Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows

Christopher Olston
Yahoo! Research
701 First Ave.
Sunnyvale, CA
olston@yahoo-inc.com

Benjamin Reed
Yahoo! Research
701 First Ave.
Sunnyvale, CA
breed@yahoo-inc.com

ABSTRACT

We demonstrate a novel dataflow introspection framework called *Inspector Gadget*, which makes it easy to create custom monitoring and debugging add-ons to an existing dataflow engine such as Pig. The framework is motivated by a series of informal user interviews, which revealed that dataflow monitoring and debugging needs are both pressing and diverse. Of the 14 monitoring/debugging behaviors requested by users, we were able to implement 12 in Inspector Gadget, in just a few hundred lines of (Java) code each.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Most data processing scenarios consist of data items being routed through a network of data transformation operators. Such *dataflows* are sometimes compiled from declarative query expressions (e.g. SQL), and are sometimes programmed more directly (e.g. extract-transform-load (ETL) pipelines [5], data visualization builders [7], data stream processing engines (some approaches) [1], web mashup tools [8], and dataflow frameworks for map-reduce [6]). One of the reasons to program dataflows directly is to exert more control over, and have a better understanding of, their run-time behavior, e.g. to predictably satisfy service-level agreements (SLAs), or to facilitate debugging.

Unfortunately, real-world dataflow implementations often fail to achieve run-time visibility and ease of debugging, for two main reasons:

- **Difficulty in providing useful status and error messages.** Status and error reporting is essential for usability of complex systems. Yet it is very challenging to ensure that status/error messages achieve the right balance between informativeness and brevity, and are expressed at the right level of abstraction for users to comprehend. Even relatively mature systems struggle with this issue. The problem is exacerbated in multi-layer systems (e.g. workflow middleware; Oozie-Pig-Hadoop [2, 3, 6]), due to the difficulties in linking status and error messages across layers, and in translating them from lower-layer terminology to upper-layer terminology.
- **Expense of retaining intermediate data and capturing provenance.** For efficiency reasons, many intermediate results (data flowing between pairs of operators) are not materialized, making post-hoc examination of the data processing sequence difficult. There is a great deal of published work on capturing and querying *data provenance* [4], but that only solves a subset of users' debugging needs (see Table 1), and it presents the dilemma of balancing forward processing efficiency against post-hoc debugging capability.

Implementors of dataflow processing engines have enough on their hands ensuring correctness, achieving good scalability and performance, and supporting new applications—"nice-to-have" usability enhancements like informative error messages and data provenance tend (sadly) to be perpetually pushed to future release cycles.

At the same time, there is great demand from users for dataflow monitoring and debugging capabilities. We conducted informal interviews of ten Yahoo employees from diverse product groups that use dataflow programming. Many of them use Pig [6], but a few use other proprietary Yahoo dataflow tools. In the interviews we asked what monitoring and debugging capabilities would be helpful. The responses are summarized in Table 1, which shows the distinct capabilities mentioned, ranked by the number of interviewees that mentioned the capability (out of 10).

1.1 Inspector Gadget

Motivated by these user interviews, we have developed a framework that makes it easy to layer monitoring and debugging capabilities on top of an existing dataflow engine. Our framework, called *Inspector Gadget*, provides abstractions for observing data passing through dataflow edges, tagging pieces of data and viewing tags at downstream observation points, and exchanging messages between pairs of observation points and with a central coordinator node. We have also developed a Pig-specific implementation of Inspector Gadget called *Penny*. Some aspects of Penny are, we believe, generalizable to other dataflow environments (e.g. the messaging implementation), whereas others are very Pig-specific (e.g. parsing and instrumentation of Pig Latin scripts). Using Penny and Inspector Gadget, we have successfully im-

SIGMOD'11, June 12–16, 2011, Athens, Greece.
ACM 978-1-4503-0661-4/11/06.

# of users	desired capability	description	lines of code
7	crash culprit determination	Determine which data record and/or processing operator triggered a crash.	141
5	row-level integrity alerts	Throw an alert whenever a record violates a given predicate (e.g. field X not null, numerical field Y $\geq$ 0, date-stamp field Z $\leq$ today).	89
4	table-level integrity alerts	Throw an alert whenever an intermediate data set violates a given predicate (e.g. cardinality $>$ 0).	99
4	data samples	Show a few samples of data on each dataflow edge, as a sanity check of the dataflow semantics and to spot fishy data (e.g. a column filled with null values).	97
3	data summaries	Compute a statistical summary (e.g. a histogram) of data values on a particular dataflow edge, and perhaps automatically compare against histograms from previous dataflow runs (on the same or related data) to spot sudden data distribution changes that might indicate a processing error.	130
3	memory use monitoring	Monitor the memory used for materializing intermediate data sets, including custom state maintained by user-defined functions.	N/A
3	backward tracing	Find the chain of input and intermediate records that led to a given output record.	237
2	forward tracing	Find the chain of intermediate and output records that stem from a given input record.	114
2	golden data/logic testing	Given a set of "golden" input/output record pairs that are known to be correct, or a function known to contain correct logic for transforming an input record into an output record, compare the dataflow input/output data against the golden pairs.	200
2	step-through debugging	Set breakpoints and perform step-through debugging of user-defined functions running on remote "cloud" nodes.	N/A
2	latency alerts	Throw an alert if one record takes much longer to process through a particular operator than the average record (e.g. the record contains a very large nested data set to be processed, or induces a large number of interactions with an external service).	168
1	latency profiling	Show the distribution of record processing latencies, perhaps in relation to record-level SLAs (e.g. certain data items related to online advertising must be processed prior to their ad campaign start date).	136
1	overhead profiling	Report the per-operator breakdown of total dataflow execution time.	124
1	trial runs	Run the dataflow on a small sample of the input data, as a quick sanity check to see whether it crashes or succeeds, and if it succeeds whether reasonable-looking output is produced.	93

Table 1: Monitoring and debugging capabilities requested by users.

plemented 12 of the 14[1] behaviors in Table 1, each in very few lines of Java code (low hundreds), as shown in the right-hand column.

2. PROGRAMMING MODEL

This section briefly describes our Inspector Gadget dataflow monitoring/debugging framework, from the point of view of a user of the framework (i.e. someone who wishes to create a particular monitoring or debugging application).

Inspector Gadget (IG) provides abstractions for inserting *monitor agents* (agents, for short) along dataflow edges to observe data records flowing through. The agents may communicate with each other and/or with a central *coordinator*. The bottom portion of Figure 1 shows how a running dataflow (shown as interconnected ovals) is instrumented with IG monitor agents (small boxes), linked to a coordinator (large box on left). The main data processing flow (the ovals) behaves as normal—from the dataflow engine runtime's point of view the monitor agents behave as no-op functions—and a separate processing and communication plane for the IG application is layered on top.

IG applications supply code that runs inside the monitor agents and coordinator. For example, in a simple (but inefficient) implementation of the **crash culprit determination** application (Table 1), each agent sends a copy of each record it sees to the coordinator; the coordinator keeps track of the latest record sent from each agent, and if a crash occurs the last-received records are flagged as candidate culprits in triggering the crash.

Each IG application has a *driver* module, shown in the top portion of Figure 1, that receives instructions from the end user (e.g. "please provide clues about why my dataflow program crashes"), configures and launches one or more IG-instrumented dataflow runs, and composes a response to the user (e.g. "your dataflow program crashes when it tries to process records X, Y and Z").

[1]The only two behaviors from Table 1 that do not fit neatly into our framework, and which we did not implement, are: (1) memory use monitoring and (2) step-wise debugging. Obtaining a detailed breakdown of memory usage would require access to the internal data structures of the dataflow system and/or user-defined functions, which is expressly outside the capabilities of our framework. Step-wise debugging can be accomplished without our framework by attaching a conventional debugger to a remote process, which poses logistical and security challenges unrelated to our framework.

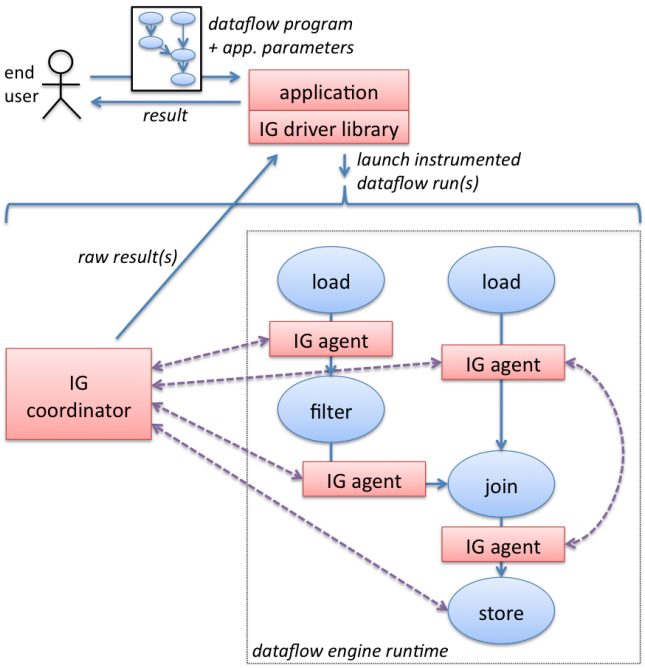

Figure 1: Instrumented dataflow.

3. EXAMPLE APPLICATION

Many applications, such as **integrity alerts**, **data samples** and **data summaries** have a very simple structure: agents gather information about records they observe and report that information to the coordinator, which relays or aggregates it. We describe a less trivial example, which is our iterative implementation of **crash culprit determination**. It invokes `launch()` n times, each time narrowing the scope of possible records "responsible" for the crash.[2] Our implementation handles parallelism in the dataflow (multiple agent instances associated with a given dataflow edge), but for simplicity we describe the non-parallel case.

Each monitor agent maintains a record counter, and reports the count (and the latest record) to the coordinator at regular intervals (determined by parameter `skip`) whenever the count exceeds a second parameter called `start`. The coordinator maintains a lower bound on the record number reported from each monitor agent. Each iteration narrows the reporting interval (e.g. `skip` = `skip`/10), while increasing `start` to the lower bound determined in the prior iteration. For example, if record # 2381 on a particular dataflow edge is responsible for the crash, the iterations might be: (1) `start` = 0, `skip` = 1000; (2) `start` = 2000, `skip` = 100; (3) `start` = 2300, `skip` = 10; (4) `start` = 2380, `skip` = 1. In the final iteration, the content of the 2381st record is reported back to the user.

We ran our crash culprit determinator on a real Pig Latin script that caused Pig to crash with error message "ERROR 2106: Error while computing count in COUNT" followed by some detailed information that nonetheless left the exact cause of the crash a mystery (for one thing, the offending record was not printed). The script was attempting to count the number of incoming links to a particular web site, starting with a large data set of the form (url, site, inlinks, ...), by first counting the number of items in each "inlinks" field, and then grouping by site and summing the per-url inlink counts to produce per-site counts (this script's goal was not to count *unique* links). Our crash culprit determinator produced from among millions of input records a handful of candidate crash culprits. Upon inspection one of those records turned out to contain a `null` value in its inlinks field, which turned out to be the cause of the crash. The problem was resolved by adding a filter expression that bypasses `null` values.

[2]Our implementation assumes that the crash is being caused by a particular, problematic record. It further assumes that the order in which records are read, and the way in which they are partitioned among stage instances, are both deterministic (these are reasonable assumptions, e.g. they hold for Pig/Hadoop).

4. DEMO DESCRIPTION

Our demo will showcase several of the applications we have built using Inspector Gadget, including the crash culprit determination application described in Section 3. The applications will be run on several realistic web analytics Pig scripts over real web data, on an eight-node Hadoop cluster. For example we will recreate the crash anecdote described at the end of Section 3 and show what our crash culprit determination application does in that case. To keep the demo interactive the web data set will be trimmed so that each application run only takes a few seconds.

We will also show the code used to write each application in our IG framework, which fits on one screen and showcases the simplicity and power of our framework for building dataflow monitoring and debugging applications.

5. REFERENCES

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.

[2] Apache. Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`.

[3] Apache. Oozie: Hadoop workflow system. `http://issues.apache.org/jira/browse/HADOOP-5303`.

[4] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[5] Extract, transform, load. `http://en.wikipedia.org/wiki/Extract,_transform,_load`.

[6] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The Pig experience. In *Proc. VLDB*, 2009.

[7] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *Proc. VLDB*, 1993.

[8] Yahoo!, Inc. Pipes: Rewire the web. `http://pipes.yahoo.com`.