Programming and Debugging Large-Scale Data Processing Workflows

Christopher Olston and many others

Yahoo! Research
Context

• Elaborate processing of large data sets
e.g.:
  • web search pre-processing
  • cross-dataset linkage
  • web information extraction
Context

storage & processing

- workflow manager
  *e.g. Nova*

- dataflow programming framework
  *e.g. Pig*

- distributed sorting & hashing
  *e.g. Map-Reduce*

- scalable file system
  *e.g. GFS*

Debugging aides:

- Before: example data generator
- During: instrumentation framework
- After: provenance metadata manager

Overview

Detail:
Inspector Gadget
Pig: A High-Level Dataflow Language and Runtime for Hadoop

Web browsing sessions with “happy endings.”

Visits = load ‘/data/visits’ as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url), time;

Pages = load ‘/data/pages’ as (url, pagerank);

VP = join Visits by url, Pages by url;
UserVisits = group VP by user;
Sessions = foreach UserVisits generate flatten(FindSessions(*));
HappyEndings = filter Sessions by BestIsLast(*);

store HappyEndings into '/data/happy_endings' ;
vs. map-reduce: less code!

“The [Hofmann PLSA E/M] algorithm was implemented in pig in 30-35 lines of pig-latin statements. Took a lot less compared to what it took in implementing the algorithm in Map-Reduce Java. Exactly that's the reason I wanted to try it out in Pig. It took 3-4 days for me to write it, starting from learning pig.”

-- Prasenjit Mukherjee, Mahout project
vs. SQL:

step-by-step style;
lower-level control

"I much prefer writing in Pig [Latin] versus SQL. The step-by-step method of creating a program in Pig [Latin] is much cleaner and simpler to use than the single block method of SQL. It is easier to keep track of what your variables are, and where you are in the process of analyzing your data."

-- Jasmine Novak, Engineer, Yahoo!

"PIG seems to give the necessary parallel programming construct (FOREACH, FLATTEN, COGROUP .. etc) and also give sufficient control back to the programmer (which purely declarative approach like [SQL on top of Map-Reduce] doesn’t)."

-- Ricky Ho, Adobe Software
Conceptually:
A Graph of Data Transformations

Find users who tend to visit “good” pages.

Load Visits(user, url, time)
Load Pages(url, pagerank)
Transform to (user, Canonicalize(url), time)
Join url = url
Group by user
Transform to (user, Average(pagerank) as avgPR)
Filter avgPR > 0.5
"ILLUSTRATE lets me check the output of my lengthy batch jobs and their custom functions without having to do a lengthy run of a long pipeline. [This feature] enables me to be productive."

-- Russell Jurney, LinkedIn
(Naïve Algorithm)

1. **Load** Visits(user, url, time)
2. **Transform** to (user, Canonicalize(url), time)
3. **Join** url = url
4. **Group** by user
5. **Transform** to (user, Average(pagerank) as avgPR)
6. **Filter** avgPR > 0.5

Data:
- (Amy, cnn.com, 8am)
- (Amy, http://www.snails.com, 9am)
- (Fred, www.snails.com/index.html, 11am)
- (Amy, www.cnn.com, 8am)
- (Amy, www.snails.com, 9am)
- (Fred, www.snails.com, 11am)
- (www.youtube.com, 0.9)
- (www.frogs.com, 0.4)
Pig Project Status

• Productized at Yahoo (~12-person team)
  – 1000s of jobs/day
  – 70% of Hadoop jobs
• Open-source (the Apache Pig Project)
• Offered on Amazon Elastic Map-Reduce
• Used by LinkedIn, Twitter, Yahoo, ...
Next: NOVA

Debugging aides:
- Before: example data generator
- During: instrumentation framework
- After: provenance metadata manager
Why a Workflow Manager?

• **Modularity:** a workflow connects N dataflow modules
  – Written independently, and re-used in other workflows
  – Scheduled independently

• **Optimization:** optimize across modules
  – Share read costs among side-by-side modules
  – Pipeline data between end-to-end modules

• **Continuous processing:** push new data through
  – Selective re-running
  – Incremental algorithms (“view maintenance”)

• **Manageability:** help humans keep tabs on execution
  – Alerts
  – Metadata (e.g. data provenance)
Example Workflow
Data Passes Through Many Sub-Systems

- GFS
- Map-Reduce
- Pig
- Nova

datum X

datum Y

provenance of X?

metadata queries
Ibis Project

Benefits:
- Provide uniform view to users
- Factor out metadata management code
- Decouple metadata lifetime from data/subsystem lifetime

Challenges:
- Overhead of shipping metadata
- Disparate data/processing granularities
What’s Hard About Multi-Granularity Provenance?

• **Inference:** Given relationships expressed at one granularity, answer queries about other granularities *(the semantics are tricky here!)*

• **Efficiency:** Implement inference without resorting to materializing everything in terms of finest granularity (e.g. cells)
Next: INSPECTOR GADGET

Debugging aides:

- Before: example data generator
- During: instrumentation framework
- After: provenance metadata manager
Motivated by User Interviews

• Interviewed 10 Yahoo dataflow programmers (mostly Pig users; some users of other dataflow environments)

• Asked them how they (wish they could) debug
## Summary of User Interviews

<table>
<thead>
<tr>
<th># of requests</th>
<th>feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>crash culprit determination</td>
</tr>
<tr>
<td>5</td>
<td>row-level integrity alerts</td>
</tr>
<tr>
<td>4</td>
<td>table-level integrity alerts</td>
</tr>
<tr>
<td>4</td>
<td>data samples</td>
</tr>
<tr>
<td>3</td>
<td>data summaries</td>
</tr>
<tr>
<td>3</td>
<td>memory use monitoring</td>
</tr>
<tr>
<td>3</td>
<td>backward tracing (provenance)</td>
</tr>
<tr>
<td>2</td>
<td>forward tracing</td>
</tr>
<tr>
<td>2</td>
<td>golden data/logic testing</td>
</tr>
<tr>
<td>2</td>
<td>step-through debugging</td>
</tr>
<tr>
<td>2</td>
<td>latency alerts</td>
</tr>
<tr>
<td>1</td>
<td>latency profiling</td>
</tr>
<tr>
<td>1</td>
<td>overhead profiling</td>
</tr>
<tr>
<td>1</td>
<td>trial runs</td>
</tr>
</tbody>
</table>
Our Approach

• **Goal:** a programming framework for adding these behaviors, and others, to Pig

• **Precept:** avoid modifying Pig or tampering with data flowing through Pig

• **Approach:** perform Pig script rewriting – insert special UDFs that look like no-ops to Pig
Pig w/ Inspector Gadget
Example:

**Crash Culprit Determination**

**Phases 1 to n-1:** record counts

**Phase n:** records

**Phases 1 to n-1:** maintain count lower bounds

**Phase n:** maintain last-seen records
Example:

Forward Tracing

IG coordinator

report traced records to user

load

filter

load

IG agent

join

IG agent

group

IG agent

count

IG agent

store

tracing instructions

traced records
Flow

end user

result

application

IG driver library

launch instrumented dataflow run(s)

raw result(s)

IG coordinator

load

load

IG agent

IG agent

filter

IG agent

IG agent

join

IG agent

store

dataflow engine runtime

dataflow program + app. parameters

app. parameters

+ dataflow program

Flow
# Agent & Coordinator APIs

## Agent Class
- `init(args)`
- `tags = observeRecord(record, tags)`
- `receiveMessage(source, message)`
- `finish()`

## Coordinator Class
- `init(args)`
- `receiveMessage(source, message)`
- `output = finish()`

## Agent Messaging
- `sendToCoordinator(message)`
- `sendToAgent(agentId, message)`
- `sendDownstream(message)`
- `sendUpstream(message)`

## Coordinator Messaging
- `sendToAgent(agentId, message)`
## Applications Developed Using IG

<table>
<thead>
<tr>
<th># of requests</th>
<th>feature</th>
<th>lines of code (Java)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>crash culprit determination</td>
<td>141</td>
</tr>
<tr>
<td>5</td>
<td>row-level integrity alerts</td>
<td>89</td>
</tr>
<tr>
<td>4</td>
<td>table-level integrity alerts</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>data samples</td>
<td>97</td>
</tr>
<tr>
<td>3</td>
<td>data summaries</td>
<td>130</td>
</tr>
<tr>
<td>3</td>
<td>memory use monitoring</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>backward tracing (provenance)</td>
<td>237</td>
</tr>
<tr>
<td>2</td>
<td>forward tracing</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>golden data/logic testing</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>step-through debugging</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>latency alerts</td>
<td>168</td>
</tr>
<tr>
<td>1</td>
<td>latency profiling</td>
<td>136</td>
</tr>
<tr>
<td>1</td>
<td>overhead profiling</td>
<td>124</td>
</tr>
<tr>
<td>1</td>
<td>trial runs</td>
<td>93</td>
</tr>
</tbody>
</table>
Rest of talk: IG DETAILS

- Semantics under parallel/distributed execution
- Messaging & tagging implementation
- Limitations
- Performance experiments
- Related work
Parallel/Distributed Execution

- load
- filter
- split
- median
- count
- store

- group
- split
- median
- count
- store

- stage (e.g. map)
- stage (e.g. reduce)
Messaging Details

• Semantics:

<table>
<thead>
<tr>
<th>Message Request</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendToCoordinator(message)</td>
<td>asynchronous, guaranteed delivery</td>
</tr>
<tr>
<td>sendToAgent(agentId, message)</td>
<td>asynchronous, best-effort delivery</td>
</tr>
<tr>
<td>sendDownstream(message)</td>
<td>“follow the arrows,” guaranteed delivery</td>
</tr>
<tr>
<td>sendUpstream(message)</td>
<td>(same-stage only:) “invert the arrows,” guaranteed</td>
</tr>
</tbody>
</table>

• Implementation:
  – Within-process: shared memory
  – Cross-process: relay through coordinator (coordinator buffers message for recipients that haven’t started yet)
Tagging Implementation

• Uses messaging APIs

• Within-stage:
  – Leverage “iterator model” synchronous pipeline execution
    1. sendDownstream(“tag future outputs with T”); release output record
    2. sendDownstream(“stop tagging”)

• Cross-stage:
  – Leverage Pig operator semantics (group-by, cogroup, join, order-by)
  – Group/cogroup: use group key
  – Join/order-by: use all record fields (back-tags dups!)
Limitations of the IG Approach

• Assumes query optimization nonexistent/disabled

• IG sits on top of Pig, so hard to correlate with lower-level logs/errors

• Crash/re-start results in record being seen by agents multiple times
  – Fortunately, all apps we’ve written can tolerate this, e.g. data only sent in finish(); rely on idempotence

• Tagging implementation not scalable

• Tagging implementation relies on Pig details
Performance Experiments

- 15-machine Pig/Hadoop cluster (1G network)
- Four dataflows over a small web crawl sample (10M URLs):

<table>
<thead>
<tr>
<th>Dataflow Program</th>
<th>Early Projection Optimization?</th>
<th>Early Aggregation Optimization?</th>
<th>Number of Map-Reduce Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct Inlinks</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Frequent Anchortext</td>
<td>Y</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Big Site Count</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Linked By Large</td>
<td>N</td>
<td>Y</td>
<td>2</td>
</tr>
</tbody>
</table>
Dataflow Running Times

![Bar chart showing running times for different scripts and dataflow operations.](chart.png)
Debugging aides:

- Before: example data generator
- During: instrumentation framework
- After: provenance metadata manager
Related Work

- **Pig**: DryadLINQ, Hive, Jaql, Scope, *relational query languages*

- **Nova**: BigTable, CBP, Oozie, Percolator, *scientific workflow, incremental view maintenance*

- **Dataflow illustrator**: [Mannila/Raiha, PODS’86], *reverse query processing, constraint databases, hardware verification & model checking*

- **Inspector gadget**: XTrace, *taint tracking, aspect-oriented programming*

- **Ibis**: Kepler COMAD, ZOOM user views, *provenance management for databases & scientific workflows*
Collaborators

Shubham Chopra
Anish Das Sarma
Alan Gates
Pradeep Kamath
Ravi Kumar
Shravan Narayanamurthy

Olga Natkovich
Benjamin Reed
Santhosh Srinivasan
Utkarsh Srivastava
Andrew Tomkins