BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark

Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, Miryung Kim

University of California, Los Angeles

Developing Big Data Analytics

- Big Data Analytics is becoming increasingly important.
- Big Data Analytics are built using data intensive computing platforms such as Map Reduce, Hadoop, and Apache Spark.

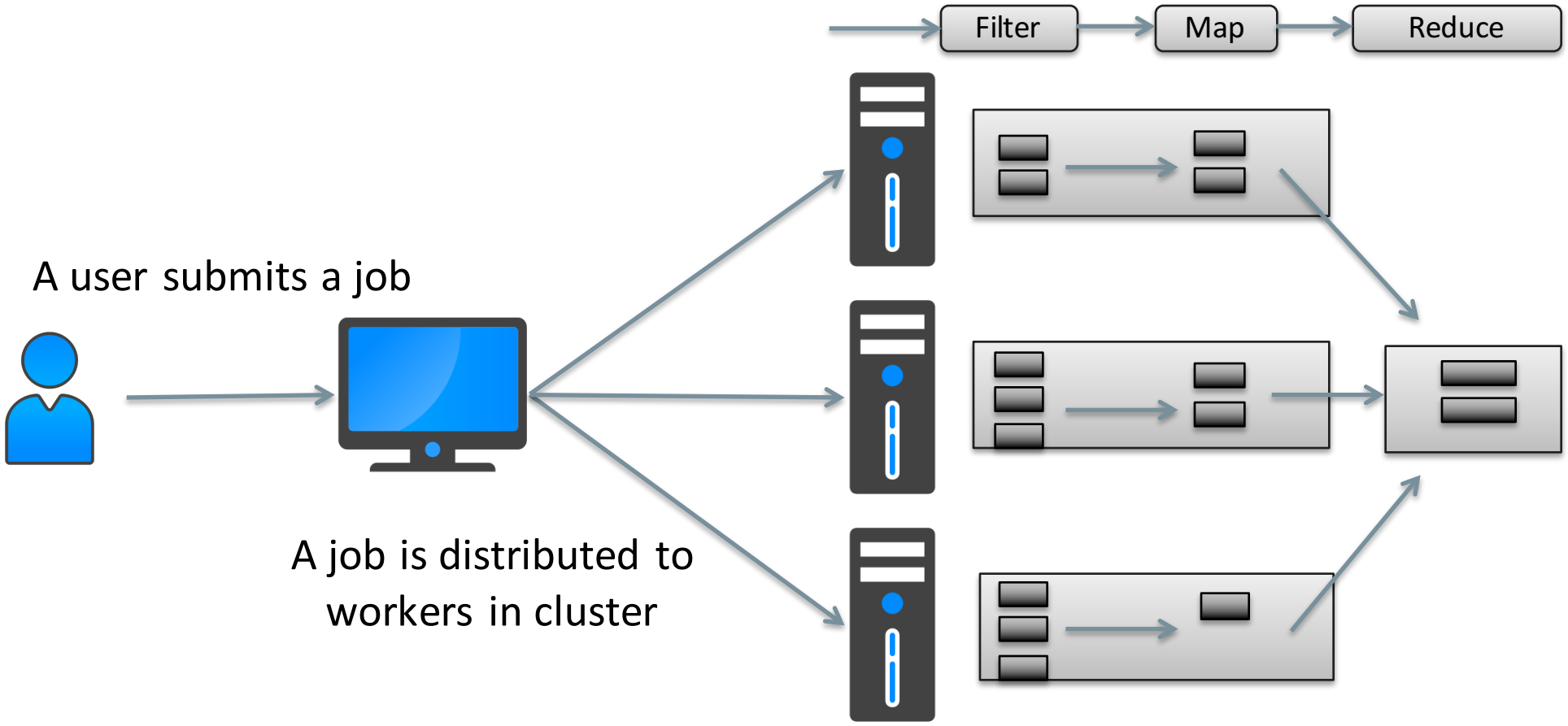
Apache Pig

Google Map Reduce

Apache Spark: Next Generation Map Reduce

- Apache Spark is up to 100X faster than Hadoop MapReduce
- It is open source and over 800 developers have contributed in its development
- 200+ companies are currently using Spark
- Spark also provides libraries such as SparkSQL and Mllib

Running a Map Reduce Job on Cluster



Each worker performs pipelined transformations on a partition with millions of records

Motivating Scenario: Election Record Analysis

- Alice writes a Spark program that runs correctly on local machine (100MB data) but crashes on cluster (1TB)
- Alice cannot see the crash-inducing intermediate result.
- Alice cannot identify which input from 1TB causing crash
- When crash occurs, all intermediate results are thrown away.

VoterID	Candidate	State	Time
9213	Sanders	TX	1440023087

```
1  val log = "s3n://poll.log"
2  val text_file = spark.textFile(log)
3  val count = text_file
4   .filter( line => line.split()[3].toInt
5  > 1440012701)
6   .map(line = > (line.split()[1] , 1))
7   .reduceByKey(_ + _).collect()
```

Motivating Scenario: Election Record Analysis

- Alice writes a Spark program that runs correctly on local machine (100MB data) but crashes on cluster (1TB)
- Alice cannot see the crash-inducing intermediate result.
- Alice cannot identify which input from 1TB causing crash
- When crash occurs, all intermediate results are thrown away.

VoterID	Candidate	State	Time
9213	Sanders	TX	1440023087

```
1  val log = "s3n://poll.log"
2  val text_file = spark.textFile(log)
3  val count = text_file
4   .filter( line => line.split()[3].toInt
5  > 1440012701)
6   .map(line = > (line.split()[1] , 1))
7   .reduceByKey(_ + _).collect()
```

```
Task 31 failed 3 times; aborting
job
ERROR Executor: Exception in
task 31 in stage 0 (TID 31)

java.lang.NumberFormatException
```

BigDebug: Interactive Debugger Features

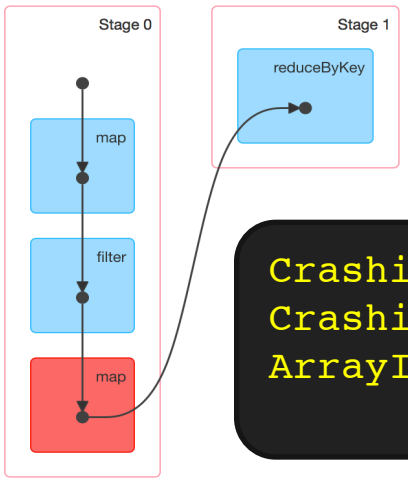
1 Simulated Breakpoint

```
93  object ElectionPoll {
94    def main(args: Array[String]) {
95      val conf = new SparkConf()
96      val log = "s3n://poll.log"
97      val text_file = spark.textFile(log)
98      val count = text_file
99          .filter( line => line.split()[3].toInt
100         > 1440012701)
101         .map(line = > (line.split()[1] , 1))
102         .reduceByKey( _ + _ ).collect
103     sc.stop()
104   }
105 }
```

2 Guarded Watchpoint

Captured Data Records
9213 Sanders TX 1440023087
9K23 Clinton TX 1440023645
9FG9 Cruz TX 1440023978
99LP Trump TX 1440023232
2FSD Cruz TX 1440026456

3 Crash Culprit Identification




```
Crashing at transformation 2
Crashing Record : "Sanders"
ArrayIndexOutofBoundException
```

4 Backward and Forward Tracing

Outline

- Interactive Debugging Primitives
    1. Simulated Breakpoint
    2. On-Demand Watchpoint
    3. Crash Culprit Identification
    4. Backward and Forward Tracing
    5. Fine Grained Latency Alert
- Performance Evaluation

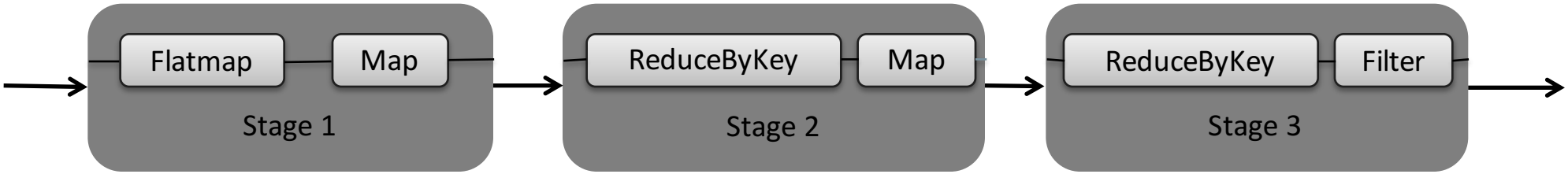
Why Traditional Debug Primitives Do Not Work for Apache Spark?

Enabling interactive debugging requires us to **re-think the features of traditional debugger** such as GDB

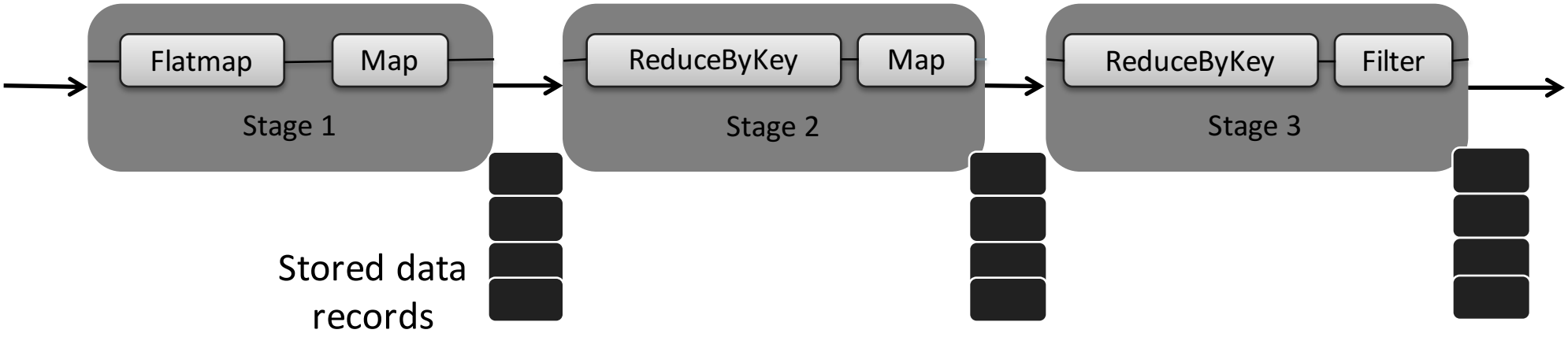
- Pausing the entire computation on the cloud could reduce throughput
- It is clearly infeasible for a user to inspect billion of records through a regular watchpoint
- Even launching remote JVM debuggers to individual worker nodes cannot scale for big data computing

Spark Program with Transformations

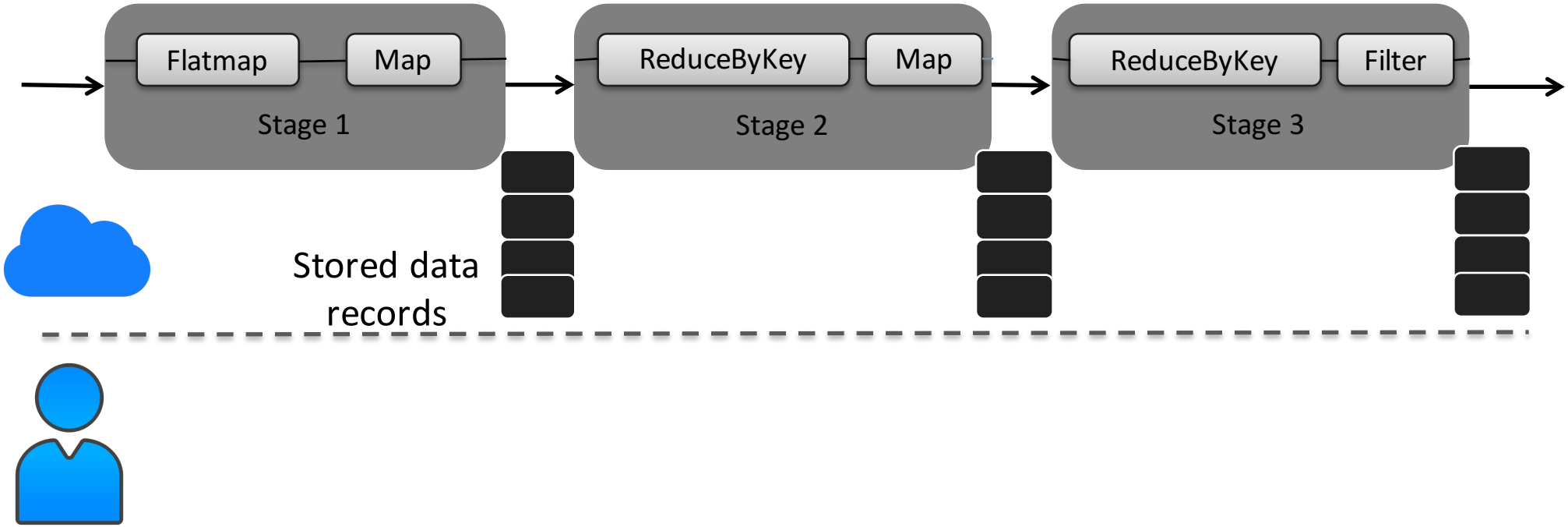
Spark Program Scheduled as Stages



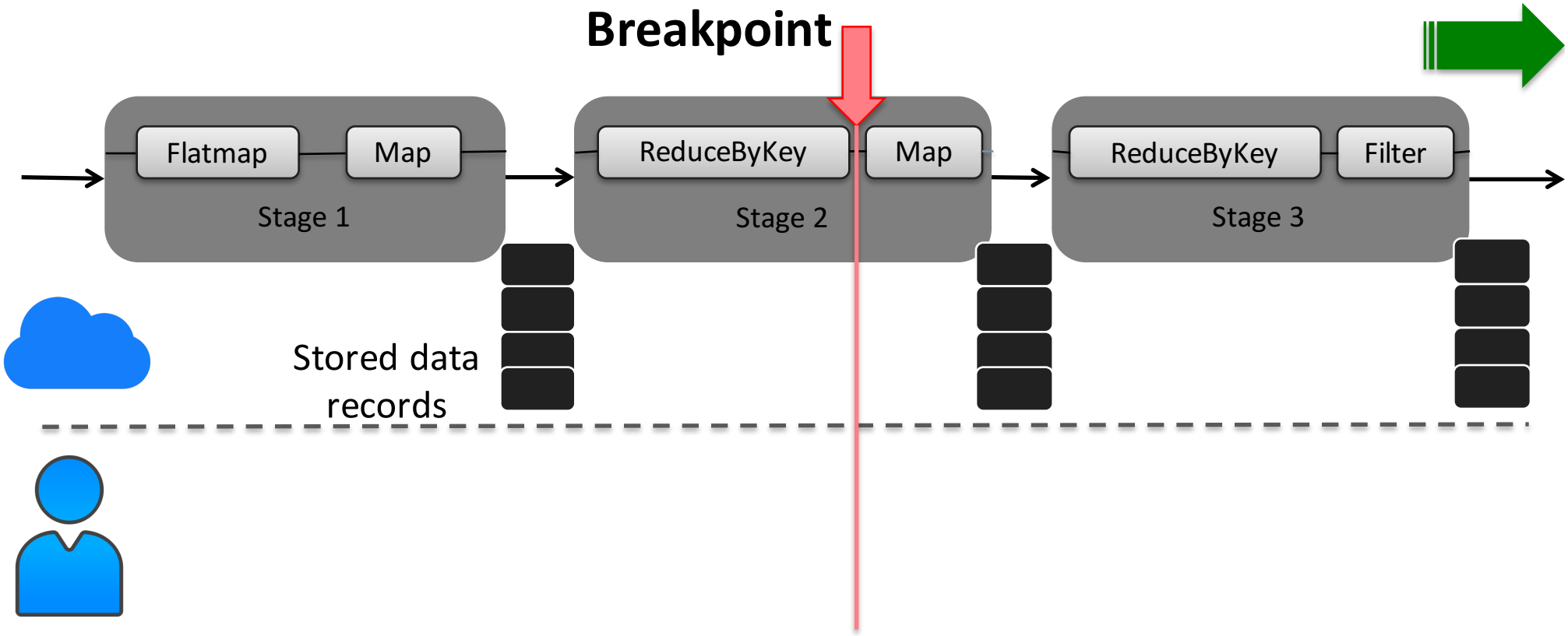
Materialization Points in Spark



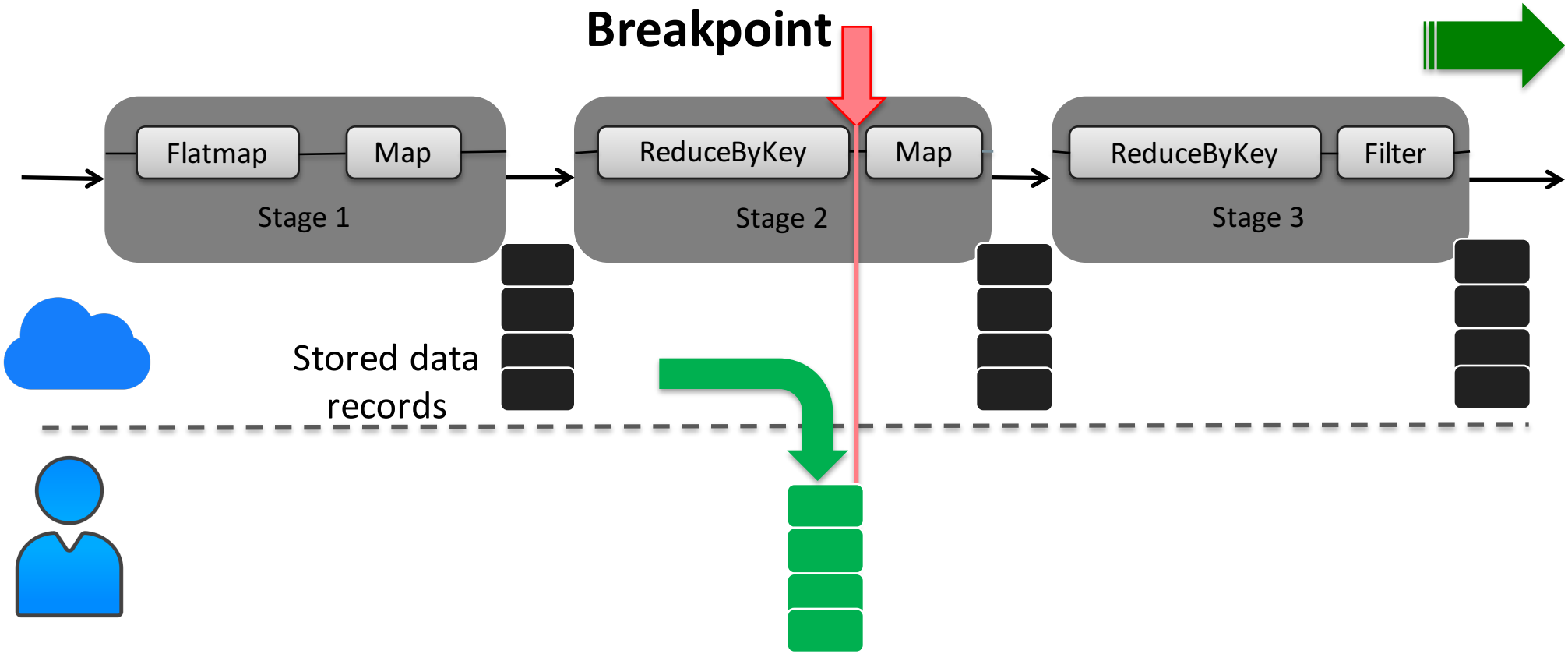
1. Simulated Breakpoint



1. Simulated Breakpoint

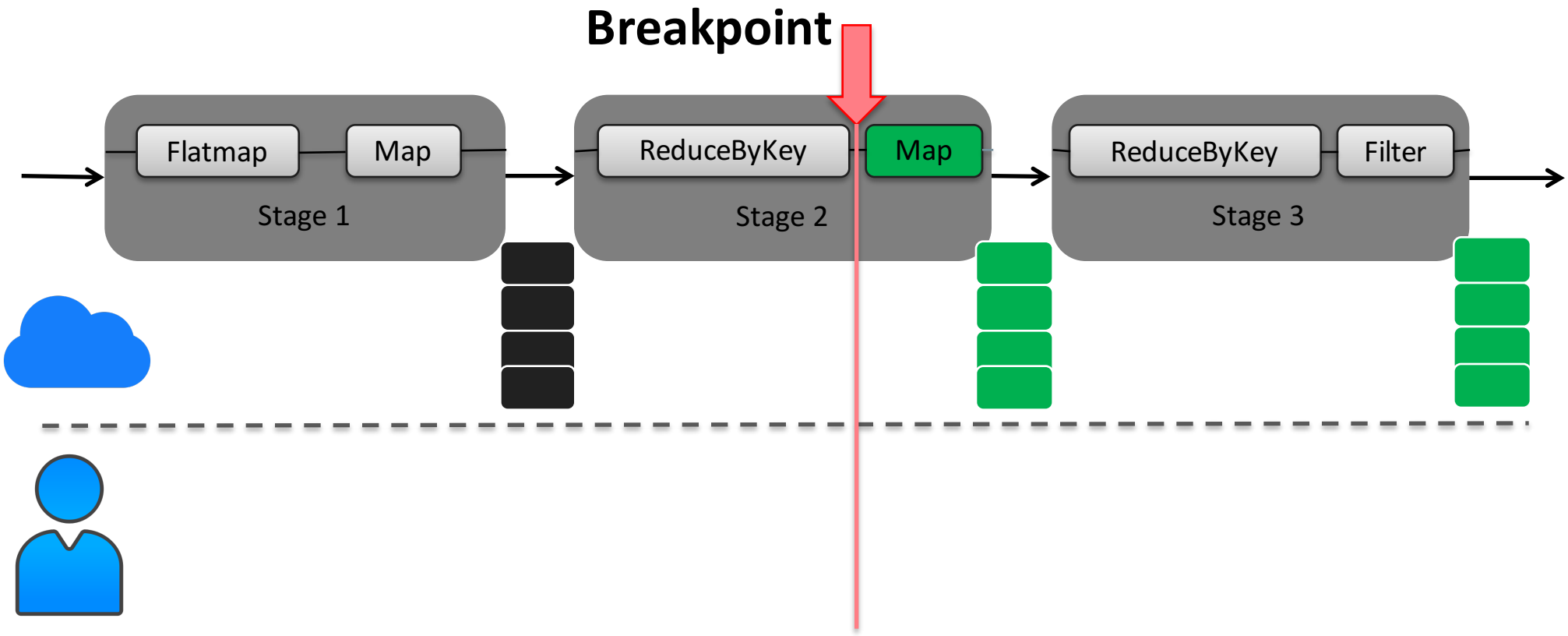


1. Simulated Breakpoint



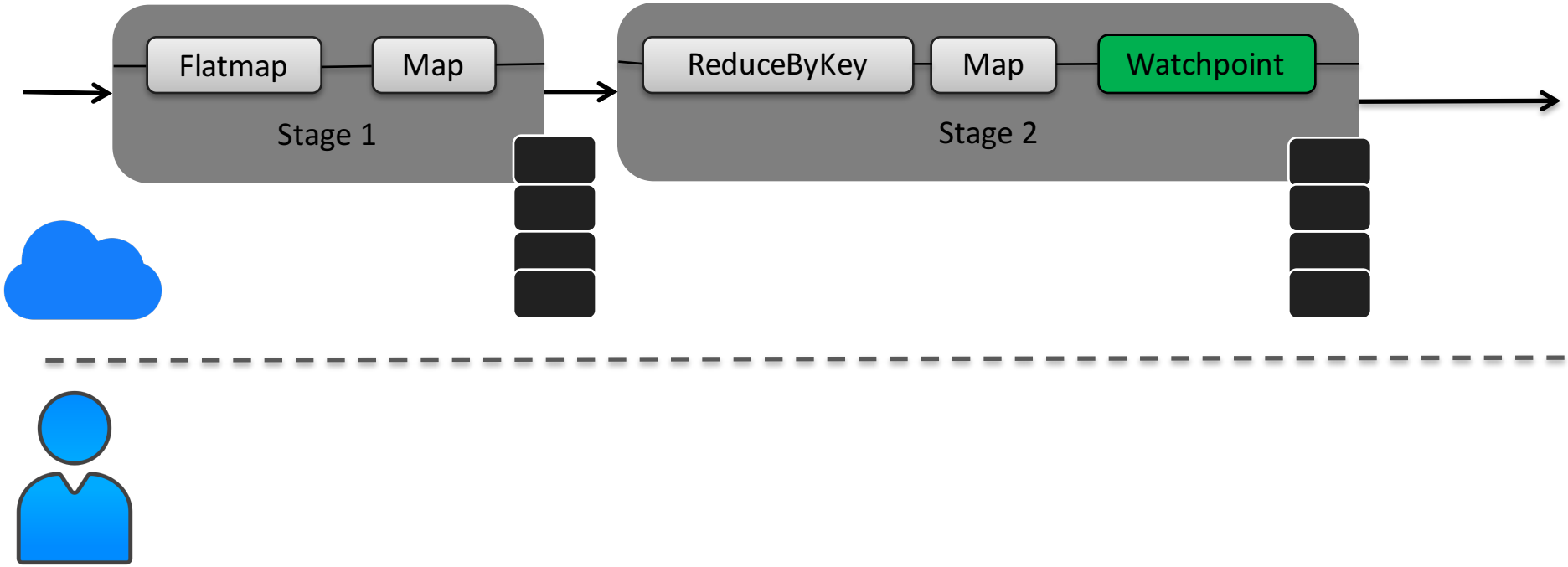
Simulated breakpoint replays computation from the latest materialization point where data is stored in memory

1. Simulated Breakpoint – Realtime Code Fix



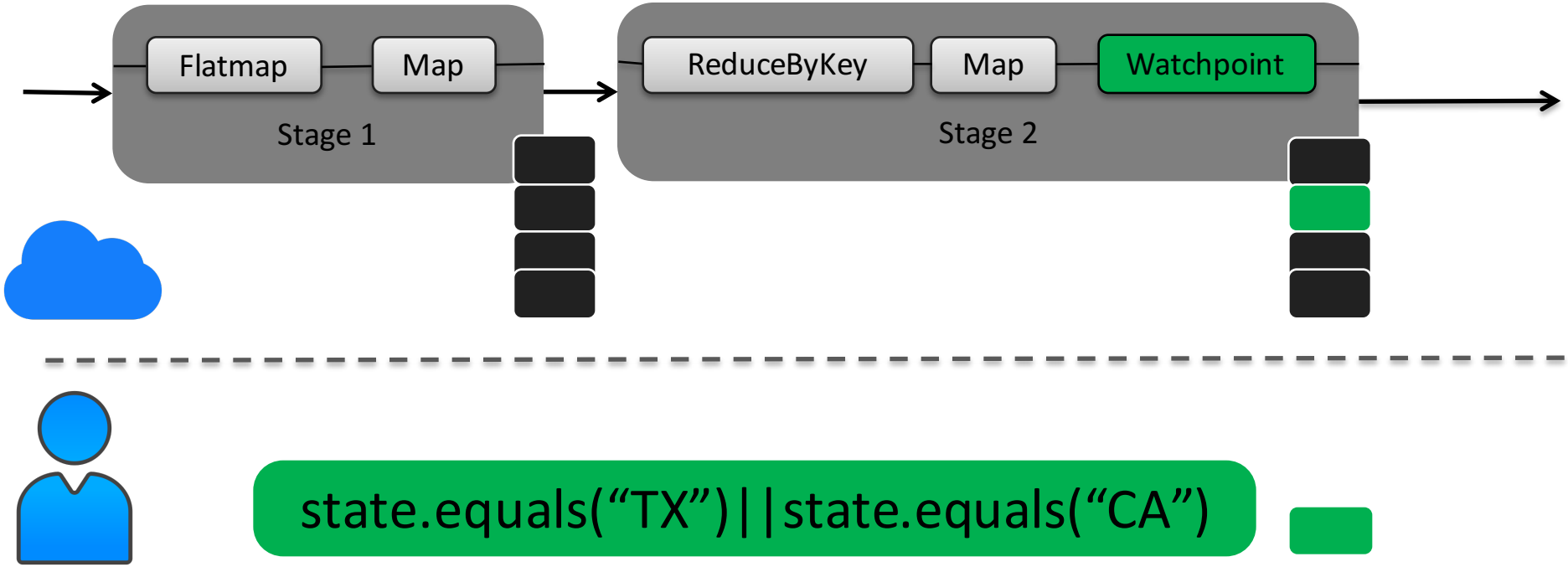
Allow a user to fix code after the breakpoint

2. On-Demand Guarded Watchpoint



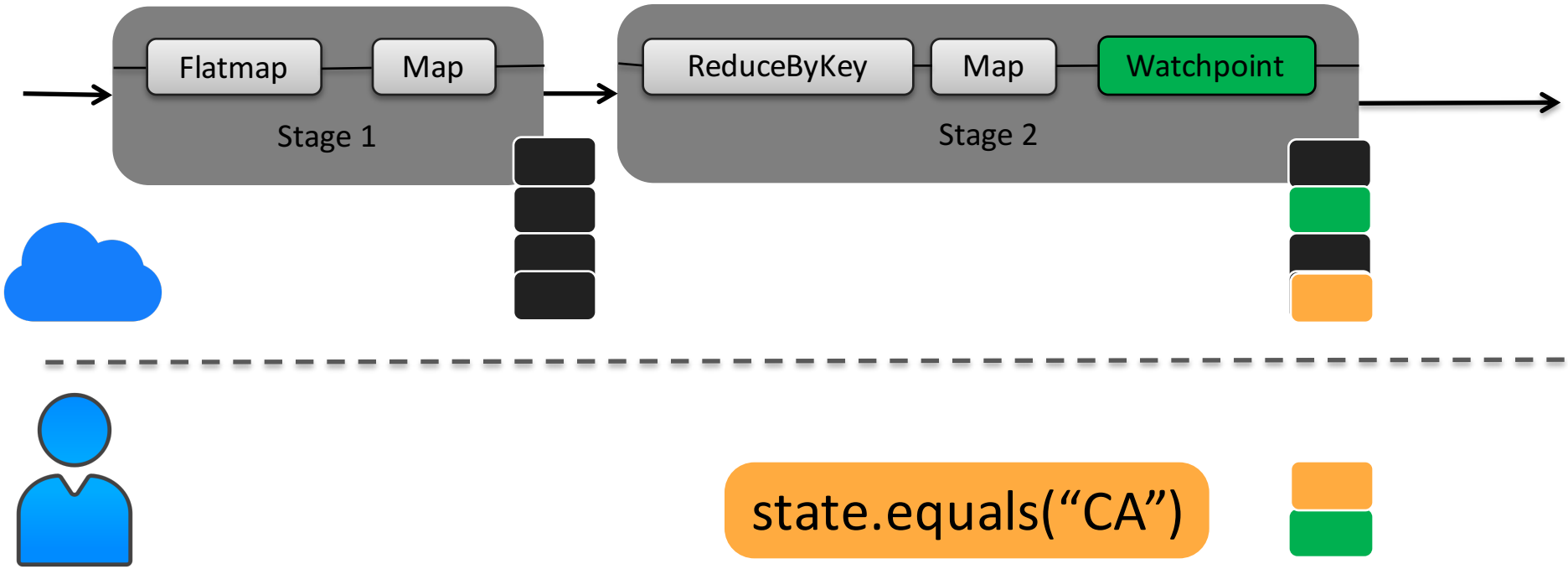
Watchpoint captures individual data records matching a user-provided guard

2. On-Demand Guarded Watchpoint



Watchpoint captures individual data records matching a user-provided guard

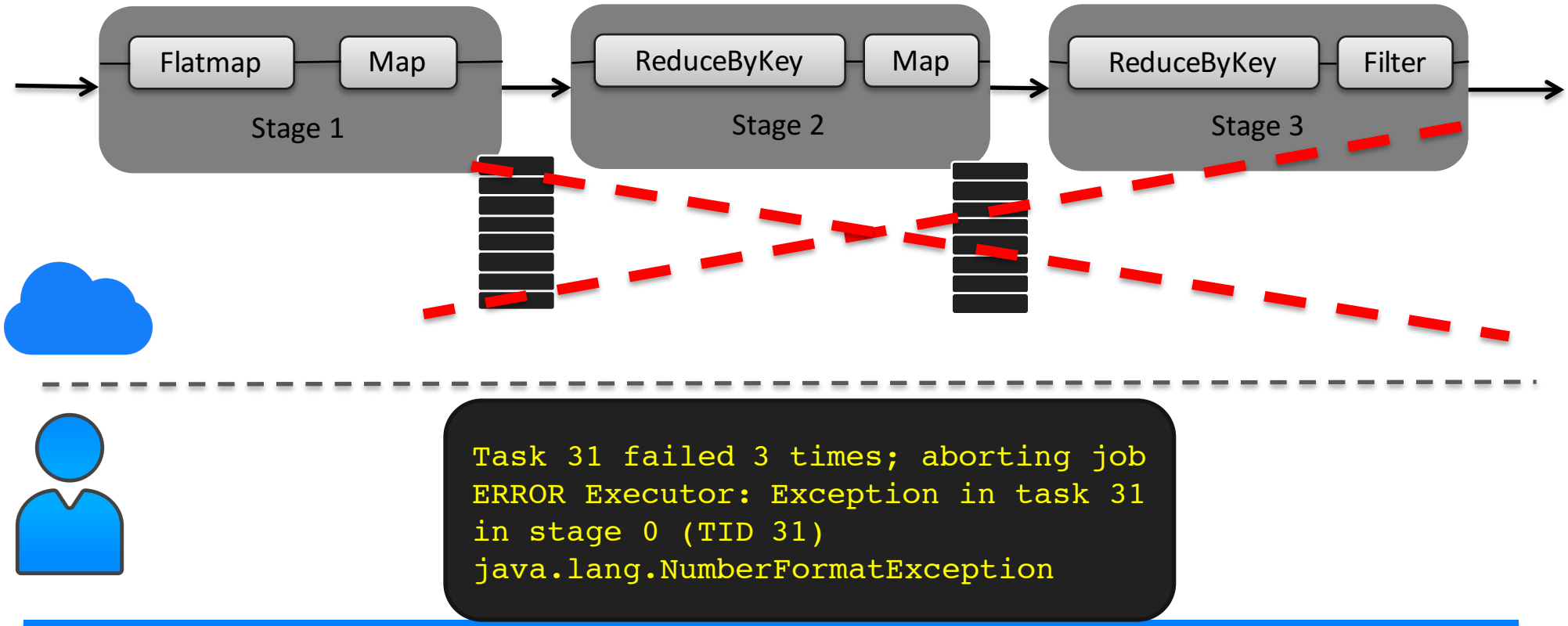
2. On-Demand Guarded Watchpoint



Watchpoint captures individual data records matching a user-provided guard

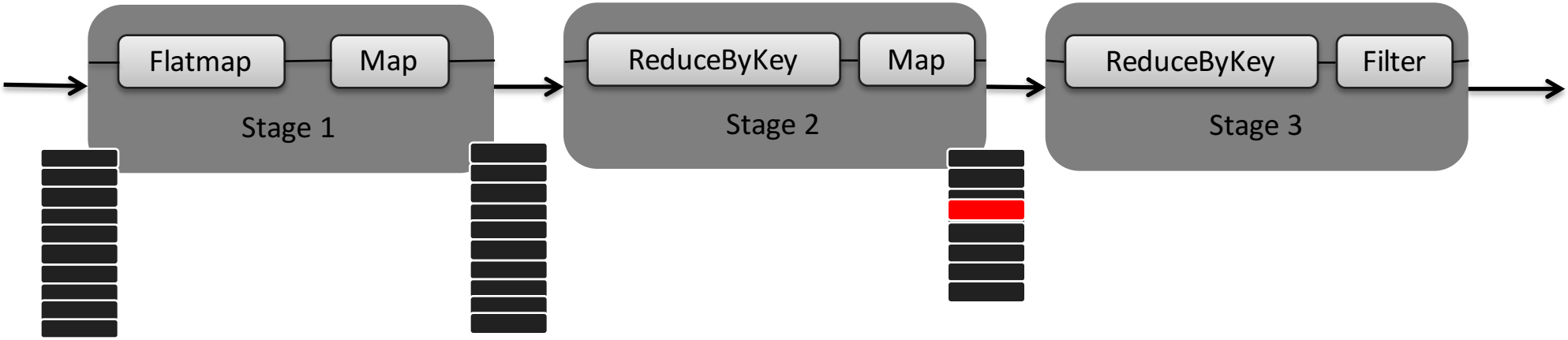
Crash in Apache Spark

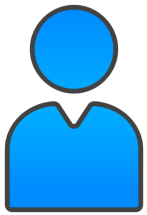
A job failure in Spark throws away the intermediate results of correctly computed stages



To recover to from crash, a user need to find input causing crash and re-execute the whole job.

3. Crash Culprit Identification

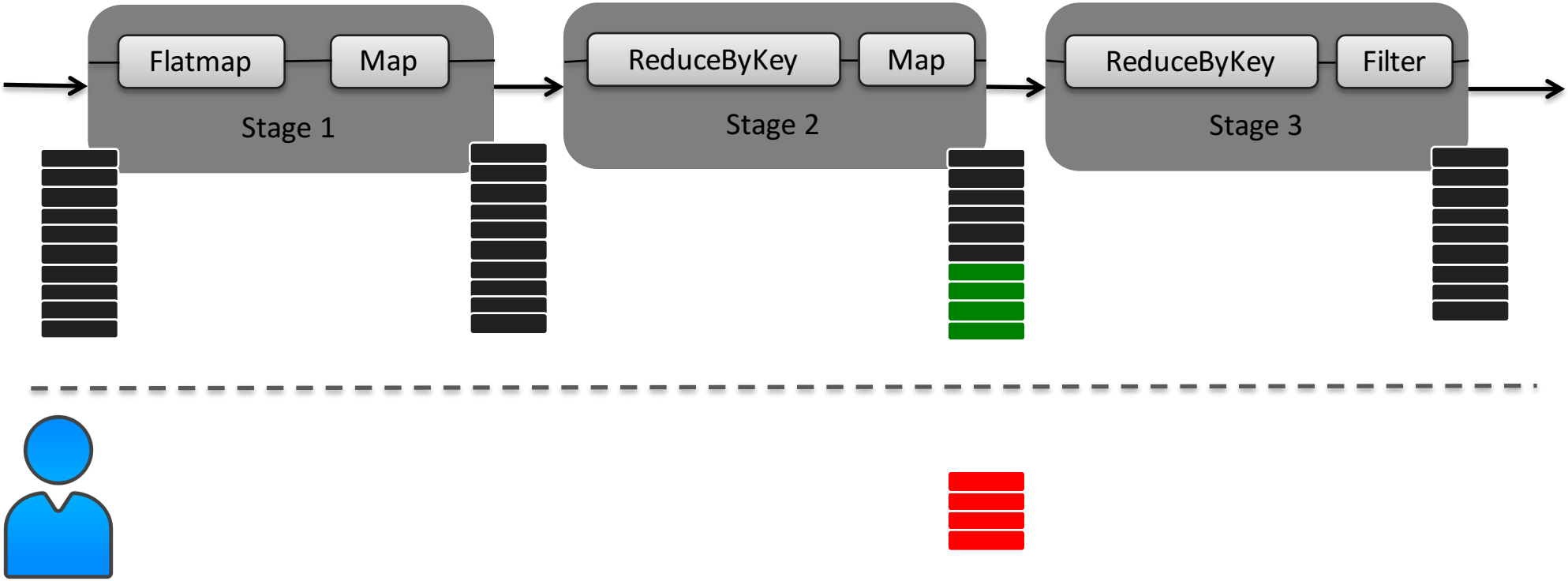




```
Crash occurred at transformation 3
Crashing Record : "Sanders"
ArrayIndexOutofBoundException
Skipping the record.
Continuing processing.
```

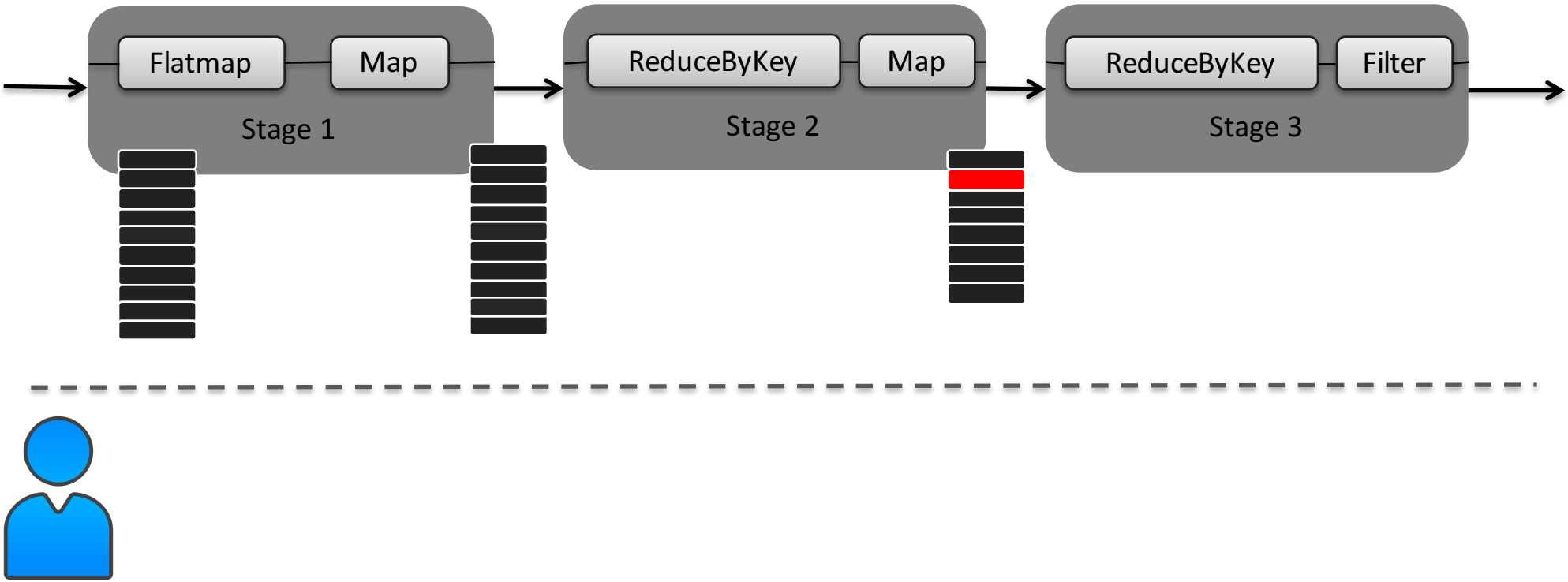
A user can see the crash-causing intermediate record and trace the original inputs leading to the crash.

3. Crash Culprit Remediation



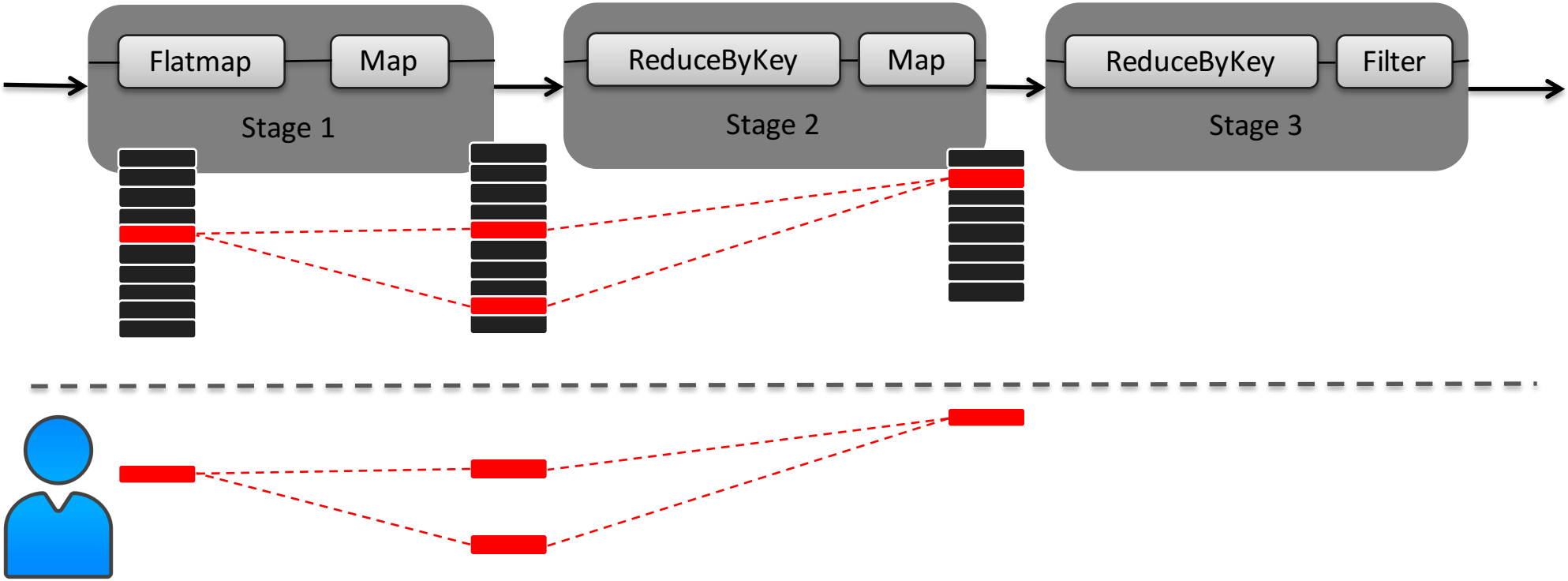
A user can either correct the crashed record, skip the crash culprit, or supply a code fix to repair the crash culprit.

4. Backward and Forward Tracing



A user can also issue tracing queries on intermediate records at realtime

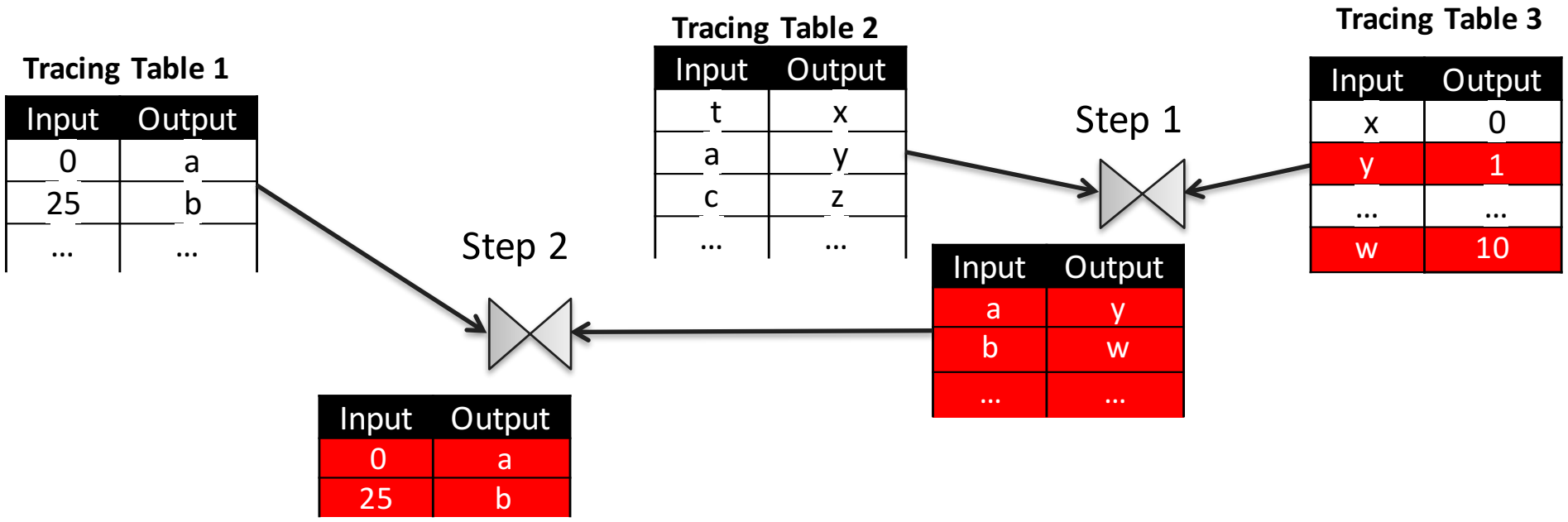
4. Backward and Forward Tracing



A user can also issue tracing queries on intermediate records at realtime

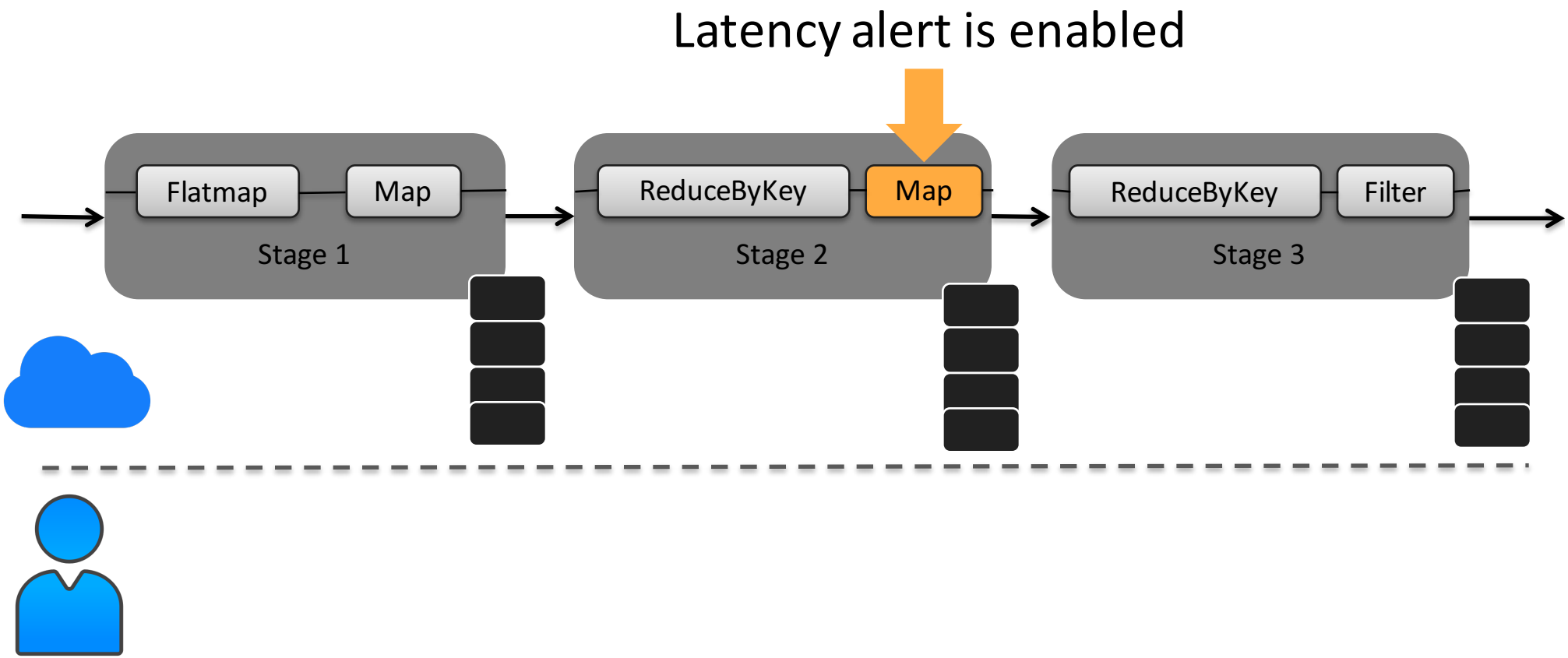
Titian: Data Provenance for Spark [PVLDB2016]

Titian instruments Spark jobs with tracing agents to generate fine grained tracing tables



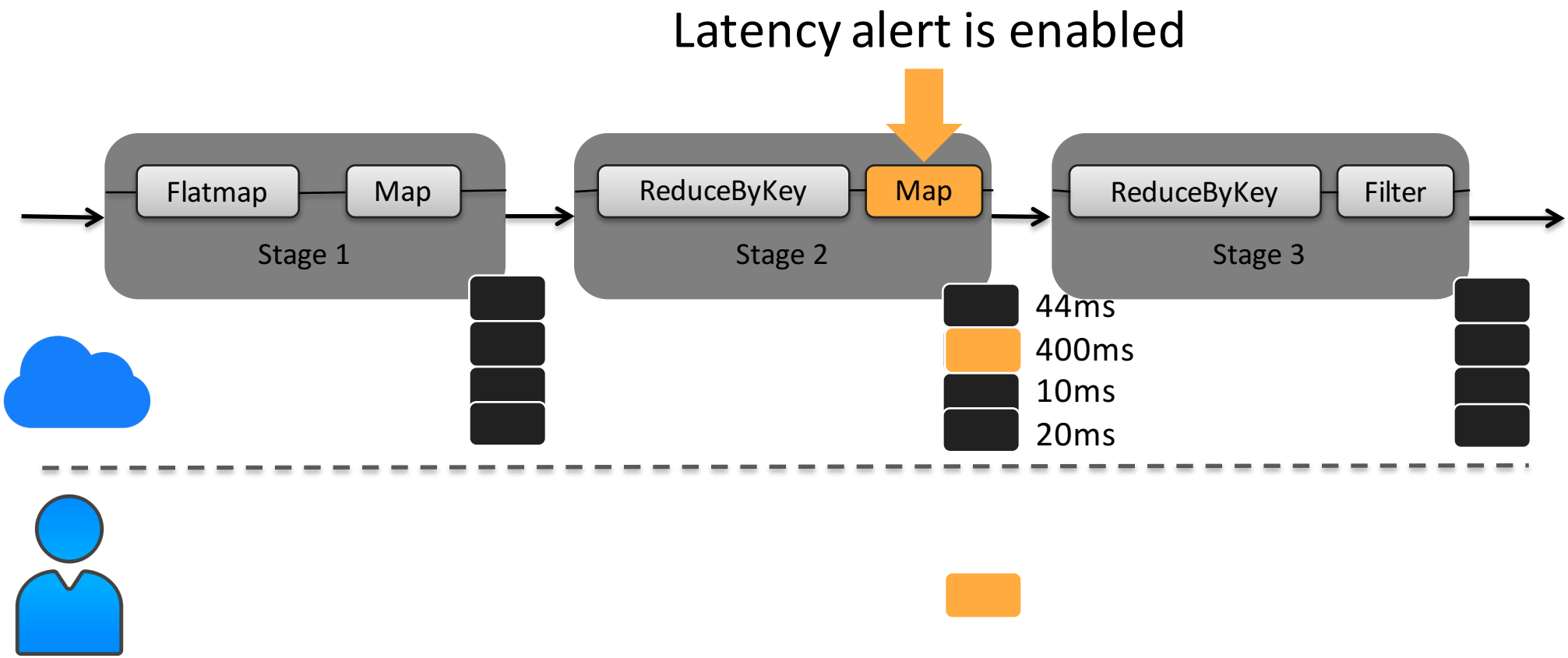
Titian logically reconstructs mapping from output to input records by recursively joining the provenance tables

5. Fine Grained Latency Alert



A latency alert is issued if the processing time is greater than k standard deviations above the moving average

5. Fine Grained Latency Alert

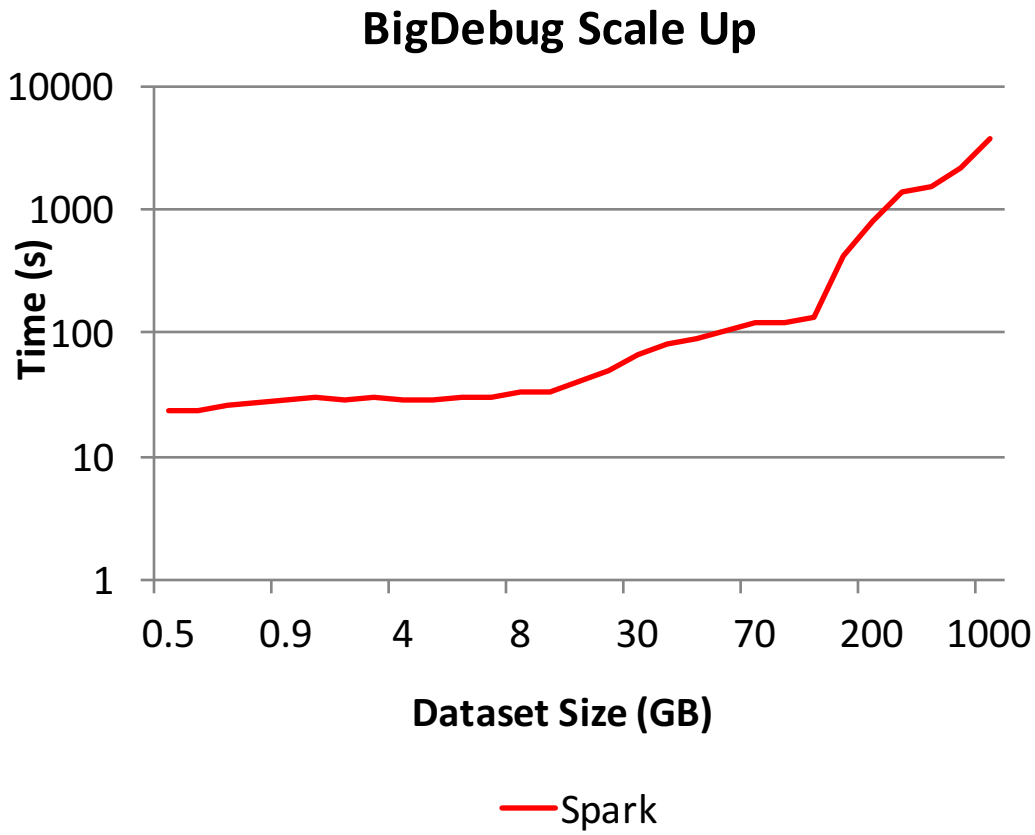


A latency alert is issued if the processing time is greater than k standard deviations above the moving average

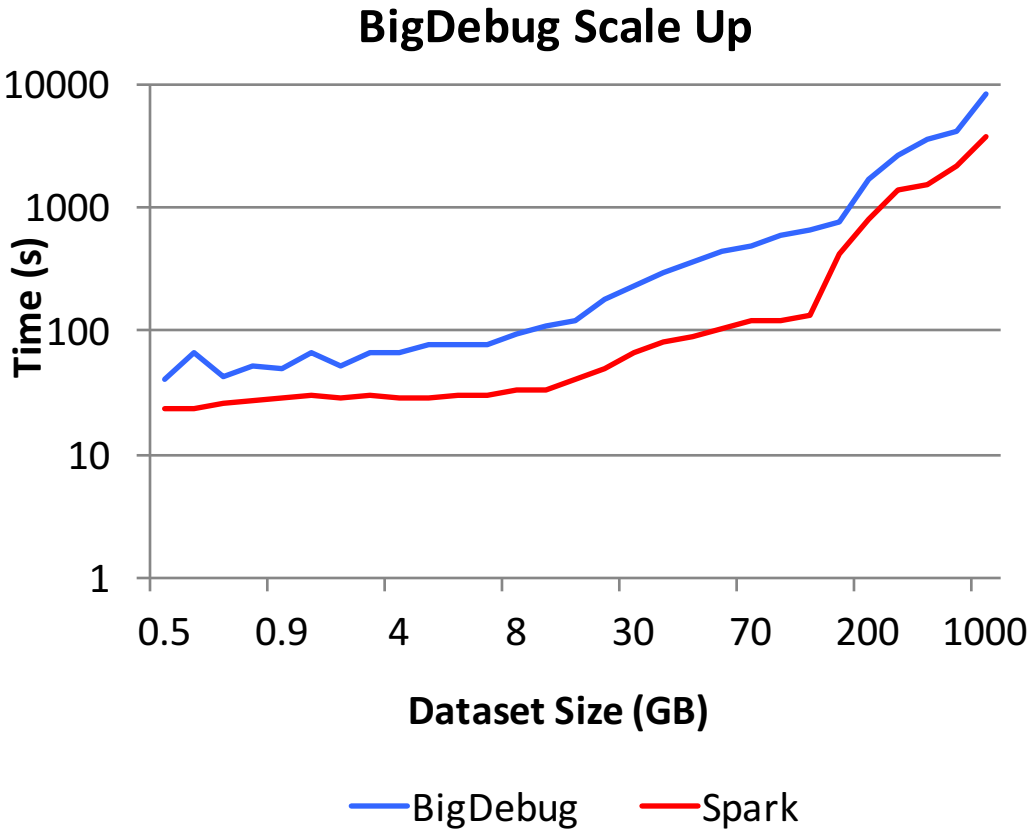
Evaluation

- Q1 : How does BigDebug **scale** to massive data?
- Q2 : What is the performance **overhead** of instrumentation and communication for debugging primitives?
- Q3 : How much **time saving** does BigDebug provide through its runtime crash remediation, in comparison to an existing replay debugger?

Q1 : How does BigDebug scale to massive data?

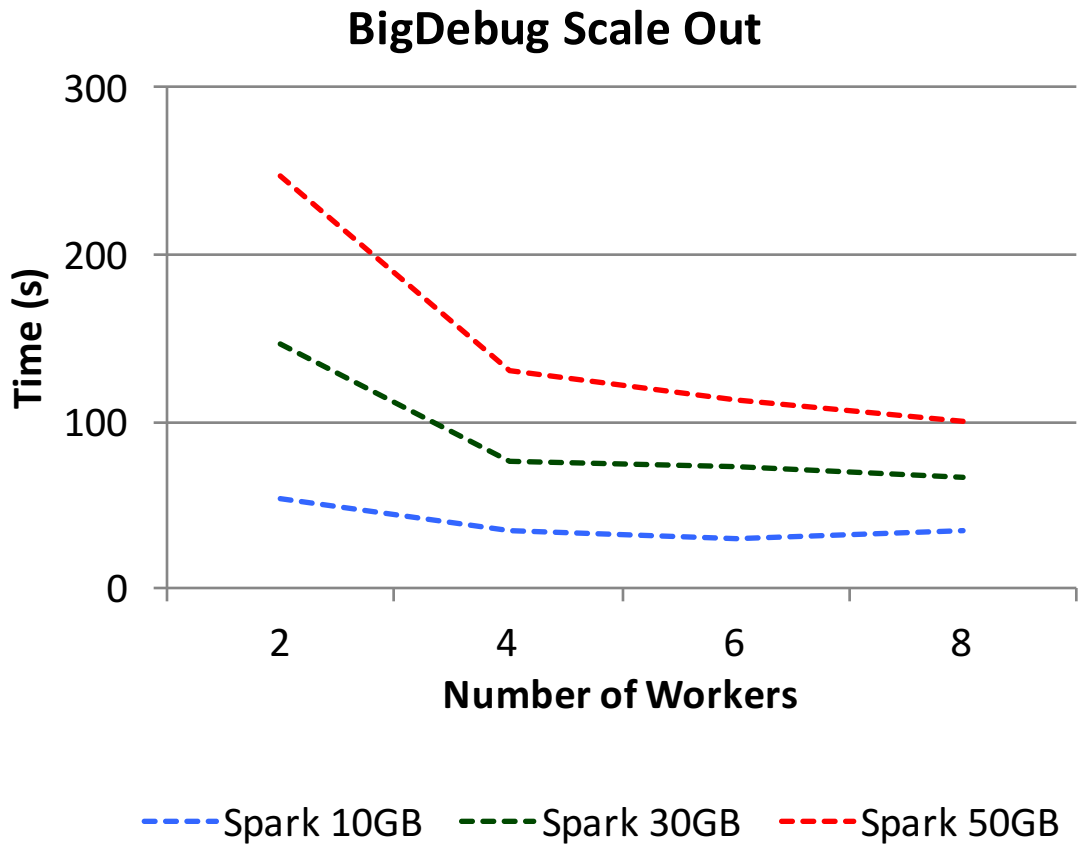


Q1 : How does BigDebug scale to massive data?

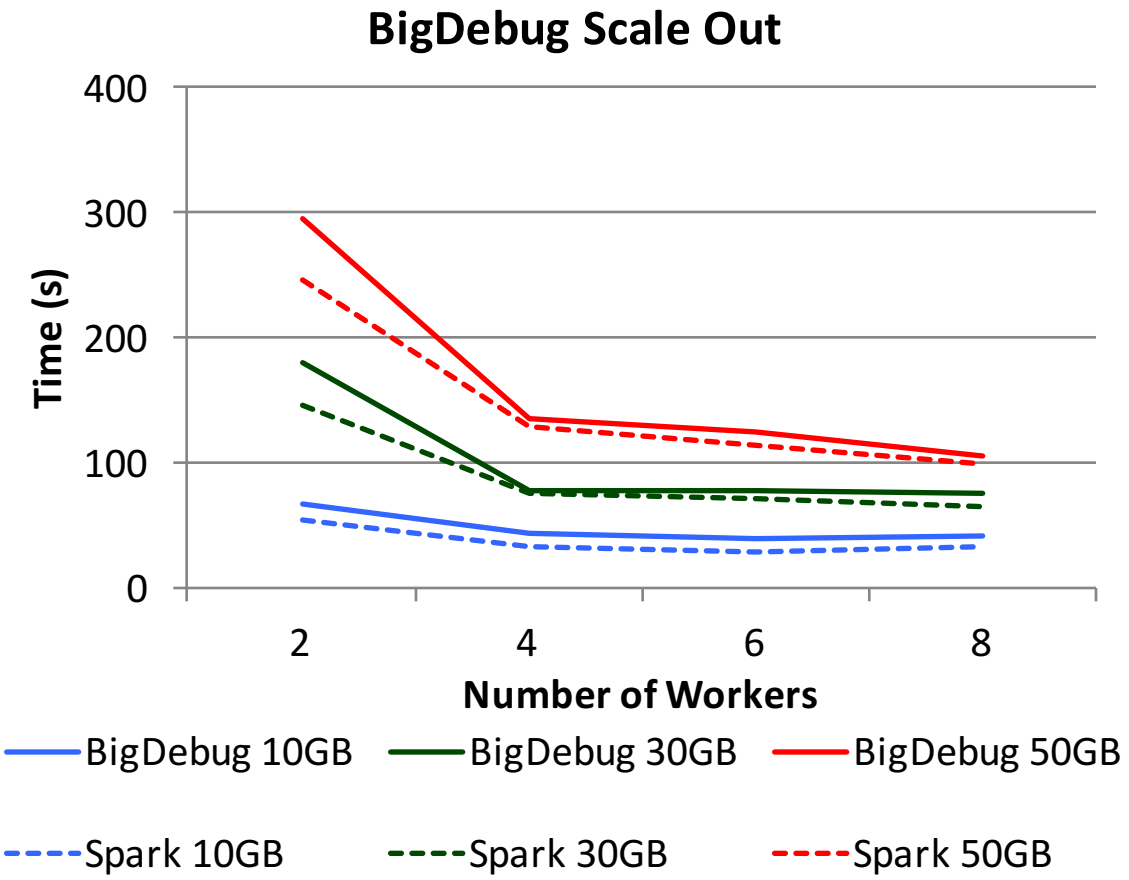


BigDebug retains scale up property of Spark. This property is critical for Big Data processing frameworks

Q1 : How does BigDebug scale to massive data?



Q1 : How does BigDebug scale to massive data?



BigDebug retains scale out property of Spark. This property is critical for Big Data processing frameworks

Q2 : What is the performance overhead of debugging primitives?

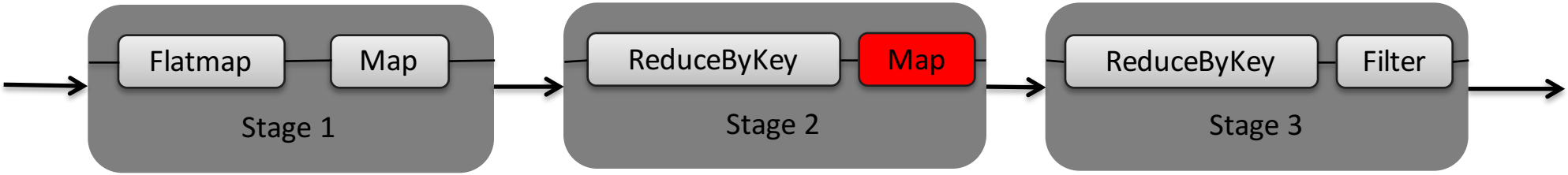
Program	Dataset size (GB)	Max	Max w/o Latency Alert	Watchpoint	Crash Culprit	Tracing
WordCount	0.5 - 1000	2.5X	1.34X	1.09X	1.18X	1.22X
Grep	20 - 90	1.76X	1.07X	1.05X	1.04X	1.05X
PigMix-L1	1 - 200	1.38X	1.29X	1.03X	1.19X	1.24X

Max : All the features of BigDebug are enabled

BigDebug poses at most 2.5X overhead with the maximum instrumentation setting.

Q3 : How much time saving does BigDebug provide when resolving crash?

Suppose that a user wants to skip or correct the crash causing inputs.

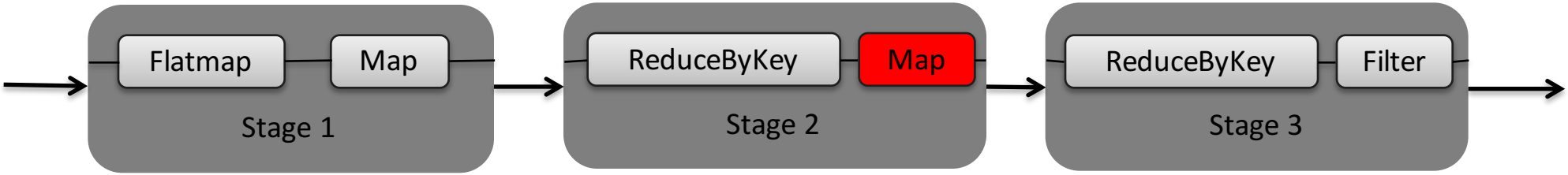


Arthur

The first run crashes

Q3 : How much time saving does BigDebug provide when resolving crash?

Suppose that a user wants to skip or correct the crash causing inputs.



Arthur

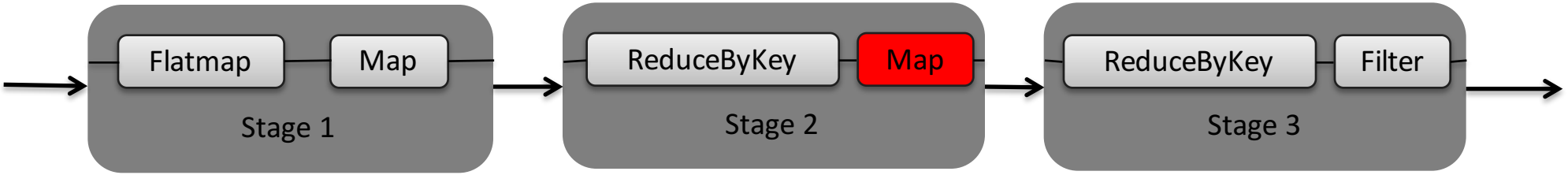
The first run crashes

The second run instruments all records leading to a crash

The third run removes the crash inducing records from the inputs.

Q3 : How much time saving does BigDebug provide when resolving crash?

Suppose that a user wants to skip or correct the crash causing inputs.



Arthur

The first run crashes

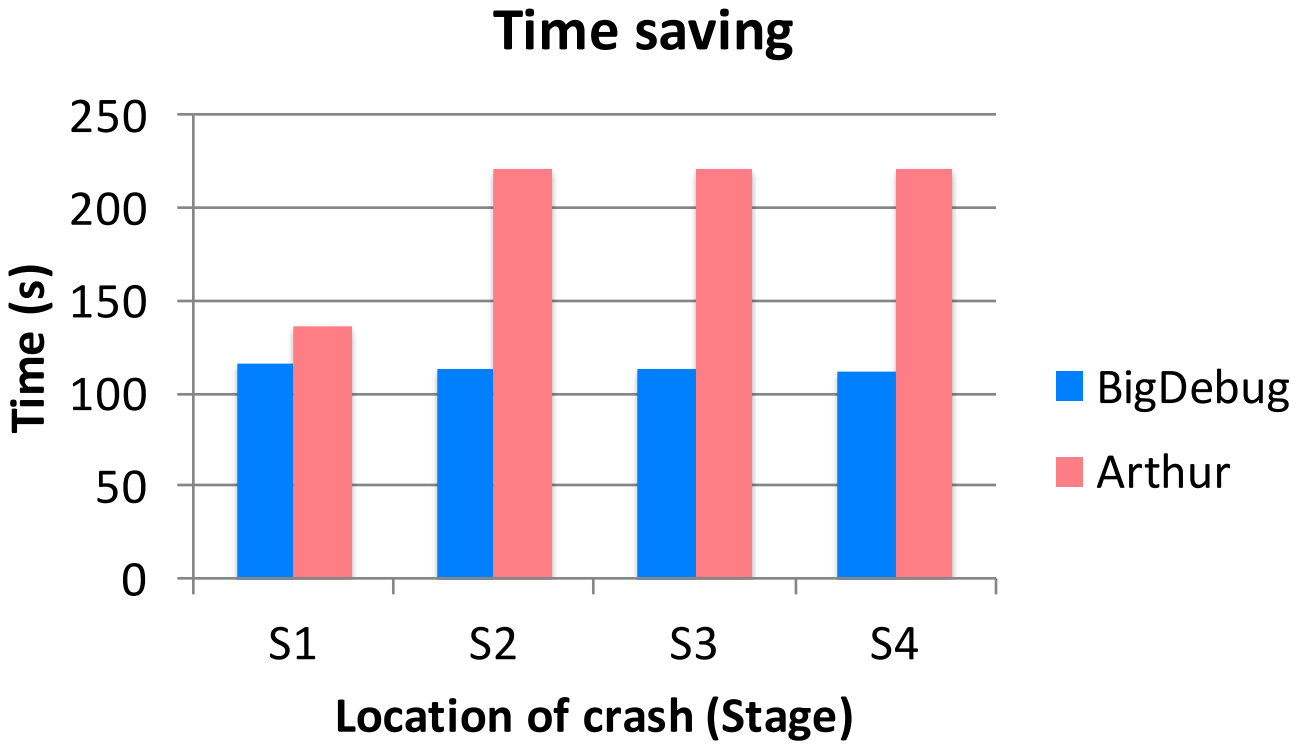
The second run instruments all records leading to a crash

The third run removes the crash inducing records from the inputs.

BigDebug

A single run can detect and remove the crash culprit and resumes the job

Q3 : How much time saving does BigDebug provide?



BigDebug finds a crash inducing record with 100% accuracy and saves upto 100% time saving through runtime crash remediation

Conclusion

- Debugging big data applications is painstaking and expensive
- BigDebug provides interactive debugging primitives for high performance in-memory processing in Spark
- BigDebug offers simulated breakpoints and guarded watchpoints with little performance overhead
- It scales to massive data in the order of terabytes, its record level tracing poses 25% overhead and provides up to 100% time saving
- BigDebug is publically available at https://sites.google.com/site/sparkbigdebug/