Would You Like a Quick Peek? Providing Logging Support to Monitor Data Processing in Big Data Applications

Zehao Wang
Software Peformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Canada
w_zeha@encs.concordia.ca

Haoxiang Zhang*
Centre for Software Excellence at Huawei
Canada
haoxiang.zhang@huawei.com

Tse-Hsun(Peter) Chen†
Software Peformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Canada
peterc@encs.concordia.ca

Shaowei Wang
University of Manitoba
Winnipeg, Canada
shaowei.wang@umanitoba.ca

ABSTRACT

To analyze large-scale data efficiently, developers have created various big data processing frameworks (e.g., Apache Spark). These big data processing frameworks provide abstractions to developers so that they can focus on implementing the data analysis logic. In traditional software systems, developers leverage logging to monitor applications and record intermediate states to assist workload understanding and issue diagnosis. However, due to the abstraction and the peculiarity of big data frameworks, there is currently no effective monitoring approach for big data applications. In this paper, we first manually study 1,000 randomly sampled Spark-related questions on Stack Overflow to study their root causes and the type of information, if recorded, that can assist developers with motioning and diagnosis. Then, we design an approach, DPLOG, which assists developers with monitoring Spark applications. DPLOG leverages statistical sampling to minimize performance overhead and provides intermediate information and hint/warning messages for each data processing step of a chained method pipeline. We evaluate DPLOG on six benchmarking programs and find that DPLOG has a relatively small overhead (i.e., less than 10% increase in response time when processing 5GB data) compared to without using DPLOG, and reduce the overhead by over 500% compared to the baseline. Our user study with 20 developers shows that DPLOG can reduce the needed time to debug big data applications by 63% and the participants give DPLOG an average of 4.85/5 for its usefulness. The idea of DPLOG may be applied to other big data processing frameworks, and our study sheds light on future research opportunities in assisting developers with monitoring big data applications.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Information systems** → **MapReduce-based systems**; • **Software and its engineering** → Software testing and debugging.

KEYWORDS

Apache Spark, Logging, Monitoring

ACM Reference Format:
Zehao Wang, Haoxiang Zhang, Tse-Hsun(Peter) Chen, and Shaowei Wang. 2021. Would You Like a Quick Peek? Providing Logging Support to Monitor Data Processing in Big Data Applications. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3468613

1 INTRODUCTION

Due to advances in data science and technologies, the amount of data that is being created and collected is tremendous. Studies [14, 32] estimate that more than 90% of the data in the world has been generated in the past few years. The vast amount of data, once analyzed, provides opportunities for governments and companies to make data-driven decisions that help improve efficiency and generate revenues.

To analyze such large-scale data, developers have created various big data processing frameworks such as Apache Spark [35], Hadoop [21], and Google's MapReduce [11]. These big data processing frameworks provide abstractions to developers so that they can focus on implementing the data analysis logic. Using these frameworks, developers can scale the computation tasks horizontally across clusters of machines with little to no code changes and speed up computation. In particular, Spark has become one of the largest and most popular big data processing frameworks due to its intuitive API design and performance [18].

In traditional software applications, developers may use logging frameworks such as Log4J to insert logging statements in application source code. Then, developers use the generated logs to assist in monitoring [13, 40], testing [6, 7], and debugging [5, 42, 43]. However, due to the abstraction provided by Spark, there may be peculiar challenges if developers want to add logging statements to monitor big data applications. First, developers often leverage

*This work is not related to his role at Huawei.
†Corresponding author: Tse-Hsun(Peter) Chen (peterc@encs.concordia.ca)

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468613

method chaining (e.g., `filter().dropna().distinct()`) to create a data processing pipeline in Spark. Method chaining is one of the core concepts in Spark's API design to promote data immutability, which helps reduce concurrency issues related to data sharing. Method chaining may introduce challenges if developers need to monitor or understand how the data is transformed in each step, as developers can only see the final output. Second, Spark implements lazy evaluation to optimize the data processing pipeline. Breaking the data processing pipeline to record intermediate information in each step may significantly affect application performance.

The default logging provided by Spark only records information related to the Spark internals such as cluster resource allocation. However, knowing the states of Spark's internals may be not sufficient for developers, and there is limited support on recording the execution information on the application side. Due to the importance and complexity of data processing applications, there is a need for logging solutions to better monitor data transformation and provide useful system execution information to assist problem diagnosis. In this paper, we focus our study on Spark due to its popularity. However, our findings are applicable to other big data frameworks.

To address the above-mentioned challenges and assist developers in various tasks related to Spark applications (e.g., monitoring and issue diagnosis), in this paper, we follow a three-phase sequential exploratory strategy [2, 9, 22]. Similar to prior research in software engineering [2, 25, 27], our goal is to first identify the challenges that developers encounter and propose an approach to assist them. First, we identify the challenges that developers encounter through a qualitative study. Specifically, we study the type of information that may be useful to developers when understanding system execution and diagnosing issues. We conduct a manual study on 1,000 Spark-related questions from Stack Overflow, which reaches a 95% confidence level and a 3% confidence interval. We found that questions related to data processing and Spark API usage are the most common challenges that developers encounter (63%). In particular, most issues are related to not knowing what are the intermediate states of the processed data, and improper usage of API that leads to unexpected results. Second, we design a logging approach, called DPLOG, to provide developers the capability to monitor and understand data processing execution. DPLOG leverages statistical sampling to minimize performance overhead, and provides intermediate processing information and hint messages in real-time for each data processing step of a chained method pipeline. Finally, we evaluate DPLOG by first measuring its performance overhead on six benchmarking programs. Through a user study, we also show that the logging information provided by DPLOG may also assist developers in diagnosing issues in data processing applications.

The contributions of this paper are as follows:

- Our empirical study on Spark-related questions on Stack Overflow uncovers common challenges that developers encounter. Most of the issues that developers have are related to data transformation and API usage. In particular, developers often have challenges knowing the intermediate data states that lead to unexpected results.
- We propose an approach, DPLOG [16], which assists developers with monitoring and understanding data processing in Spark.
- Through an evaluation of six benchmarking programs, we find that DPLOG has a relatively small overhead. Compared to without using DPLOG, the response time when processing 5GB data increases by less than 10%. DPLOG reduces the overhead by over 500% compared to the baseline.
- We demonstrate the usefulness of DPLOG through a user study. Our user study with 10 professional developers and 10 graduate students shows that DPLOG can reduce the needed time to diagnose issues in big data applications by an average of 63%. On average, the participants give DPLOG 4.85/5 for its usefulness.
- We discuss the implications of our findings and future research opportunities in assisting developers with developing and debugging big data applications.

In summary, we proposed a data-driven solution (i.e., DPLOG) based on real-world Spark challenges. DPLOG provides support to Spark developers to address/alleviate such challenges, and our evaluation of DPLOG demonstrates its small performance overhead and its usefulness in helping monitor and diagnose big data applications. Although DPLOG was implemented for Spark applications, the idea of DPLOG can be migrated to other big data frameworks, for which method chains are employed and the intermediate information of data is difficult to access. For instance, Hadoop also employs method chains on reducers and mappers for data processing jobs, so our approach can be migrated to Hadoop to monitor the intermediate states of data processing.

Paper Organization. Section 2 discusses the background of Spark and related work. Section 3 presents challenges in developing Spark applications that we uncover from Stack Overflow questions. Section 4 presents the design of DPLOG. Section 5 evaluates DPLOG. Section 6 discusses implications and future work. Section 7 discusses threats to validity. Section 8 concludes the paper.

2 BACKGROUND AND RELATED WORK

Background of Apache Spark. Apache Spark is a distributed cluster-computing framework that can execute the computation in parallel in a cluster. To assist developers with big data processing, Spark abstracts the underlying parallel computation and cluster management from developers. Spark provides APIs for four programming languages: Scala, Java, Python, and R. To process data, Spark provides three abstractions for distributed data: RDD (resilient distributed dataset), DataSet, and DataFrame. RDD is an immutable distributed collection of data elements that can be operated in parallel. After Spark 2.0, the Spark official guideline suggests replacing RDD with Dataset and DataFrame, which provide richer APIs and better performance optimization. DataSet and DataFrame both abstract the representations of distributed data, whereas the difference is that the data in DataSet is strongly-typed. In this paper, we implement our logging solution for Spark's Python API (PySpark). However, the concepts are applicable to other programming languages and our prototype solution can be easily extended. Note that PySpark only supports DataFrame since objects in Python are weakly-typed. Below, we focus our discussion on DataFrame.

Spark APIs leverage two important concepts in its design: **method chaining** and **lazy evaluation**. Method chaining is used to ensure data immutability (i.e., DataFrame objects are immutable to

avoid concurrency issues) and allows developers to create a data processing pipeline by chaining multiple data processing methods. For example, developers can call `dataFrame.filter(x > 3).dropDuplicates().sort()` as a chain. By chaining the three data processing methods (i.e., `filter`, `dropDuplicates`, and `sort`), each method would return a new DataFrame object that is used as the input for the next method. The final returned DataFrame object will have values larger than three being filtered, duplicates removed and sorted. Method chaining also provides an intuitive way for developers to combine multiple data processing methods.

To optimize the performance of the chained methods, Spark employs lazy evaluation for optimization. There are two types of methods: transformation and action. For example, `filter` is a transformation method and `count` is an action method (i.e., return the number of rows in the data). All the transformation methods are lazily evaluated until an action method is called. When an action method is called, similar to compiler optimization, some intermediate data processing steps in Spark may be optimized, combined, or even eliminated to improve performance. If no action method is called, Spark would not execute any transformation method. For example, `dataFrame.filter(x > 3)` will not filter the data, unless an action method is called, like `count`. Then, Spark will start to filter data and calculate the number of rows of the filtered data.

Existing Logging and Monitoring Supports for Spark.
Logging in Spark: In traditional software applications, developers add logging statements in source code to record the program state and application runtime information. These logs provide valuable information for developers to monitor application status and diagnose issues [5, 13, 40, 42, 43]. Spark is no exception and uses Log4J for logging. All activities that occur inside Spark can get logged to the shell console and/or the configured underlying storage (e.g., to files on the disk or in databases). By default, logging in Spark only records the information about Spark's internals and does not record application execution information (i.e., does not show how the data is processed or transformed). However, developers may need to record the intermediate application and data states, in a case when there is an issue during program execution or with the final result, developers may be left in the dark. Nevertheless, adding logging statements to retrieve and record the intermediate information of each data processing step can invalidate lazy evaluation, and cause significant performance overhead (e.g., the data needs to be collected from all the worker nodes for each step).
Cluster Resource Monitoring: Spark provides a web user interface (UI) that allows developers to monitor the status and resource consumption of a Spark cluster. In the web UI, developers can monitor information such as job status and directed acyclic graphs that show how Spark schedules and optimizes the data processing methods. However, the web UI only shows the internal execution information of the Spark framework. When there is an unexpected data processing output or error on the application side, the web UI cannot provide much useful information.

To assist developers with logging and monitoring big data applications, in this paper, we first conduct an empirical study on the development challenges that developers encounter. We manually analyze Spark-related questions on Stack Overflow, with a focus on understanding the real-world challenges that developers encounter when running Spark application code, and what kind of information, if recorded, is useful for developers to understand the intermediate outcome for supporting the effective development of Spark applications. Based on our findings, we design a logging solution, called DPLOG, which can better monitor data transformation and provide useful system execution information to developers, especially supporting the commonly occurred issues that are encountered based on our empirical study.

Below, we discuss related work on the challenges and supports in developing big data applications.

Understanding the Challenges of Developing Big Data Applications. Bagherzadeh et al. [1] applied topic modeling (i.e., LDA) to study the topics of the questions that developers ask on Stack Overflow. They find that developers ask questions about MapReduce, debugging, and basic concepts more frequently than some questions such as performance. Their exploratory study provides a landscape of big-data questions that developers ask and is a starting point for future research. However, since the study is entirely quantitative, the study provides limited insights on what types of information could be helpful for developing and debugging big data applications. For example, they did not discuss the challenges of using API functions or the data processing problems that developers encounter. Kim et al. [24] surveyed 793 Microsoft data scientists on the common challenges that they encounter. They find that the most common challenges are related to data quality and the scale of the data. Fisher et al. [12] also interviewed 16 data analysts at Microsoft and they found that debugging in a distributed cloud environment is extremely challenging. Zhou et al. [45] analyzed 210 issue reports from one of Microsoft's big data platforms. They find that more than 30% of the issues are related to application design and code logic. In this paper, we focus our qualitative study on Stack Overflow questions related to Spark development. Through our qualitative study, we observe that questions related to data processing and Spark API usage are the most common challenges that developers encounter. In particular, most questions are related to understanding how the data is processed and its intermediate state in a data processing pipeline. We then design an approach aiming to assist developers in mitigating such challenges.

Debugging Big Data Applications. Dave et al. [10] proposed Arthur, which is a debugger for Hadoop and Spark. Arthur enables a user to selectively replay a part of the original computation. Gulzar et al. [17–20] developed a series of techniques to support debugging and testing for big data applications. Gulzar et al. proposed BIGDEBUG [18], which is a debugger for Spark applications. After specifying the breakpoints manually, users can use BIGDEBUG to debug Spark applications without needing to interrupt or re-run Spark applications during debugging. In another work, Gulzar et al. developed another debugging tool called BigSift [17, 20]. Given a known error caused by the input data, users can specify a predicate that helps flag the problematic data entries. Then, BigSift applies delta debugging to find the data entry, for which the corresponding output violates the pre-defined predicate. Different from prior debugging studies, in this paper, we focus on providing logging supports to developers. We first conduct an empirical study to identify the challenges that developers encounter when developing Spark

applications and identify the types of information that may assist developers in monitoring Spark applications. Unlike debuggers, logging provides insights on application execution and requires low performance overhead, since it is often used in production settings. Debugger, on the other hand, is used to debug a known issue, often makes the application runs hundreds of times slower, and is used only in development settings.

3 CHALLENGES IN DEVELOPING SPARK APPLICATIONS

In this section, we analyze Spark-related questions that were asked on Stack Overflow. We wish to understand the real-world challenges that developers encounter, and what kind of logging supports may assist developers with monitoring and developing Spark applications.

Stack Overflow is being widely studied for understanding the challenges in various areas of software engineering, such as security, mobile development, and AI-based systems [29, 31, 41, 44]. Similarly, we analyze Spark-related questions on Stack Overflow to understand the challenges in developing Spark applications. We download the Stack Overflow data dump that was released in September 2019. The data dump contains detailed information on every question and answer on Stack Overflow. Stack Overflow requires every question to have at least one tag to illustrate its topic. We use the tag *apache-spark* to select all Spark-related posts (i.e., questions and the associated answers). We follow prior studies [30, 39] to select only the questions that have a score that is higher than zero and has an accepted answer. Moreover, we filter out the questions that do not have any code snippets, since we wish to study the code snippets to further understand the possible causes of the challenge that the asker encountered. We collected 12,217 Spark-related questions that were asked between 2014 to 2019 (Spark 1.0 was released in 2014).

We conduct a qualitative study on a statistically significant sample of questions and their associated answers. More specifically, we randomly sample 1,000 questions among these 12,217 questions, achieving a confidence level of 95% and a confidence interval of 3%. We performed a lightweight open coding-like process that involves three phases and is performed by two authors (i.e., A1 and A3) in the paper. We describe the phases to conduct this qualitative study as follows:

- Phase I: A1 and A3 collaboratively go through 200 questions and their associated answers to derive an initial list of the challenges that developers encounter.
- Phase II: A1 and A3 independently go through the rest of the 1,000 posts, and assign the derived categories to these posts. In this phase, we did not find any new categories.
- Phase III: A1 and A3 compare their assigned categories and any disagreement is discussed until a consensus is reached. The inter-rater agreement has a Cohen's Kappa of 0.825 before the consensus is reached, which is a high-level agreement [28]. Our manual study result is publicly available [16].

We find that the most common challenge that developers encounter is related to Data Processing (43.2%). In general, there are two categories of issues that developers encounter during data processing. The first issue type is that the application may return unexpected data processing results (e.g., a bug in the code), but developers may have trouble in identifying which data processing method causes the issue. For example, a developer on Stack Overflow transformed the data by method chaining several data processing methods [38]. In this question, the developer wishes to understand and verify the result of each step for testing purposes. The suggested answer is to break the chained methods and test them separately. The second issue type is that, due to the vast number of supported frameworks and APIs in Spark, developers may be unfamiliar with some API usage or data format. Without knowing how the data looks like and how it is processed, developers may encounter unexpected challenges.

The second most common challenge is related to Spark API Usage (19.1%). Most of the problems in this category are caused by improper uses of APIs, which leads to unexpected data processing results without any indication of errors (e.g., no exceptions). Since Spark integrates the functional programming paradigm in its API design to abstract big data processing, sometimes developers may not be familiar with the working mechanism of an API and can use the API incorrectly. For example, a developer asked a question on Stack Overflow that the `fillna` method did not fill the null value as expected [37]. The developer planned to fill the integer 10 into all the cells that currently have a null value. However, the data type of the column is String while the developer planned to fill the data with integers. In Spark, if the data type of the filled value does not match with the data type of a column, the replacement would simply have no effect. There will be no warning messages, so it is difficult for developers to notice the issue. Some data processing methods also contain optional parameters that provide different ways to process data, but developers may not always be aware of such options. As an example, a developer is confused about the difference between `dropDuplicates` and `distinct` [36]. Both methods can remove duplicated data, and `dropDuplicates` has an additional parameter that is optional, from which the developer can specify the duplicated columns to be removed. In this case, providing some hints on anomalous data processing results and parameter usage may help developers understand how the data is processed.

We also find that developers often encounter challenges in configuring Spark (15.1%) and its interaction with other data sources (11.4%). Developers often encounter configuration problems due to the variety and flexibility of configuration parameters. As Spark can integrate with a variety of data sources, such as databases, developers may have problems during this process. We find that 5.5% of the questions are related to performance and logging issues in Spark deployment. Developers have difficulties in configuring Spark's default logging, monitoring Spark execution in the cluster, or improving the performance of data processing. Finally, there are some questions that we categorize into Other category (5.4%), which includes known and unresolved bugs in Spark or questions that are related to programming language syntax.

In short, we find that questions related to Data Processing and Spark API Usage are the most common challenges that developers encounter – accounting for 63% of the studied questions. Our manual analysis suggests that developers may need to know intermediate results after each data processing method is executed step by step to gain an overview of how their data is processed in

the pipeline. Providing hints or warning messages on API parameter usage and anomalous data processing results may also provide additional support to developers. Such execution information can help developers understand and monitor their applications. Another observation is that, in most studied questions, developers are more interested in knowing examples of how the data is processed. Therefore, showing samples of data processing results may provide values for monitoring purposes. Due to the popularity of these issues, we design a logging approach that may assist developers in monitoring data processing.

Below, we summarize the Data Processing and Spark API Usage challenges that we manually uncovered.

- **Challenge 1:** Data processing in Spark usually involves a series of steps to transform the raw data into an understandable/usable format. However, due to Spark's method chaining and lazy-evaluation features, it is usually impossible for developers to know the intermediate results (or state) of the processed data during monitoring.
- **Challenge 2:** Due to the vast number of APIs and their rich options, developers may pass incorrect API options and result in unexpected results. Having a warning message on the used API and related options for each intermediate state may provide hints to developers on how the data is processed, especially when API methods are chained together to form a complex task.
- **Challenge 3:** Most of the answers in the studied questions are related to re-running the data processing methods separately. However, re-running the application when large input data can be time consuming. There is a lack of tooling supports that allow developers to monitor the data processing details with a reasonable runtime overhead.

To address the above-mentioned challenges, in the next section, we discuss the design of a logging approach that has a small performance overhead and can be easily adapted to existing Spark applications.

4 THE DESIGN OF DPLOG

We present the design of our approach, DPLOG, which assists developers with monitoring and understanding the data processing execution. DPLOG is a logging approach that provides the intermediate information (e.g., data changes and states, and anomalies in the data processing) from each of the executed Spark methods. Table 1 shows the list of data processing methods that are supported by DPLOG. These methods cover all the basic data processing methods provided by PySpark's DataFrame [33]. Based on our empirical study results, we follow the requirements described below when designing DPLOG:

- **REQ1: DPLOG should provide step-by-step data processing execution information to address challenge 1.** To assist developers with Spark development and provide necessary information for monitoring and diagnostic purposes, DPLOG needs to show how the data is transformed/processed after calling each method.
- **REQ2: DPLOG should provide hints to developers when a potential issue occurs during data processing to address challenge 2**. To assist developers with using data processing methods in Spark and identify potential issues with either the results or method usage, DPLOG needs to provide some hints to developers to help locate or avoid misuses (i.e., similar to *warn* level logs in traditional logging [26]).
- **REQ3: DPLOG should be scalable and have a low performance overhead to address challenge 3.** To assist developers with getting the intermediate information during runtime, DPLOG needs to have a relatively low performance overhead.

Below, we discuss the design of DPLOG that fulfills the three above-mentioned requirements.

4.1 REQ1: Step-Wise Application Execution Information

Recording intermediate information: DPLOG records information of the data after each data processing method is executed during runtime. Note that there are some technical challenges in obtaining the intermediate results before a data processing pipeline is finished. As discussed in Section 2, Spark allows application developers to create a data processing pipeline by chaining methods and leverages lazy-evaluation to optimize performance (Section 4.4 discusses the implementation details of DPLOG to address the challenges). In particular, DPLOG records two types of information: *data state* and *data processing*. For *data state*, DPLOG records the information of the data state before and after each method. DPLOG records the data state differently for each method. For example, for `filter`, DPLOG records the number of rows before and after applying `filter`. For `withColumn` (i.e., for creating a new column), DPLOG records the number of rows and the statistics of the newly added column, such as max, min, mean, and standard deviation. Recording such data state information helps gain a high-level overview of the data and how it changes. For *data processing*, DPLOG records a small sample of the data (e.g., 10 rows for display purposes) before and after applying each data processing method for the showcase. Therefore, developers can see examples of how the data is transformed and processed through each step of the data processing pipeline. If there is a logical bug in the data transformation process, developers may be able to spot the bug and identify where the bug happens in the pipeline with the information provided by DPLOG.

4.2 REQ2: Providing Hints on the Executed Data Processing Methods

To address REQ2 and provide hints to developers about the usage and potential issue for each data processing method, we design DPLOG to record the anomalous result for each method call and provide possible hint messages. DPLOG provides two types of hint messages: anomalies in the data processing result, and hints on the used values for the optional parameters in the data processing methods.

Hints on anomalous data processing results: Bugs that developers face do not always run into exceptions or failures, but may also be related to incorrect calculation or data. For example, if a developer wishes to delete a column in the data but, instead, the developer gives the name of another column by mistake. In this case, there will be no exceptions, but the processed data will be incorrect.

Table 1: The intermediate information and hints of the data processing methods recorded by DPLOG.

	REQ1		REQ2	
Method	**Data state**	**Data processing**	**Anomalies hint**	**Parameter hint**
filter	Filter condition, number of rows (before and after), percentage of data changed	Display samples of the filtered data	No data or over 70% are filtered	N/A
dropDuplicates	Number of rows (before and after), percentage of data changed	Display samples of the removed data	No data or over 70% are deleted	Use default value for optional parameter
distinct	Number of rows (before and after), percentage of data changed	Display samples of the removed data	No data or over 70% are deleted	N/A
dropna	Number of rows (before and after), number of nulls (before and after), percentage of data changed	The distribution of null values	No data or over 70% are deleted	Use default value for optional parameter
drop	Drop condition, Number of columns (before and after), percentage of data changed	Display samples of the dropped data	Number of deleted columns is	N/A not expected
fillna	Number of nulls (before and after), percentage of data changed	The distribution of null values	There are still null values	Use default value for optional parameter
join	Join condition, Number of rows (before and after), Percentage of data changed	N/A	N/A	Use default value for optional parameter
withColumn	Data type of the new column, number of columns (before and after), statistical information about the new column (max, min, mean, std dev)	Display samples of the new data	N/A	N/A
sort	Sort condition, samples of the original data	Display samples of the sorted data	N/A	N/A

As shown in Table 1, DPLOG provides hints on anomalous data processing results for various methods. For `filter`, `dropDuplicates`, `distinct`, and `dropna`, DPLOG provides hints if the resulting data changes significantly or does not change at all: either no data or over 70% of the data is removed. The assumption is that developers often apply the methods to remove some data, but if no data or too much data is removed, a hint message to warn the developers may be helpful. Note that developers can adjust the threshold value if needed. For `fillna`, we provide a hint message if there still exist null values in the data after executing `fillna`. Similarly, for `drop`, we provide a hint message if the specified column is not dropped as expected.

Hints on method parameters: As we found in Section 3, developers sometimes may not be familiar with the parameters used in data processing methods. To assist developers, DPLOG checks the values of the parameters given to the data processing method. If the parameter value is not given and the default value is used, DPLOG will provide a hint message on the effect of the default parameter value. For example, if the *subset* parameter of `dropDuplicates` is empty, by default, Spark will apply deduplication to all the columns, and the behaviour of `dropDuplicates` becomes the same as `distinct`. In this case, DPLOG will provide a hint message on the effect of not providing the *subset* parameter. *The rationale is that if the developer has provided a value for the optional parameter, the developer likely knows the effect of that parameter value.* In addition to `dropDuplicates`, `dropna` and `fillna` also have the *subset* parameter. Similarly, DPLOG will provide a hint message if the value for *subset* is not provided (i.e., the operation will be applied to columns). We also provide hints for `dropna` and `join`. For example, there is an optional parameter *how*, which changes the behaviour of the method. For `dropna`, when *how* is set to "any", it drops a row if it contains any nulls; when *how* is set to "all", it drops a row if all of its values are null. Similarly, the *how* parameter in `join` specifies how the data will be joined (e.g., inner join and left outer join). DPLOG will give a hint message on these optional parameters if developers did not provide any value. For the remaining methods that have no optional parameters, the hint messages are not provided.

4.3 REQ3: Minimizing Performance Overhead

To make DPLOG practical, DPLOG must be scalable so that it can handle large datasets and DPLOG must have a reasonable low-performance overhead. Spark optimizes data processing pipelines (i.e., method chaining of multiple calls to data processing methods) using lazy-evaluation and other optimization techniques (e.g., removing redundant computation). Therefore, if we directly record the intermediate information from every data processing method, we would make the optimization techniques invalid and affect the performance. Fortunately, many of the big data processing issues that we found during our manual analysis may also happen in a non-big data setting. Therefore, to minimize the performance overhead of DPLOG, DPLOG first creates a statistically significant sample of the data and spawns a new Spark job that applies the data processing methods step-by-step. Note that DPLOG processes the original data and the sampled data simultaneously. When applying the data processing methods on the sampled data, DPLOG records the intermediate information and provides hint messages. DPLOG supports random data sampling, and developers can choose the confidence level and confidence interval. Sampling is an effective way to provide a statistically significant representation of the data which is often precise and accurate [4]. By default, DPLOG applies random sampling with a 99% confidence level and a 3% confidence interval.

4.4 Implementation of DPLOG

To minimize code changes and configurations when using DPLOG, we implement DPLOG as an independent package. We implement

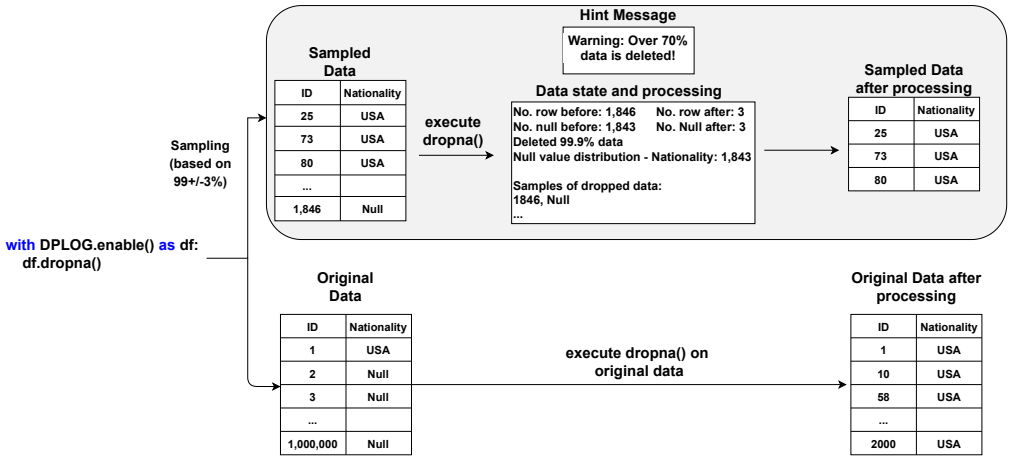

Figure 1: A working example of DPLOG.

DPLOG in Python and support PySpark, Spark's official Python APIs. Our implementation is based on Python 3.7.3 and is evaluated on Spark 2.4.4. DPLOG extends the functionality of PySpark, but developers do not need to learn the working principle of DPLOG and hardly need to modify any of the existing code. Developers can import DPLOG as a package and enable DPLOG by simply adding "`with DPLOG.enable() as df:`" (as shown in Figure 1). As shown in Table 1, DPLOG supports the following APIs in PySpark: `filter`, `drop`, `dropDuplicates`, `distinct`, `dropna`, `fillna`, `withColumn`, `sort`, and `join`. DPLOG covers all the basic data processing methods provided by PySpark's DataFrame [33]. When users read the data into a DataFrame object, DPLOG creates a new DataFrame object that stores the statistically significant sample of the original data. DPLOG processes the original data and the sampled data simultaneously according to the developers' source code. Note that, if needed, developers can also run DPLOG on the original data, even though the overhead will be significant (i.e., similar to the *debug* level in traditional logging frameworks).

DPLOG does not modify PySpark's source code. Instead, it uses the Adapter design pattern to extend PySpark's data processing methods without affecting their original implementations. Therefore, even if there is a new release of PySpark or the method implementation is modified, DPLOG can still be applied. The output of DPLOG (i.e., intermediate information and hint messages) can be saved to the location that the developer specified, or be recorded together with Spark's default logger. In addition to the messages, DPLOG will also save the sampled data to further assist monitoring and diagnosis if needed (i.e., developers can load the sampled data and diagnose potential issues).

Figure 1 shows a working example of DPLOG. First, DPLOG creates a sampled data based on 99% confidence level and 3% confidence interval. Then, DPLOG processes the sampled data and provides both the hint message and data state. Developers only need to add `with DPLOG.enable()` to enable DPLOG. In this example, the dataset contains a larger number of null values in the column *Nationality*. When the method `dropna` is called, DPLOG provides a hint message that over 70% of the data is deleted, and shows the statistics and samples of the dropped data. Since DPLOG is executed concurrently with the original dataset, the final result on the original data is not affected.

5 EVALUATION OF DPLOG

We evaluate DPLOG along two dimensions: performance overhead, and whether the recorded information can assist developers in understanding data processing and diagnosing potential issues.

RQ1: What is the performance overhead of DPLOG?

Motivation. As mentioned in Section 4, to make DPLOG practical and scalable, one of the requirements of DPLOG is to minimize the performance overhead. Therefore, in this RQ, we evaluate the performance overhead and scalability of DPLOG.

Approach. Our goal is to measure both the performance overhead and scalability of DPLOG. In particular, we implement six Spark benchmarking programs for our experiment. Table 2 provides an overview of the programs. These programs showcase common approaches of how developers use Spark for data processing [34], and are similar to the programs used in a prior study [18]. To minimize possible performance costs related to other non-Spark code, we design the programs so that they only leverage Spark APIs (which is also how big data processing applications are typically designed and implemented [18]). The implementation of our benchmarking programs is available online [16]. We measure the original response time (without using DPLOG), the overhead of running DPLOG, and the overhead of initializing DPLOG (i.e., sampling the data and creating a new DataFrame). To evaluate the effectiveness of our sampling mechanism, we also measure the response time of running DPLOG, while without sampling as a baseline. Namely, the baseline profiles every data processing step. By default, the unmodified Spark application would not monitor anything.

To measure the scalability of DPLOG, we run each program using three different levels of data size (i.e., small, medium, and large): 50MB, 500MB, and 5GB. The data size is increased tenfold for each level to better illustrate the scalability. Georges et al. [23] found that performance measurements suffer from instability, which may even lead to incorrect results. To mitigate the issue, we follow prior

Table 2: The description of the test programs that are used for performance benchmarking.

	Description	Executed Spark Method
P1	Drop null value and Filter data and add a new column based on condition.	`filter`, `dropna` `withColumn`
P2	Drop column and remove duplicate data	`distinct`, `drop`
P3	Drop column and null value and join two dataframe	`drop`, `dropna`, `join`
P4	Fill null value and drop columns and filter data	`drop`, `fillna`, `filter`
P5	Remove duplicate data and sort the data.	`dropDuplicates`, `sort`
P6	Join two dataframes and filter the data	`join`, `filter`

studies [8, 15, 23] and repeat each performance measurement 20 times. We run each program 80 times (i.e., 20 times for each data size) and report the mean and standard deviation of the response time (in seconds). We run our experiments on a server with a 2.6 GHz 6-Core Intel Core i7 CPU, 16GB DDR4 memory, and 500GB SSD disk.

Results. Table 3 shows the response time without DPLOG and with DPLOG, the total response time with DPLOG after including the data sampling step, and the baseline without sampling. Overall, we find that the overhead of DPLOG is consistent across programs. The runtime overhead (w/ DPLOG − w/o DPLOG) of DPLOG is around 1 to 3 seconds for the programs executed with three different data sizes. The overhead remains approximately the same even when the data size is increased by 100 folds (i.e., from small to large). We did a t-test to compare DPLOG and the baseline. The results are all statistically significant (p-values are less than 0.05) with large effect sizes. Our finding shows that DPLOG has good scalability since the overhead is consistently small even as the amount of data increases. The reason may be that the sample sizes do not increase much especially when the data population is large, so the overhead of applying the data processing methods is relatively consistent.

We also find that there is a larger overhead related to the initialization process when sampling the data (w/ sampling overhead − w/ DPLOG). To sample the data, DPLOG needs to first decide which data records should be sampled by generating a list of random indices. Then, as the data is not indexed, DPLOG needs to scan the entire data to find the corresponding data records, which results in a larger sampling overhead. However, DPLOG only needs to perform sampling once even if there are multiple data processing methods in the pipeline. Furthermore, we observe that the overhead of the baseline program is about 2 to 31 times higher than the overhead of DPLOG, and the overhead grows with the data size. It can be estimated that if the data size increases, the overhead of the baseline will increase significantly. In contrast, our sampling mechanism reduces the overhead by at least 500% when the data size is large, and the reduction is higher when the data size increases.

Finally, we examine if the performance overhead of running DPLOG grows linearly as the data size increases. We compute the response time ratio between running the programs with DPLOG and without DPLOG. The ratio of the response time of with DPLOG over response of without DPLOG against different data sizes. We observe that the ratios of the average overhead for small, medium, and large data sizes are 137.47%, 64.74%, and 9.25%, respectively. Namely, the ratio of the overhead decreases as the data size increases. One possible explanation is that, as we explained above, the sample sizes do not vary much when the data population size is large (e.g., the sample size eventually converges to 1,849 when the confidence level is 99% and the confidence interval is 3% no matter how big the data size is), so the overhead of applying DPLOG is relatively consistent rather than growing linearly with the data size.

> The overhead of DPLOG is significantly smaller than the baseline (reduced by an average of over 500%). DPLOG is also scalable as we find that the relative performance overhead decreases to less than 10% as the data size increases.

RQ2: How effective is DPLOG in assisting developers with issue diagnosis?

Motivation. The execution information provided by DPLOG may be used for various monitoring tasks. We also found in Section 3 that developers may encounter challenges in diagnosing issues in the data processing pipeline. As found by Beller et al. [3], most developers rely on logs to examine intermediate application execution state for issue diagnosis. Thus, in this RQ, we investigate the effectiveness of DPLOG in assisting developers in diagnosing data processing-related tasks.

Approach. We design a user study involving 20 participants (10 professional developers and 10 graduate students). These participants have one to five years of experience in either Spark or big data analysis. We design six issue diagnosis tasks based on the Stack Overflow questions that we studied in Section 3. Each task involves some data processing code and an injected issue. We abstract the irrelevant details from Stack Overflow posts and create a consistent format for the tasks so that developers can focus more on diagnosing the task itself. The user study also facilitates benchmarking the efficiency improvement by using our tool, since in real Stack Overflow questions, developers may need to spend more time to read and comprehend the question. To ensure the diversity of the selected tasks, each task has a different issue either in the data or in the used data processing methods. The tasks cover all the data processing methods that DPLOG supports. The tasks are related to filtering data based on some conditions, removing certain columns in the data, filtering data, removing duplicates, and filling or dropping N/A values. The description of the tasks is available online [16].

Each participant is assigned all six tasks and is required to diagnose three tasks with the help of DPLOG and diagnose another three tasks without using DPLOG. We randomize the order of the tasks for each participant to reduce the bias from the learning curve. Note that all the necessary working environments are set up for the participants, including the required packages and IDE. We provide detailed instructions on how to use DPLOG to each participant before starting the user study. During the experiment, each participant is provided with six source code files, where each file corresponds to each debugging task. The participants are allowed to run the program and make any necessary changes to identify the problem. When the participants believe that they have found the root cause of the problem in the program, we stop the timer. We record the time it takes for each participant to finish each task, and ask the participant to rank the usefulness of DPLOG on a scale from one to five, where one is considered as strongly disagree (i.e., not useful), and five is considered strongly agree (i.e., extremely useful).

Table 3: The response time of the studied programs (measured in seconds). The data size is increased by tenfold for each size. We show the average response time and standard deviation computed from the 20 repeated runs. *w/o DPLOG* shows the response time without DPLOG, *w/ DPLOG* shows the response with DPLOG (considering only the runtime overhead), *w/ sampling overhead* shows the total response time including both the runtime and initialization (i.e., sampling) overhead, and baseline shows the response time of the baseline (without sampling).

	Small Data Size				Medium Data Size				Large Data Size			
	w/o DPLOG	w/ DPLOG	w/ sampling overhead	baseline	w/o DPLOG	w/ DPLOG	w/ sampling overhead	baseline	w/o DPLOG	w/ DPLOG	w/ sampling overhead	baseline
Program1	0.86±0.16	2.45±0.40	3.46±0.53	6.07±0.95	1.83±0.13	3.46±0.39	4.71±0.57	21.82±2.12	13.61±1.24	14.54±1.65	19.97±2.12	214.22±10.64
Program2	2.17±0.31	4.18±0.57	5.32±0.70	7.08±0.99	4.13±0.47	6.39±0.67	7.89±0.85	17.91±1.63	29.14±2.27	31.93±2.70	37.94±3.35	134.18±13.25
Program3	0.98±0.19	2.07±0.30	3.21±0.45	3.85±0.49	2.61±0.29	3.54±0.46	4.91±0.71	16.57±1.62	23.11±2.05	23.79±2.24	29.46±2.90	179.87±13.31
Program4	0.84±0.15	2.81±0.44	3.87±0.60	5.40±0.80	1.84±0.22	3.80±0.60	5.12±0.82	17.05±1.34	12.91±1.32	14.75±1.50	20.41±2.10	140.95±9.44
Program5	3.12±0.41	5.67±0.99	6.73±1.16	9.93±1.51	4.97±0.58	7.07±1.14	8.27±1.30	26.86±2.49	28.27±2.10	30.94±3.34	36.22±3.86	208.38±15.38
Program6	1.01±0.11	2.00±0.31	3.06±0.50	3.75±0.48	1.92±0.17	2.96±0.39	4.30±0.60	11.73±1.02	13.32±1.70	14.56±1.49	20.12±1.75	100.08±6.42

Table 4: The average time for the participants to finish the given task with and without DPLOG.

	Avg. time w/o DPLOG (min)	Avg. time w/ DPLOG (min)	Improvement
T1	12.26	4.39	64%
T2	10.73	3.29	69%
T3	13.62	5.18	62%
T4	10.72	3.86	64%
T5	8.96	3.43	62%
T6	13.03	5.66	57%
Total	69.28	25.81	63%

Result. *On average, DPLOG reduces the needed time for the participants to diagnose the given tasks by 63%.* Table 4 shows the average time it takes for the user to diagnose the programs. Without using DPLOG, on average, the participants spent around 9 – 13 minutes to point out the potential causes of the issue. When using DPLOG, the average time reduced significantly to 3 – 5 minutes. For every task, using DPLOG helps reduce the debugging time by 57% to 69% (an average of 63%). For each task in the user study, we did a t-test to compare the needed time with and without the help of DPLOG. The results are statistically significant and all effect sizes are large. Our findings show that DPLOG is effective in assisting the participants in monitoring and diagnosing data processing issues in Spark applications.

Participants all agree that DPLOG is effective in helping with debugging (i.e., the average rating is 4.85/5). 100% of the participants either agree or strongly agree that DPLOG is effective in assisting them with monitoring and diagnosing data processing in Spark. For example, one participant mentioned "*Developers could easily find which step caused wrong data. This saves a lot of time.*" Among the 20 participants, 17 of them strongly agree that DPLOG provides the needed support, and 3 of them agree that DPLOG provides assistance in diagnosing issues. Some participants mention that when the data processing pipeline is longer, any issue that occurs during the pipeline becomes harder to diagnose, and "*DPLOG is even more useful when there are more data processing methods in the pipeline*".

Our user study finds that DPLOG can reduce the needed time to diagnose the given tasks by an average of 63%. The participants gave an average rating of 4.85/5 to DPLOG. All of the participants either agree or strongly agree that DPLOG helps them with monitoring and diagnosing Spark applications.

6 DISCUSSION

Implications of Our Study. Below, we discuss the insights that we observed in our analysis on Stack Overflow posts and user study. Although the observations are related to data processing in Spark, our research framework of the empirical analysis and the proposed logging support tool can be easily extended to other big data applications in future research.

Knowing the intermediate information of data is important for monitoring and debugging data processing applications. In our user study, for the tasks where the participants are not allowed to use DPLOG, we observe that some participants tried to analyze Spark's logs and use Spark's cloud resource monitoring tool to debug the programs. However, even if the participants knew that there exist some issues in the program, they could not identify the root causes using the existing approaches. In most cases, the participants found that manually printing the data state (e.g., calling `print`) is the only useful approach for debugging. Some participants kept decomposing the chained methods, printing the output of each individual method, and checking for potential issues. Although we found that there were fewer manually-added print statements when the participants use a Python debugger, they still need to continuously decompose the chained methods to manually debug the result of each data processing method. More importantly, for the tasks in which DPLOG is not used, even after the participants found the issue, they still need to conduct extra analysis to find the root cause of the issue in the programs.

Different from existing debugging supports, DPLOG provides the intermediate information of data processing methods, which helps avoid decomposing the chained methods and reduce debugging effort. For instance, one participant mentioned that "*The information provided by the tool is very useful and precise. I can find the reason for the problem much quicker based on the given information.*" Another participant mentioned that "*The tool is significantly better than printing information from the code. The information provided by the tool is quite rich and helpful for locating the problem.*" Another participant mentioned, "*The tool is very easy to use and provides useful information without manual debugging.*" We also observe that in 86.7% of the tasks in which DPLOG is used, participants successfully identified the cause of the issue, which is significantly higher than that of the tasks in which DPLOG is not used (70%). In other words, providing the intermediate information does help participants identify an issue and its root cause.

Providing hints on anomalous data processing results helps identify issues more quickly. As discussed in Section 4, DPLOG analyzes the execution of data processing methods and the value of their optional input parameters. If there is an anomalous result, DPLOG would provide a hint message. In our user study, we observe that such hints are useful for participants to notice the existence of an issue in the program. For example, one participant mentioned, "*the most promising advantage of the tool is it can alarm users for anomalous behavior.*" Another participant mentioned, "*Through these hints, it is easier for developers to quickly locate the abnormal behavior of the method or abnormal data.*"

Limitations and Future Work. Our study provides an initial step towards understanding the data processing execution. Even though our findings show that the overhead of DPLOG is small and it can assist developers with issue diagnosis, there are still some limitations and future research opportunities in assisting the development of big data applications.

Data Visualization and Information Presentation. One common suggestion from the participants is related to improving the UI design. Currently, DPLOG records the intermediate information and hint messages in the form of text (e.g., similar to traditional logs recorded by Log4J) without providing a rich user interface. A participant said, "*It would be perfect if the tool can finally come as an interactive form.*" Another participant said "*it is hard to find the warning message and useful tips. I would suggest making the warning and tips easier to identify.*" In addition to UI design, we also observed that visualizing the results of the data processing methods may also help developers quickly identify issues and understand the data state. For example, we find that in certain cases, some participants wanted to visualize the data using histograms to understand how the data distribution changes. Therefore, future studies may also consider leveraging data visualization to assist developers with monitoring big data applications.

Customizable Information Recording. Currently, DPLOG records all the data processing information that is described in Table 1. However, sometimes developers may already have an idea about possible issues that may occur, and they only want to record certain information. For example, in our user study, a participant mentioned that "*There is too much log information and it is not easy to locate the log I need immediately.*" The participant is a professional developer who has years of experience in developing Spark applications. Even though the participant found DPLOG to be useful (gave DPLOG 5/5 in terms of usefulness), he suggested a customizable configuration for recording only the needed data processing information. One approach may be to provide different logging levels, such as *debug*, *info*, *warn*, and *error*. Due to the vast amount of data and the complexity of big data applications, future studies may also investigate approaches, such as providing a domain-specific language, that could allow developers to record more customized and focused information to further assist monitoring and debugging.

More Advanced Debugging Assistance. We uncover common challenges that developers encounter by analyzing questions on Stack Overflow. We then design an approach, DPLOG, and evaluate it by conducting a user study. Although our user study shows promising results, there is still other information that can be added to assist developers. For example, future studies may investigate more advanced techniques for providing hint messages for anomalous data processing results using machine learning or artificial intelligence. Moreover, to reduce the overhead of DPLOG, we apply random sampling to select a statistically significant subset of data. Although sampling is an effective technique to reduce the data size while providing good precision on the original, future studies may investigate different sampling techniques and how they affect the effectiveness of debugging big data applications.

7 THREATS TO VALIDITY

External validity. Threats to external validity relate to the generalizability of our findings. In Section 3, we studied the Spark-related questions on Stack Overflow. The number of questions is large and it is impossible to study all of the questions qualitatively. To minimize the bias, we randomly sampled 1,000 statistically representative questions, giving a confidence level of 95% and a confidence interval of 3%. We implement DPLOG to support only Spark's Python version. However, our proposed methodology could be applied to the other languages and frameworks. Future research is encouraged to enhance the support for other programming languages. Similarly, we do not cover all the APIs for data processing. However, in this study, we cover all the basic data processing APIs for PySpark's DataFrame [33] and our user study demonstrates that DPLOG is effective in helping developers identify issues and their root causes. Future research is encouraged to apply our approach to other data processing APIs.

Internal Validity. Threats to internal validity are related to experimenter errors and bias. We conducted a qualitative study in Section 3 which was performed by humans and bias may be introduced. To reduce the bias, each question is examined by two of the authors individually and discrepancies are discussed until a consensus is reached. We measured the level of the inter-rate agreement in our qualitative study, and the agreement value is high (i.e., 0.825).

8 CONCLUSION

Big data technologies have changed how companies and organizations make decisions. Spark, as one of the most popular big data processing frameworks on the market, has been widely used in developing big data applications. In this study, we analyze the challenges that Spark developers encounter and propose DPLOG to assist developers in monitoring their big data applications. In short, this paper makes the following contributions: 1) We conduct an empirical study of Spark-related questions on Stack Overflow and identify the major challenges that Spark developers encounter: unknown intermediate data processing result and no support of warnings on improper API usages. 2) We propose an approach, DPLOG, to help developers monitor and diagnose data processing in Spark and implement it as a Python package. 3) DPLOG has a small runtime overhead. Through a user study, we find that DPLOG effectively reduces the average debugging time by 63%, and the participants highly praised the usefulness of DPLOG. 4) We discuss the implication of our findings and future research direction that can further help developers develop and debug Spark applications.

REFERENCES

[1] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going Big: A Large-scale Study on What Big Data Developers Ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. 432–442.

[2] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering (ICSE âĂŹ18)*. 752âĂŞ763.

[3] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior among Programmers *(ICSE '18)*. 572âĂŞ583.

[4] S. Boslaugh and P.A. Watters. 2008. *Statistics in a Nutshell: A Desktop Quick Reference.* O'Reilly Media.

[5] An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2021. Demystifying the challenges and benefits of analyzing user-reported logs in bug reports. *Empir. Softw. Eng.* 26, 1 (2021), 8.

[6] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18')*. 305–316.

[7] Jinfu Chen, Weiyi Shang, Ahmed E. Hassan, Yong Wang, and Jiangbin Lin. 2019. An Experience Report of Generating Load Tests Using Log-Recovered Workloads at Varying Granularities of User Behaviour. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. 669–681.

[8] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 1001âĂŞ1012.

[9] J.W. Creswell and J.D. Creswell. 2017. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.* SAGE Publications.

[10] Ankur Dave, M. Zaharia, S. Shenker, and I. Stoica. 2013. Arthur : Rich Post-Facto Debugging for Production Analytics Applications.

[11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. 137–150.

[12] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. 2012. Interactions with Big Data Analytics. *ACM Interactions* (May 2012).

[13] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Proceedings of the 36th International Conference on Software Engineering (ICSE-SEIP '14)*. 24–33.

[14] J. Gantz and David Reinsel. 2012. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East.

[15] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA âĂŹ07)*. 57âĂŞ76.

[16] Github. 2021. DPLOG repository. https://github.com/SPEAR-SE/FSE2021_DPLOG. Last accessed June 2021.

[17] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*. 520–534.

[18] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 784–795.

[19] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. White-Box Testing of Big Data Analytics with Complex User-Defined Functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. 290âĂŞ301. https://doi.org/10.1145/3338906.3338953

[20] Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. BigSift: Automated Debugging of Big Data Analytics in Data-Intensive Scalable Computing *(ESEC/FSE 2018)*. 863âĂŞ866. https://doi.org/10.1145/3236024.3264586

[21] Apache Hadoop. 2021. Hadoop. https://hadoop.apache.org/.

[22] W. E. Hanson, J. Creswell, V. P. Clark, K. Petska, and J. D. Creswell. 2005. Mixed Methods Research Designs in Counseling Psychology.

[23] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM âĂŹ13)*. 63âĂŞ74.

[24] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. 2018. Data Scientists in Software Teams: State of the Art and Challenges. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1024–1038.

[25] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2019. DLfinder: Characterizing and Detecting Duplicate Logging Code Smells. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 152âĂŞ163.

[26] Zhenhao Li, Heng Li, Tse-Hsun (Peter) Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. 12 pages.

[27] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. 1013âĂŞ1024.

[28] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.

[29] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 372âĂŞ383.

[30] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. 2014. Understanding and classifying the quality of technical forum questions. In *2014 14th International Conference on Quality Software*. 343–352.

[31] C. Rosen and Emad Shihab. 2015. What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering* (01 2015), 1–32.

[32] SINTEF. 2013. Big Data, for better or worse: 90% of world's data generated over last two years. https://www.sciencedaily.com/releases/2013/05/130522085217.htm. Last accessed June. 2021.

[33] Spark. 2021. PySpark 3.1.2 documentation. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#dataframe-apis. Last accessed June 2021.

[34] Spark. 2021. PySpark DataFrame basic usage. https://spark.apache.org/docs/latest/sql-getting-started.html. Last accessed June. 8 2021.

[35] Apache Spark. 2021. Spark. https://spark.apache.org.

[36] StackOverflow. 2021. The difference between dropDuplicates and distinct. Last accessed June. 8 2021.

[37] StackOverflow. 2021. Spark fillNa not replacing the null value. https://stackoverflow.com/questions/40395932/spark-fillna-not-replacing-the-null-value. Last accessed June. 8 2021.

[38] StackOverflow. 2021. Unit Testing spark dataframes transformation chaining. https://stackoverflow.com/questions/54403226/unit-testing-spark-dataframes-transformation-chaining. Last accessed June. 8 2021.

[39] Shaowei Wang, Tse-Hsun Peter Chen, and Ahmed E Hassan. 2018. How do users revise answers on technical Q&A websites? A case study on Stack Overflow. *IEEE Transactions on Software Engineering* (2018).

[40] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[41] Xin-Li Yang, David Lo, Xin Xia, Zhiyuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *Journal of Computer Science and Technology* 31 (09 2016), 910–924.

[42] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. 143–154.

[43] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*. 3–14.

[44] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE '19)*. 104–115.

[45] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. 2015. An Empirical Study on Quality Issues of Production Big Data Platform. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. 17âĂŞ26.