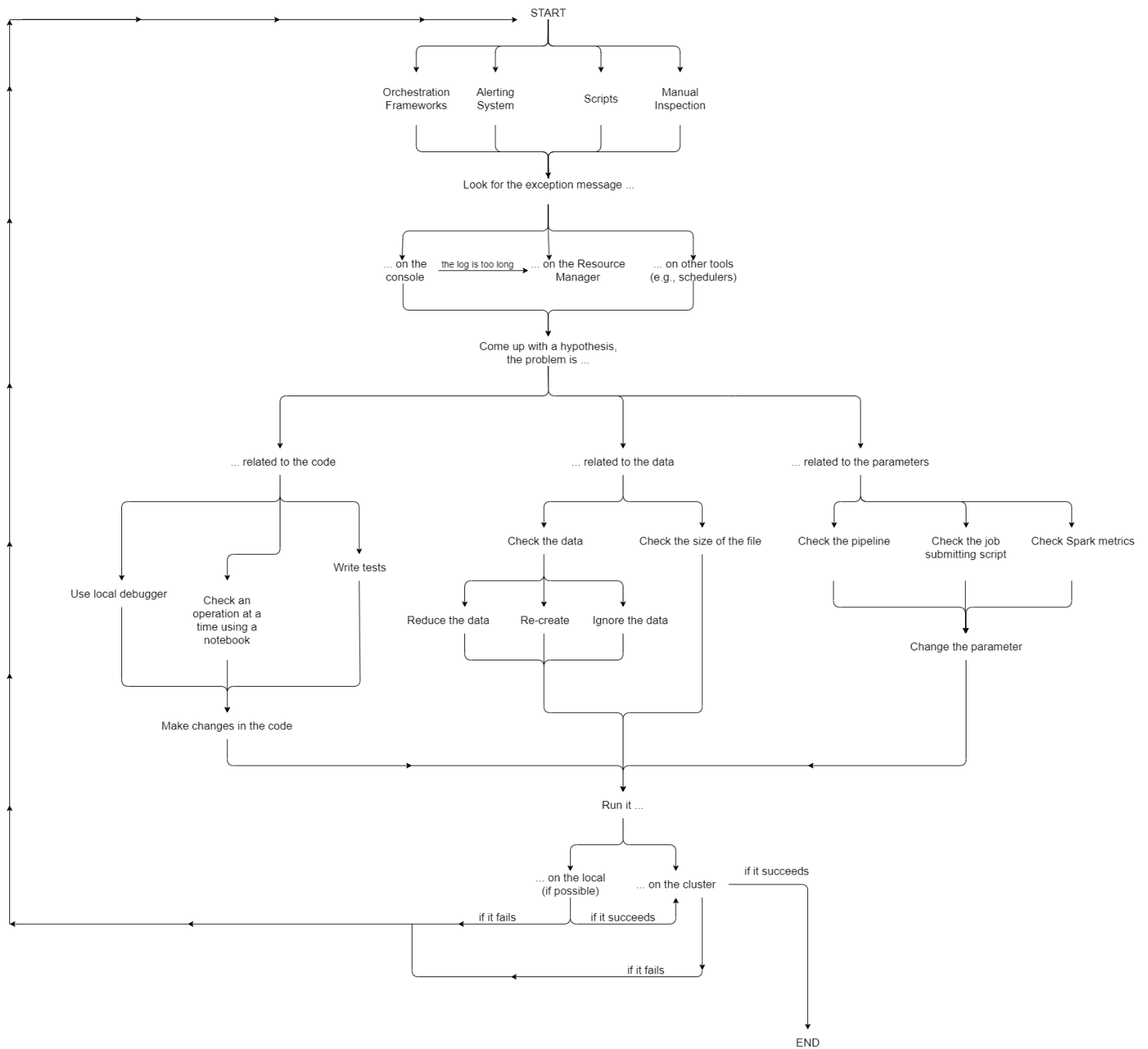


Figure C.1: Debugging Flow (complete version)

Bibliography

- [1] Rohan Achar, Pritha Dawn, and Cristina V. Lopes. GoTcha: An Interactive Debugger for GoT-Based Distributed Systems. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 94–110, New York, NY, USA, 2019. Association for Computing Machinery.
- [2] Cédric Aguerre, Thomas Morsellino, and Mohamed Mosbah. Fully-Distributed Debugging and Visualization of Distributed Systems in Anonymous Networks. 02 2012.
- [3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.
- [4] R. Balzer. EXDAMS: Extendable Debugging and Monitoring System. *Proc. AFIPS Sprint Joint Computer Conference*, 34:567–580, 05 1969.
- [5] Mortiz Beller, Niels Spruit, and Andy Zaidman. How Developers Debug. *PeerJ Preprints* 5:e2743v1, 2017.
- [6] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging Distributed Systems. *Commun. ACM*, 59(8):32–37, July 2016.
- [7] W. H. Cheung, J. P. Black, and E. Manning. A Framework for Distributed Debugging. *IEEE Software*, 7(1):106–115, Jan 1990.
- [8] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live Debugging of Distributed Systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, CC '09*, page 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.
- [9] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: Visual Signatures for Problem Diagnosis in Large Hadoop Clusters. In *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques*, lisa'12, page 33–42, USA, 2012. USENIX Association.
- [10] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, page 21, USA, 2007. USENIX Association.
- [11] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, page 18, USA, 2007. USENIX Association.

- [12] Herb Krasner. The Cost of Poor Quality Software in the US: A 2018 Report, September 2018.
- [13] Kyu Hyung Lee, Nick Sumner, Xiangyu Zhang, and Patrick Eugster. Unified Debugging of Distributed Systems with Recon. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN '11, page 85–96, USA, 2011. IEEE Computer Society.
- [14] Max Leske, Andrei Chiş, and Oscar Nierstrasz. A Promising Approach for Debugging Remote Promises. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 18:1–18:9, New York, NY, USA, 2016. ACM.
- [15] Henry Lieberman. The Debugging Scandal and What to Do About It (Introduction to the Special Section). *Commun. ACM*, 40:26–29, 1997.
- [16] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 423–437, USA, 2008. USENIX Association.
- [17] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, page 19, USA, 2007. USENIX Association.
- [18] Kamal Sheel Mishra, Anil Kumar Tripathi, Hiroshi Tamura, Futoshi Tasaki, and Ajay D. Kshemkalyani. Some Issues, Challenges and Problems of Distributed Software System. 2014.
- [19] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2003.
- [20] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, and Marcus Denker. Mercury: Properties and Design of a Remote Debugging Solution using Reflection. *The Journal of Object Technology*, 14:1:1, 09 2015.
- [21] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation - Volume 3*, NSDI'06, page 9, USA, 2006. USENIX Association.
- [22] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, April 2006.
- [23] Doug Woos, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. A Graphical Interactive Debugger for Distributed Systems. 2018.
- [24] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, page 213–228, USA, 2009. USENIX Association.
- [25] Haihan Yin, Christoph Bockisch, Mehmet Aksit, Wouter Borger, Bert Lagaisse, and Wouter Joosen. Debugging Scandal: The Next Generation. *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, June 2011.