Interactive Debugging for Big Data Analytics

Muhammad Ali Gulzar, Xueyuan Han, Matteo Interlandi, Shaghayegh Mardani, Sai Deep Tetali, Tyson Condie, Todd Millstein, Miryung Kim
University of California, Los Angeles
Debugging Big Data Analytics

• Today’s platforms lack debugging support
  – Programs (i.e., queries, jobs) are batch executed / black boxes
  – Errors reflect low-level details (e.g., task id?!?) not relevant to the logical bug
  – Long program execution time => long development cycles

• What do programmers do?
  – Trial and error debugging on sample data
  – Post-mortem analysis of error logs
  – Analyze physical view of the execution (a job id, failed node, etc).
“I would like to understand the flow of control through the Spark source code on the worker nodes when I submit my application ... I am assuming I should setup Spark on Eclipse ... to enable stepping through Spark source code on the worker nodes.”
After a year, still no good answers!

Add the relevant spark jars to the eclipse project. And then set the master in the code. And now 

Have you tried passing remote debug parameters to worker JVM? I think its something like 

You could run the Spark application in local mode if you just need to debug the logic of your 

When you run a spark application on yarn, there is an option like this:

You can add it to yarn-env.sh and remote debugging will be available via port 5455.

If you use spark in standalone mode, I believe this can help:

export SPARK_JAVA_OPTS=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005

share improve this answer

edited Jul 20 at 21:23

answered Jul 3 at 9:14

Cleb
1,510 • 2 • 12 • 24

user3504158
BigDebug Project Overview

BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark [ICSE 2016]

Simulated Breakpoint
On-Demand Watchpoint
Crash Culprit Remediation
Forward Backward Tracing

Titian: Data Provenance for Fine-Grained Tracing [PVLDB 2016]

Vega: Incremental Computation for Interactive Debugging [Under Review]
Example Query Development Session

• Dataset: NYC Open Data Project
  – Calls to non-emergency service centers
  – Dataset contains call records for 2010-2015
    • Record contents: call time, agency, caller location, etc.

• Query: Identify the agencies that received the most calls during busy hours
  – E.g., busy hour if number of calls > 10,000
case class Calls(id:String, hour:Int, agency: String, ...)
format = new SimpleDateFormat("M/d/y h:m:s a")
input = sc.textFile("hdfs://...")
calls = input.map(_.split(","))
    .map(r => Calls(r(0), format.parse(r(1)).getHours, r(2),...))
calls.registerTempTable("calls")
hist = sqlContext.sql("SELECT agency, count(*) FROM calls JOIN (
    SELECT hour FROM calls GROUP BY hour
    HAVING count(*) > 100000
) counts ON calls.hour = counts.hour
GROUP BY agency")
hist.show()
Extract Dataset from HDFS
Transform it into a DataFrame (i.e., table)
Load it into Spark SQL

case class Calls(id:String, hour:Int, agency:String,...)
    format = new SimpleDateFormat("M/d/y h:m:s a")
    input = sc.textFile("hdfs://...")
    calls = input.map(_.split(","))
        .map(r => Calls(r(0), format.parse(r(1)).getHours, r(2),...))
calls.registerTempTable("calls")
hist = sqlContext.sql("SELECT agency, count(*)
    FROM calls
    JOIN (SELECT hour
    FROM calls
    GROUP BY hour
    HAVING count(*) > 100000)
    counts
    ON calls.hour = counts.hour
    GROUP BY agency")
hist.show()
Express Query in Spark SQL

case class Calls(id:String, hour:Int, agency:String,...)
    format = new SimpleDateFormat("M/d/y h:m:s a")
    input = sc.textFile("hdfs://...")
    calls = input.map(_.split(","))
        .map(r => Calls(r(0), format.parse(r(1)).getHours, r(2),...))
    calls.registerTempTable("calls")
    hist = sqlContext.sql("SELECT agency, count(*) FROM calls JOIN (SELECT hour FROM calls GROUP BY hour HAVING count(*) > 100000) counts ON calls.hour = counts.hour GROUP BY agency")
    hist.show()
Debugging Query Results

• Analyst observes some unexpected results
  – Agencies that should not appear
    • e.g., Brooklyn Public Library
  – Expected agencies that should appear
    • e.g., NYPD, NYFD

• Titian support for query triage
  – Analyst can trace back from outlier results to contributing data at some intermediate stage
  – Analyst can execute queries against intermediate data leading to outlier results
Query Triage with Titian

- Intermediate results for subquery
  - Trace back to subquery and show distribution of calls per hour
  - On intermediate data leading to outlier results

SELECT hour, count(*)
FROM calls
GROUP BY hour

![Histogram showing significant skew in midnight hour=0]
Identify Bug and Revise the Query

• The Bug
  – System assigns default value hour=0 for...
  – Calls that did not log a time

• Possible course of action
  – Filter out calls assigned to hour=0

```
SELECT agency, count(*)
FROM calls
JOIN (SELECT hour
       FROM calls
       WHERE hour != 0
       GROUP BY hour
       HAVING count(*) > 100000 ) counts
ON calls.hour = counts.hour
GROUP BY agency
```

Introduce predicate that filters out midnight hour
Vega: Re-execute revised Query

• Vega materializes intermediate stage results – i.e., The previous subquery result is saved

• Vega Query Rewriter leverages this to rewrite the query into...

```
SELECT agency, count(*)
FROM calls
JOIN counts
WHERE counts.hour != 0
ON calls.hour = counts.hour
GROUP BY agency
```
Vega: Modified Query Evaluation

• Execute an incremental join
  – “Diff” records specify changes in the (join) result
  – For this example, we incrementally remove all records for hour 0 from join and final aggregation results

• Vega Optimizer Results

  Consequence: over an order-of-magnitude runtime improvement
Automated Isolation of Failure-Inducing Inputs for Big Data Analytics

• When a program fails, a user may want to investigate a subset of the original input inducing a crash, a failure, or a wrong outcome.

• **Delta Debugging [Zeller 1999]**
  – Well known debugging algorithm for minimizing failure-inducing inputs
  – Requires multiple runs to isolate failure-inducing inputs
First we run the test to find the failure inducing input dataset
Background: Delta Debugging

[Zeller, FSE 1999]

First, we run the test to find the failure inducing input dataset
Background: Delta Debugging

[Zeller, FSE 1999]

Second, we split the failing input data
Background: Delta Debugging

[Zeller, FSE 1999]
Background: Delta Debugging

[Zeller, FSE 1999]
Background: Delta Debugging

[Zeller, FSE 1999]
Scalable Automated Isolation of Failure-Inducing Inputs

• Leverage data provenance to reduce search space
  – Avoid costly executions on data not relevant to the bug
• Leverage Vega optimize subsequent runs.
Conclusion

• **BigDebug Project**
  – Debugging Primitives for Interactive Big Data Processing in Apache Spark
  – [https://sites.google.com/site/sparkbigdebug/](https://sites.google.com/site/sparkbigdebug/)

• **Titian: Interactive Data Provenance**
  – Supports trace back queries from a set of results
  – Execution replay from an intermediate point

• **Vega: Optimizing modified query execution**
  – Novel query rewrite mechanism that pushes changes backwards to save work
  – Incremental evaluation that operates on data changes induced by query modifications