**Debugging Big Data Analytics in Spark with BigDebug**

*Abstract*— This paper mainly focus on Big Debug which provides real-time interactive debugging support for Data-Intensive Scalable Computing (DISC) systems, or more particularly, Apache Spark.

Keywords—big debug, big data, Spark, Apache, Hadoop

# Introduction

# An abundance of data in science, engineering, national security, and health care has led to the emerging field of big data analytics. To process massive quantities of data, developers leverage data-intensive scalable computing (DISC) systems in the cloud, such as Google's MapReduce, Hadoop, and Apache Spark. While DISC systems help to address the scalability challenges of big data analytics, they also introduce new challenges in debugging. The "big data debugging" project at UCLA addresses this debugging challenge by combining insights from both software engineering and database research communities.

# Background

## Big Data

Big data refers the large volume of data – both structured and unstructured – that inundates a business on a day-to-day basis. Big data can be analyzed for insights that lead to better decisions and strategic business moves. There are technology that are used to access the data stores on the big data and this technologies have also evolved with new advancements, features and functionalities. In this paper we use Apache Spark for most of the debugging analysis.

## Hadoop

Hadoop is a set of open source programs and procedures ie. essentially, they are free for anyone to use or modify, with a few exceptions which anyone can use as the "backbone" of their big data operations. Hadoop framework is based on a simple programming model (MapReduce) and used HDFS (Hadoop Distributed File System) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective.

## Data Provenance

Data provenance refers to the process of tracing and recording the origins of data and its movement between databases. Provenance is also essential to the business domain where it can be used to drill down to the source of data in a data warehouse, track the creation of intellectual property, and provide an audit trail for regulatory purposes. The use of data provenance is proposed in distributed systems to trace records through a dataflow, replay the dataflow on a subset of its original inputs and debug data flows.

## Resilient Distributed Datasets

It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. An RDD is a read-only, partitioned collection of records. RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset. RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure.

## Data Lineage

When a transformation (map or filter etc.) is called, it is not executed by Spark immediately, instead a lineage is created for each transformation. A lineage will keep track of what all transformations must be applied on that RDD, including the location from where it must read the data. When a new RDD has been created from an existing RDD, that new RDD contains a pointer to the parent RDD. Similarly, all the dependencies between the RDDs will be logged in a graph, rather than the actual data. This graph is called the lineage graph.

# Spark

Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational software process. Spark is not a modified version of Hadoop and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark. Spark uses Hadoop in two ways – one is storage and second is processing. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

Apache Spark is a fast and general-purpose cluster computing system used to analyze massive quantities of data. Apache Spark is Data-Intensive Scalable Computing (DISC). Apache Spark uses resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.

# Working Of Spark

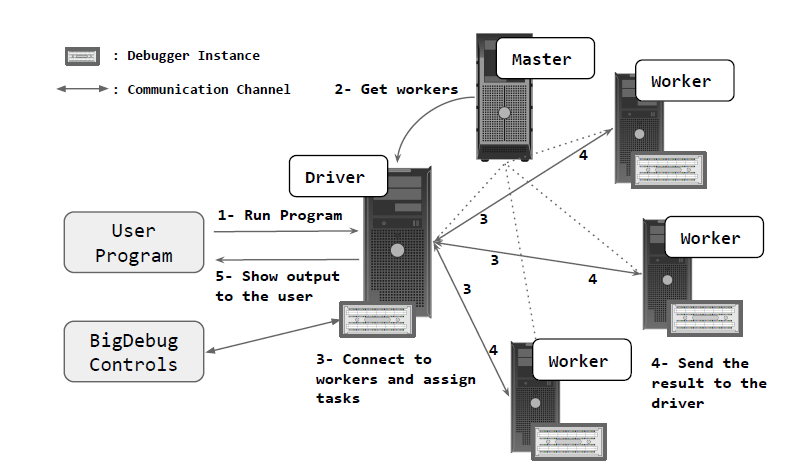
Spark translates the RDD transformations into something called DAG (Directed Acyclic Graph) and starts the execution. At high level, when any action is called on the RDD, Spark creates the DAG and submits to the DAG scheduler. (1) The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together. For e.g. Many map operators can be scheduled in a single stage. The result of a DAG scheduler is a set of stages. (2) The Stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies of the stages. (3) The Worker executes the tasks on the Slave.

The Spark platform consists of three modules: a driver, a master, and a worker. A master node controls distributed job execution and provides a rendezvous point between a driver and the workers. The master node monitors the liveliness of all worker nodes and tracks the available resources (i.e., CPU, RAM, SSD, etc.). Worker nodes are initiated as a process running in a JVM.

# Debugging With Apache Spark

Current approaches supporting data lineage in DISC systems do not meet our goals due to the following limitations:

* They use external storage such as a shared DBMS or distributed file systems (HDFS) to retain lineage information.
* Data provenance queries are supported in a separate programming interface.
* They provide very little support for viewing intermediate data or alternative data processing steps on intermediate data.



***Fig .1. Architecture of Spark with BigDebug***

# Titian

Titian, a library that enables data provenance tracking data through transformations in Apache Spark. Titian is built directly into the Spark platform and offers data provenance support at interactive speeds. Titian integrates with the Spark programming interface, which is based on a Resilient Distributed Dataset (RDD) abstraction defining a set of transformations and actions that process datasets. The data from a sequence of transformations, leading to an RDD, can be cached in memory. Spark maintains the program transformation lineage so that it can reconstruct lost RDD partitions in the case of a failure.

# Contributions - Titian

Titian offers the following contributions:

* A data lineage capture and query support system in Apache Spark.
* Lineage capturing design that minimizes the overhead on the target Spark program—most experiments exhibit an overhead of less than 30%.
* We show that our approach scales to large datasets with less overhead compared to prioir work.
* Interactive data provenance query support that extends the familiar Spark RDD programming model.
* An evaluation of Titian that includes a variety of design alternatives for capturing and tracing data lineage.

# Titian Record Generation

Titian generates new records in three places

* **Input:** Data imported from some external source *e.g.,* HDFS, Java Collection, etc.
* **Stage:** The output of a stage executed by a task.
* **Aggregate:** In an aggregation operation *i.e.,* combiner, group-by, reduce, and join

Each stage executes a series of RDD transformations until a shuffle step is required. Input agents generate and attach a unique identifier to each input record.  Stage input records could come from an external data source (e.g., HDFS) or from the result of a shuffle step. Aggregate agents generate unique identifiers for each output record, and relate an output record to all input records in the aggregation operation *i.e.,* combiner, reduce, group-by, and join.

# Bigdebub :Interactive Debugger

Since DISC systems offer very little tooling for debugging and, as a result, programmers spend countless hours analyzing log files and performing trial and error debugging. To aid this effort, UCLA developed Big Debug, an interactive debugging tool and automated fault localization service to help Apache Spark developers in debugging big data analytics.

## Vision

## The vision of BigDebugis to provide interactive, real-time debugging primitives for Big Data processing. Designing BigDebugrequires re-thinking the traditional step-through debugging primitives. Pausing the entire computation across distributed worker nodes causes significant delay and reduces overall throughput. Naively inspecting millions of records flowing through a data-parallel pipeline is too time-consuming and infeasible for an end user. BigDebugmust tag how individual records are flowing through individual worker nodes and transfer the requested debug information from the distributed worker nodes to the driver in an efficient manner. In other words, BigDebugmust meet the requirements of *low overhead*, *scalability*,and *fine-granularity*, while providing expressive debugging primitive.

## Contribution

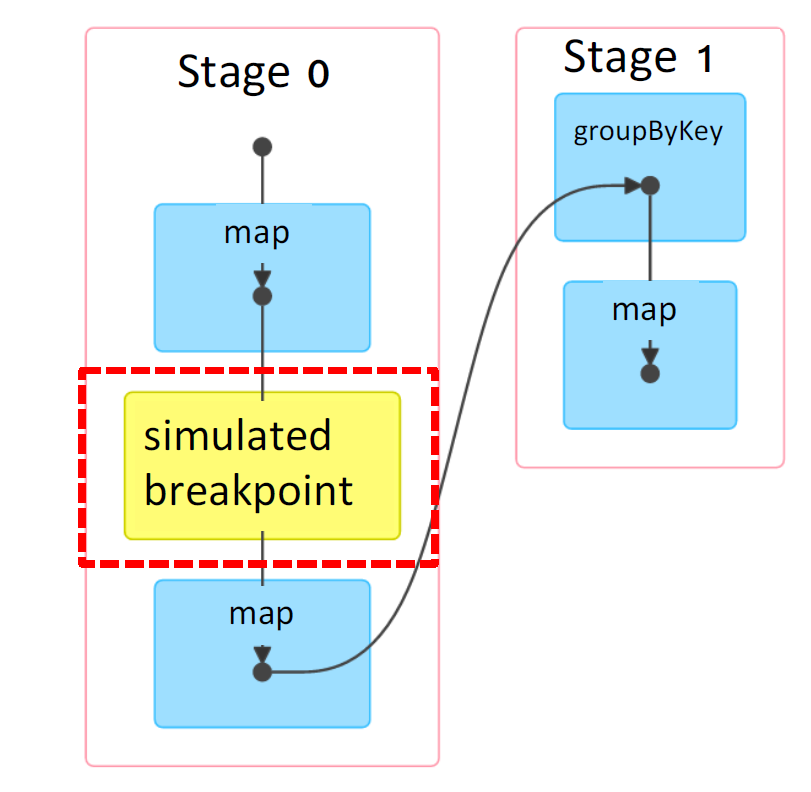
BigDebugoffers the following contributions:

* BigDebugprovides breakpoints and watchpoints with minimal performance impact.
* BigDebugexhibits less than 24% overhead for record-level tracing, 19% overhead for crash monitoring, and 9% for on-demand watchpoint on average.
* BigDebugquick fix and resume feature allows a user to avoid re-running a program from scratch, resulting in up to 100% time saving.
* BigDebugnarrows down the scope of failure inducing data by orders of magnitude through fine-grained tracing of individual records within the distributed, data processing pipeline.

# Interactive Debugger Features

**1.Simulated Breakpoints**

BigDebug provides simulated breakpoints that enable a user to inspect a program without pausing the entire distributed computation. There are two technical challenges in implementing breakpoints in Spark. First the traditional breakpoint will pause the entire execution while a user investigates an intermediate program state. Second, Spark optimizes its performance through pipelining transformations in a single stage. Therefore there is a mismatch between the logical view of data transformation and the physical view of data processing during debugging. It also supports on-demand watchpoints that enable a user to retrieve intermediate data using a guard predicate and transfer the selected data on demand. When a user requests intermediate results from the simulated breakpoint, BigDebug then recomputes the intermediate results and caches the results. BigDebug uses resume and step over command during the program implementation. When a user enters a resume command in BigDebug User interface, It will automatically jump to the original workflow running in the background. This procedure improves the overall throughput of the distributed processing. When a user enters a step over command to investigate the state after the next transformation in the UI, Bigdebug replays the execution to the next instruction only from the latest materialization point. This feature differentiates BigDebug from an existing replay debugger.

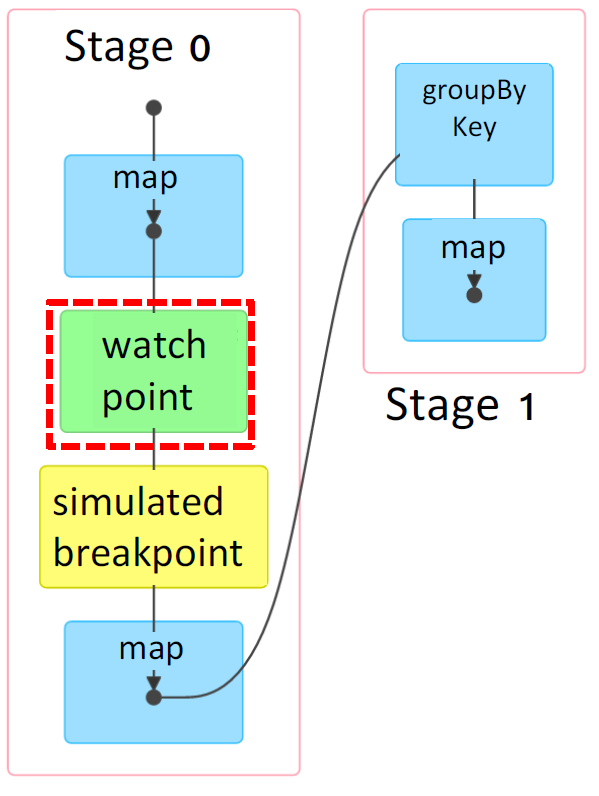


***Fig.2. Stimulated Breakpoint***

When a user finds inconsistency in intermediate data, the only option is to terminate the job and rewrite the program to handle the output. Terminating at later stage will case u loss in computation and cost. To save the cost of re-run, BigDebug allows a user to replace any code in the succeeding RDDs after the breakpoint. If a user wants to modify code, BigDebug applies the fix from the last materialization point rather than the beginning of the program to reuse previously computed results.

**2.Guarded Watchpoints**

BigDebug provides **guarded watchpoints,** which dynamically retrieve only those records that match a user-defined guard predicate. BIGDEBUG supports **fine-grained forward and backward tracing** at the level of individual records by leveraging prior work on data provenance within Spark. To avoid restarting a job from scratch in case of a crash, BIGDEBUG provides a **real-time quick fix and resume** feature where a user can modify code or data at runtime. It also provides **fine-grained latency monitoring** to notify a user which records are taking much longer than other records.



***Fig.3. Guarded Watchpoint***

**3**.**Crash**

While running a program on a much bigger data stored using a cluster mode a crash is encounters. Spark reports the physical view of the crash only—the type of crash, with a stack trace, the id of a failed task, the id of an executor node encountering the crash, the number of re-trials before reporting the crash, etc. However, such physical-layer information does not help. Individual have to debug which specific input log entry is causing the crash. Though Spark reports the task ID of a crash, it is impossible to know which records were assigned to the crashed executor and which specific entry is causing the crash. Even after identifies a subset of input records assigned to the task, it is not feasible for to manually inspect millions of records assigned to the failed task. Whereas when a crash occurs at an executor, BigDebug sends all the required information to the driver, so that the user can examine crash culprits.

***BigDebug reports:***

(1) a crash culprit—an intermediate record causing a crash

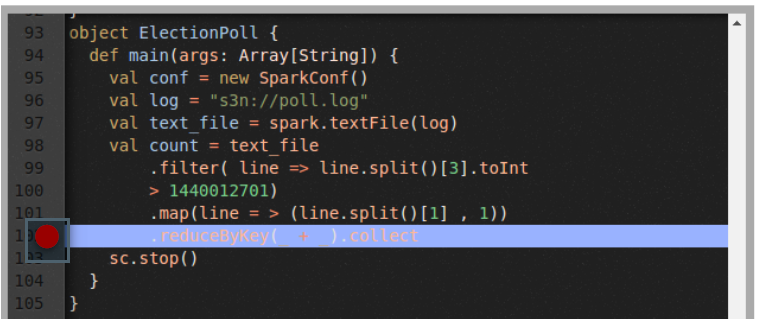
(2) a stack trace

(3) a crashed RDD

(4) The original input record inducing a crash by leveraging backward tracing.

**4.Real Time Code Fix and Resume**

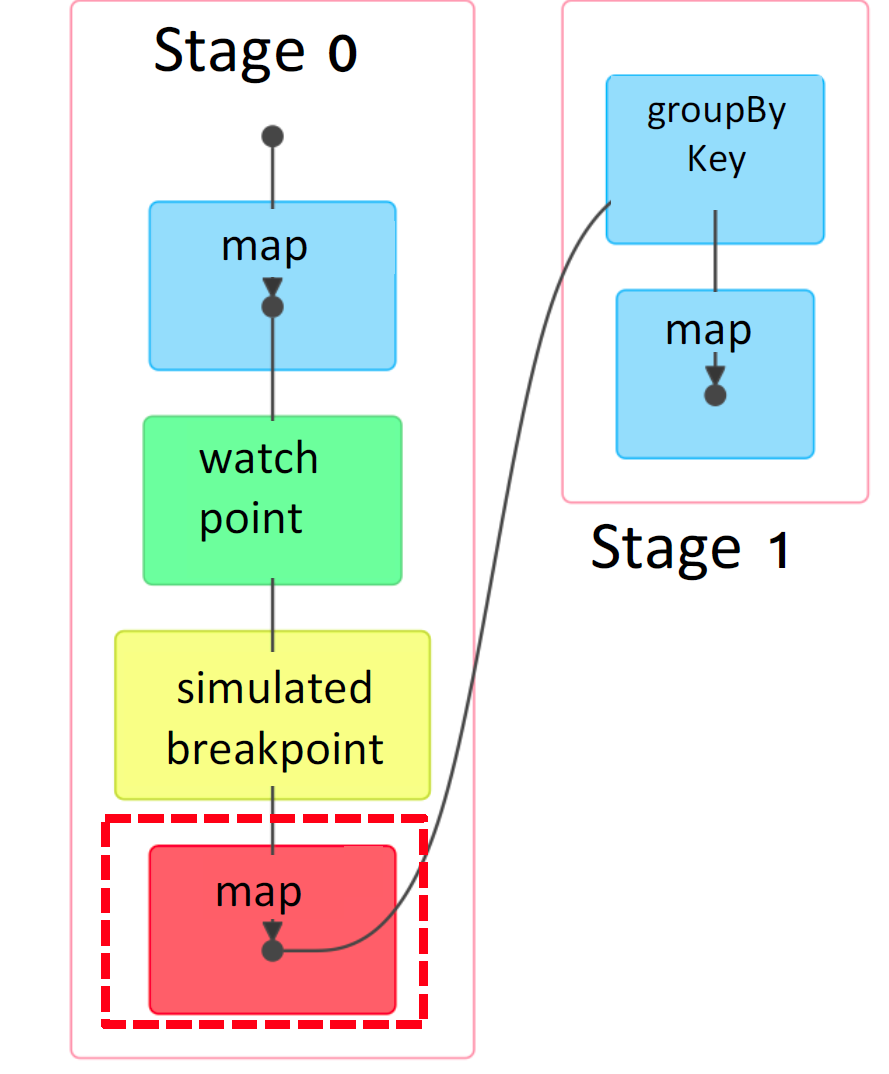
Without BigDebug, we modify the input data and restart the job from scratch, incurring wasted computation in running the services. However, using BigDebug, the code is fix on the fly by replacing the original crash point and resuming the failed computation.



***Fig.4. Real time Code fix***

**5.Crash Culprit Determination**

BigDebug provides the specific record causing the crash. Using backward tracing feature, it is possible to locates the specific log entry in the cluster or the data storages input causing the crash. BIGDEBUG runs pending tasks continuously to utilize idle resources in order to achieve high throughput. If a crash occurs, the original job keeps on running, while the user is notified of the fine-grained details of the crash. Once the crash culprit is reported to the user, the user can choose among three crash remediation options. First, a user can choose to skip the crash inducing record. The final output, in this case, will not reflect the skipped records. Second, a user can modify crash culprit records in Realtime, so that the modified record can be injected back into the pipeline. Third, a user can repair code. The whole process of modifying crash culprits is optimized through lazy remediation. While the user takes time to resolve crash culprits, BIGDEBUG continues processing the rest of the records, while also reporting any additional crashing record.

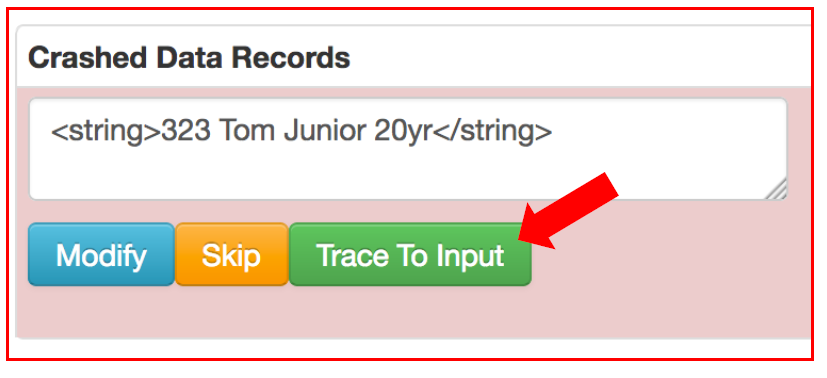


***Fig.5. Crash culprit identification***

**6. Forward and Backward Tracing**

BigDebug supports fine-grained tracing of individual records by invoking a data provenance query on the fly. BIGDEBUG uses data provenance capability implemented through an extension of Spark’s RDD abstraction.Fine-grained tracing allows users to reason about the faults in the program output or intermediate results and explain why a certain problem has occurred. Using backward tracing, a crash culprit record can be traced back to the original inputs responsible for the crash record. Forward tracing allows user to find the output records

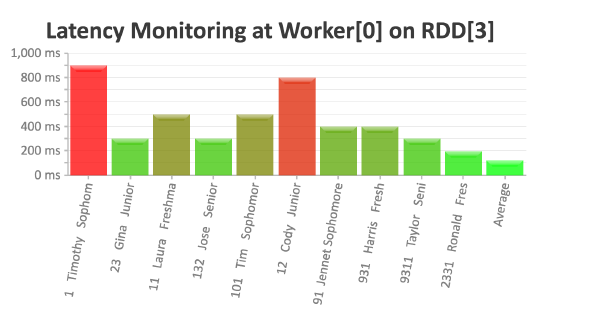
affected by a selected input.



***Fig.6. Trace Input***

**7.Fine Grained Latency Monitoring**

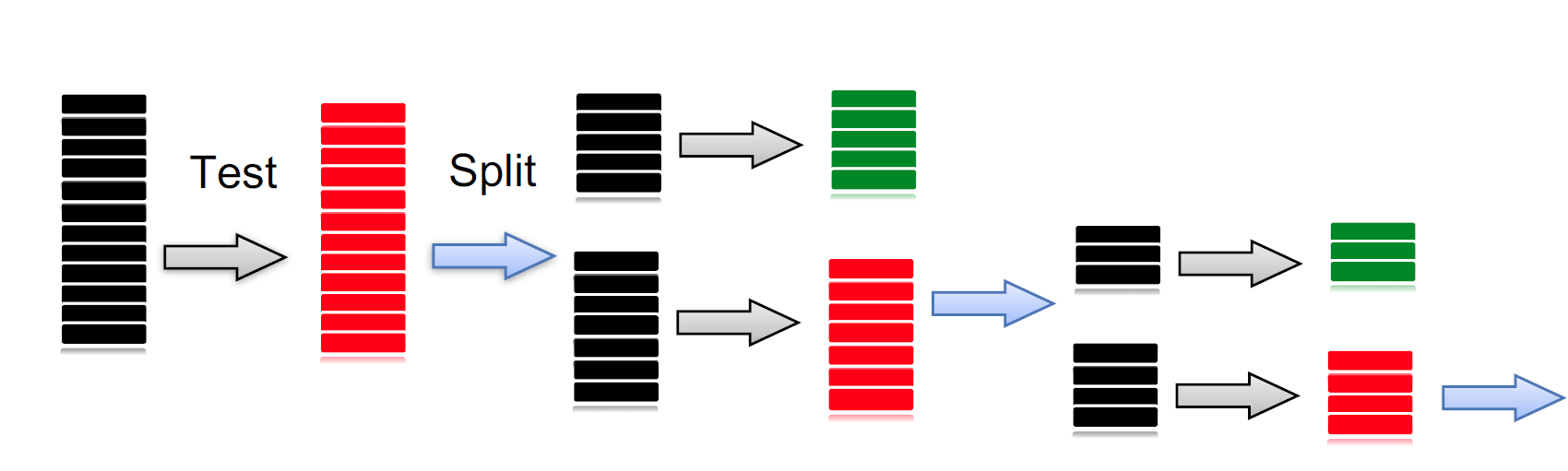
BigDebug identifies the record causing delay and computes the time taken to process each record .It stores the moving average and sends a report to monitor if the time is greater than standard deviation of each record.



***Fig.7. Latency Report***

**7.Automated fault Localization**

Delta Debug Fault localization algorithm is implemented to identify and diagnose the failure inducin.DD performs repetitive runs on different configurations of input to identify the root cause of failures.DD algorithm splits the original input into different sub-configurations using binary search strategy .The same sub-configurations serves as input for the same program.



***Fig.8. Fault Localization***

# Conclusion

Debugging Big data is daunting and a expensive process in terms of both the resources and time. BigDebug offers interactive debugging primitives for an in-memory data-intensive scalable computing (DISC) framework. To emulate interactive step-wise debugging without reducing throughput, BigDebug provides simulated breakpoints that enable a user to inspect a program without actually pausing the entire distributed computation. It also supports on-demand watchpoints that enable a user to retrieve intermediate data using a guard predicate and transfer the selected data on demand. To understand the flow of individual records within a pipeline of RDD transformations, BigDebug provides data provenance capability, which can help understand how errors propagate through data processing steps. To support efficient trial-and-error debugging, BigDebug enables users to change program logic in response to an error at runtime through a realtime code fix feature, and selectively replay the execution from that step. Finally, BigDebug proposes an automated fault localization service that leverages all the above features together to isolate failure-inducing inputs, diagnose the root cause of an error, and resume the workflow for only affected data and code.

##### References

[1] M. A. Gulzar and et al. Bigdebug: Debugging primitives for interactive big data processing in spark. In ICSE, 2016.

[2] M. A. Gulzar, X. Han, M. Interlandi, and et al. Interactive

debugging for big data analytics. In HotCloud, 2016.

[3] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim.

Bigdebug: Interactive debugger for big data analytics in

apache spark. In FSE, pages 1033–1037, 2016.

[4] <https://sites.google.com/site/sparkbigdebug/>

[5]<https://isr.uci.edu/sites/isr.uci.edu/files/speakers/slides/Kim-Feb-2018.pdf>

[6] A. Dave, M. Zaharia, and I. Stoica. Arthur: Rich post-facto

debugging for production analytics applications. Technical

report, Citeseer, 2013.