Automated Debugging In Data Intensive Scalable Computing Systems

Muhammad Ali Gulzar\(^1\)  Siman Wang\(^{1,2}\)  Miryung Kim\(^1\)

\(^1\)University of California, Los Angeles
\(^2\)Hunan University
Big Data Debugging in the Dark

1. Develop locally
2. Hope it works
3. Run in cloud
4. Bug!
5. Guesswork

Google
Map Reduce
hadoop
Spark
HIVE
Motivating Example

• Alice writes a Spark program that identifies, for each state in the US, the delta between the minimum and the maximum snowfall reading for each day of any year and for any particular year.

<table>
<thead>
<tr>
<th>Zip Code</th>
<th>Date</th>
<th>Snowfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>99504</td>
<td>01/01/1994</td>
<td>245mm</td>
</tr>
<tr>
<td>99504</td>
<td>01/01/1993</td>
<td>85mm</td>
</tr>
<tr>
<td>90031</td>
<td>02/01/1991</td>
<td>0mm</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Problem Definition

- Using a test function, a user can specify incorrect results

```python
def test(key: String, delta: Float) : Boolean = {
    delta < 6000
}
```

Given a test function, the goal is to identify a minimum subset of the input that is able to reproduce the same test failure.
Existing Approach 1: Data Provenance for Spark

It over-approximates the scope of failure-inducing inputs i.e. records in the faulty key-group are all marked as faulty.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary-like search on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Automated Debugging in DISC with BigSift

Input: A Spark Program, A Test Function

Output: Minimum Fault-Inducing Input Records

Data Provenance + Delta Debugging

Test Predicate Pushdown
Prioritizing Backward Traces
Bitmap based Test Memoization
A sample dataflow program

```scala
val sc = new SparkContext(sparkConf)
val input = sc.textFile(logFile)
findDelta(input).collect()

def findDelta(input: RDD): RDD = {
  ...
}
```

Invocation of dataflow program in Apache Spark

Dataflow program that returns the transformed input data
Invoking BigSift’s API

```scala
val sc = new SparkContext(sparkConf)
val bsift = new BigSift(sc, logFile)
bsift.runWithBigSift[_, _](findDelta)
```

BigSift can be used by initiating the `BigSift` object and then invoking the API `runWithBigSift` with the program method.

```scala
def findDelta(input: RDD): RDD = {
  ...
}
```

Dataflow program that returns the transformed input data.
BigSift’s Interactive User Interface

• After invoking BigSift programmatically, a user can interact with BigSift’s UI at port 8989.

• When the program completes, BigSift visualizes the output and reports the execution time as well as input data size.
Defining Test Oracle Function Interactively

- A user can write a predicate to be applied to each final output record to distinguish correct outputs from incorrect.
- BigSift also enables user to choose from a list of pre-defined test predicate functions

```scala
def test(record : Any) : Boolean = {
  //Implement Test function here
  record.asInstanceOf[_,Float]._2 > 6000.0
}
```

Select one of the following test options:

- Explain input records that lead to a minimum output
- Explain input records that lead to a maximum output
- Explain input records that lead to output values not in 5-Sigma range of median
- Explain input records that lead to a NaN or a Null
- Explain input records that lead to output values failing the test predicate in code box

Run BigSift!
Real-time Automated Debugging

- When user submits test predicate, BigSift shows real-time area chart and stream debugging progress from the cloud.
- A user can click on any part of the chart to view sample fault-inducing input records at the selected time.
Live Demonstration
Optimization 1: Test Predicate Pushdown

- **Observation:** During backward tracing, data provenance traces through all partitions even though only a few partitions contain faulty intermediate data.

If applicable, BigSift pushes down the test function to test the output of combiners in order to isolate the faulty partitions.
Optimization 2: Prioritizing Backward Traces

- **Observation**: The same faulty input record may contribute to multiple faulty output due to operators such as Join or Flatmap.

In case of multiple faulty outputs, BigSift overlaps two backward traces to minimize the scope of fault-inducing input records.
Optimization 3: Bitmap Based Test Memoization

- **Observation:** Delta debugging may try running a program on the same subset of input redundantly.

- BigSift leverages bitmap to compactly encode the offsets of original input to refer to an input subset.

We use a bitmap based test memoization technique to avoid redundant testing of the same input dataset.
Evaluation: Performance Improvement

<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Fault</th>
<th>Running Time (sec)</th>
<th>Debugging Time (sec)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original Job</td>
<td>DD</td>
<td>BigSift</td>
</tr>
<tr>
<td>Movie Histogram Code</td>
<td>Code</td>
<td>56.2</td>
<td>232.8</td>
<td>17.3</td>
</tr>
<tr>
<td>Inverted Index Code</td>
<td>Code</td>
<td>107.7</td>
<td>584.2</td>
<td>13.4</td>
</tr>
<tr>
<td>Rating Histogram Code</td>
<td>Code</td>
<td>40.3</td>
<td>263.4</td>
<td>16.6</td>
</tr>
<tr>
<td>Sequence Count Code</td>
<td>Code</td>
<td>356.0</td>
<td>13772.1</td>
<td>208.8</td>
</tr>
<tr>
<td>Rating Frequency Code</td>
<td>Code</td>
<td>77.5</td>
<td>437.9</td>
<td>14.9</td>
</tr>
<tr>
<td>College Student Data</td>
<td>Data</td>
<td>53.1</td>
<td>235.3</td>
<td>31.8</td>
</tr>
<tr>
<td>Weather Analysis Data</td>
<td>Data</td>
<td>238.5</td>
<td>999.1</td>
<td>89.9</td>
</tr>
<tr>
<td>Transit Analysis Code</td>
<td>Code</td>
<td>45.5</td>
<td>375.8</td>
<td>20.2</td>
</tr>
</tbody>
</table>

BigSift provides up to a 66X speed up in isolating the precise fault-inducing input records, in comparison to the baseline DD.
## Evaluation: Debugging Time vs. Original Job Time

<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Fault</th>
<th>Running Time (sec)</th>
<th>Debugging Time (sec)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original Job</td>
<td>DD</td>
<td>BigSift</td>
</tr>
<tr>
<td>Movie Histogram</td>
<td>Code</td>
<td>56.2</td>
<td>232.8</td>
<td>17.3</td>
</tr>
<tr>
<td>Inverted Index</td>
<td>Code</td>
<td>107.7</td>
<td>584.2</td>
<td>13.4</td>
</tr>
<tr>
<td>Rating Histogram</td>
<td>Code</td>
<td>40.3</td>
<td>263.4</td>
<td>16.6</td>
</tr>
<tr>
<td>Sequence Count</td>
<td>Code</td>
<td>356.0</td>
<td>13772.1</td>
<td>208.8</td>
</tr>
<tr>
<td>Rating Frequency</td>
<td>Code</td>
<td>77.5</td>
<td>437.9</td>
<td>14.9</td>
</tr>
<tr>
<td>College Student</td>
<td>Data</td>
<td>53.1</td>
<td>235.3</td>
<td>31.8</td>
</tr>
<tr>
<td>Weather Analysis</td>
<td>Data</td>
<td>238.5</td>
<td>999.1</td>
<td>89.9</td>
</tr>
<tr>
<td>Transit Analysis</td>
<td>Code</td>
<td>45.5</td>
<td>375.8</td>
<td>20.2</td>
</tr>
</tbody>
</table>

On average, BigSift takes 62% less time to debug a single faulty output than the time taken for a single run on the entire data.
Evaluation: Debugging Time vs. Original Job Time

On average, BigSift takes 62% less time to debug a single faulty output than the time taken for a single run on the entire data.
Conclusion

• BigSift is the first piece of work in automated debugging of big data analytics in DISC.

• It provides up to **66X speed up** in debugging time over baseline Delta Debugging.

• In our evaluation we have observed that, on average, BigSift finds the faulty input in **62% less** than the original job execution time.
Questions?